Delphi 6

Programmer's Choice

Pearson Education

ADDISON-WESLEY

Elmar Warken

Delphi 6

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen
eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Abbildungen und Texten wurde mit größter
Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben
und deren Folgen weder eine juristische Verantwortung noch
irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und
Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der
Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten
ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden,
sind gleichzeitig eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem
und recyclingfähigem PE-Material.

5 4 3 2 1

05 04 03 02 01

ISBN 3-8273-1773-8

© 2001 by Addison-Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München / Germany
Alle Rechte vorbehalten
Einbandgestaltung: Christine Rechl, München
Titelbild: Cotoneaster, Felsenmispel © Karl Blossfeldt Archiv –
Ann und Jürgen Wilde, Zülpich / VG Bild-Kunst Bonn, 2001.
Lektorat: Frank Eller, feller@pearson.de
Korrektorat: Simone Burst, Großberghofen
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Bercker, Kevelaer
Printed in Germany

I Inhalt

Vorwort

Nachdem Delphi 5 im Jahr 1999 noch ganz im Zeichen der allgemeinen Internet-Euphorie stand und Borland (damals für eine gewisse Zeit unter dem Namen Inprise firmierend) von den »Webvolutionaries« sprach, konzentrierte man sich im Folgejahr auf die inneren Kräfte und entwickelte jene neuen Produkte, welche nun in der ersten Hälfte von 2001 »geerntet« wurden: Im Delphi-Bereich sind dies das völlig neue Kylix für die Delphi-Entwicklung unter Linux und Delphi 6 für die Entwicklung von Windows-Anwendungen.

Delphi 6 beweist einmal mehr, wie mächtig das technologische Fundament der VCL und der Two-Way-Tools ist, auf dem Delphi schon seit sieben Jahren basiert: Erneut ist es Borland gelungen, die aktuellsten Technologien wie XML und SOAP so in die Komponentenbibliothek und die IDE zu integrieren, dass der Umgang damit so einfach wird, wie man es unter Delphi von anderen Technologien bereits gewohnt ist: Entwickler brauchen sich nicht um die Einhaltung von komplexen Protokollen und Standards zu kümmern, sondern können sich darauf verlassen, dass diese implizit von Delphi eingehalten werden. Der Entwickler hat folglich mehr Energie für die Lösung des eigentlichen Problems übrig[1].

Die Unterstützung aktueller Standards und die hervorragende Erweiterbarkeit von Delphi wird gerne auch unter dem Begriff der »Zukunftssicherheit/-fähigkeit« genannt. Rückblickend auf die »vergangene Zukunft« kann man sehen, dass sich die Zukunftssicherheit von Delphi immer wieder bestätigt hat. Da aber die »zukünftige Zukunft« niemals vorhersehbar sein kann, lege ich Wert auf die Feststellung, dass Delphi vor allem außerordentlich *gegenwartsfähig* ist: Es steht damit *jetzt* ein einzigartiges Werkzeug zur Verfügung, das für viele Entwickler ideal geeignet ist, wobei natürlich nur jeder Entwickler-Mensch selbst entscheiden kann, ob das für sie oder ihn zutrifft.

---

1 Beispiele für die erwähnten »anderen Technologien« sind ActiveX (Entwicklung von Steuerelementen, Automations-Servern und -Clients), der Datenbankzugriff oder ganz allgemein die Entwicklung der Programm-Oberfläche, bei der die VCL dem Entwickler ja schon seit Delphi 1 die komplizierte Programmierung des Windows-API abnimmt.

Empfehlenswerte Vorkenntnisse

Dieses Buch behandelt neben vielen spezielleren Themen wie der Entwicklung von Datenbank- und Web-Server-Anwendungen auch alle wesentlichen Bereiche von Delphi, die für die allgemeine Anwendungs- und Komponenten-Entwicklung erforderlich sind. Was die Entwicklung mit Delphi betrifft, ist es somit selbst für Einsteiger geeignet. Die Darstellung der Sprache Object Pascal betont besonders die neueren Sprachmerkmale, und da die grundlegenden Spracheigenschaften nur kurz besprochen werden, sollten Sie bereits elementare Programmiererfahrungen besitzen, bevor Sie sich dem ersten Kapitel zuwenden. Diese Erfahrungen können Sie beispielsweise bei der Programmierung mit Java, Visual Basic, Turbo Pascal, Perl, Datenbank-Programmiersprachen, C oder C++ oder natürlich bei der Durcharbeitung eines Delphi-Einsteiger-Buchs erhalten haben. Das Buch enthält Abschnitte, die speziell für Umsteiger vorgesehen sind, so zum Beispiel das Kapitel 2.1, das einige Object-Pascal-Konzepte einführt, die in manchen Basic- oder Skriptsprachen nicht anzutreffen sind. Darüber hinaus ist vor allem das erste Kapitel bis einschließlich Kapitel 1.8 so geschrieben, dass Sie auch dann alles verstehen können, wenn Sie Pascal nicht kennen. Dies ändert sich in Kapitel 3, von dem an die Sprache Object Pascal als weitgehend bekannt vorausgesetzt wird.

Das Buch und die Delphi-Versionen

Nicht alle Entwickler verwenden immer die aktuellste Delphi-Version – verständlicherweise, wenn sie auch bei den hervorragenden Vorgängerversionen noch aus dem Vollen schöpfen können. Auch dieses Buch ist nicht nur für Neubesitzer von Delphi 6 gedacht, sondern auch für Benutzer von Vorversionen, die sich das Update für später aufheben oder noch gar nicht daran denken. Im Text wird im Allgemeinen erwähnt, ab welcher Delphi-Version eine bestimmte Funktion verfügbar ist. Einige Besonderheiten älterer Delphi-Versionen, die in den neuen Versionen obsolet geworden sind, werden allerdings nicht mehr im Buchtext beschrieben, sind aber als Anmerkungen zu den entsprechenden Kapiteln auf der CD zu finden.

Manche Beispielprogramme haben natürlich höhere Delphi-Versionen als Voraussetzung. Insbesondere die aktuelle Version des TreeDesigners und diejenigen Datenbank-Anwendungen, welche die neue dbExpress-Bibliothek verwenden, lassen sich zunächst nur unter Delphi 6 kompilieren. Doch finden Sie auf der CD auch eine Version des TreeDesigners, die noch für frühere Delphi-Versionen geeignet ist: Es handelt sich um den TreeDesigner 3.0, der in eigenen »verkleinerten« Versionen für alle Delphi-Versionen bis Delphi 1 vorliegt. Der Delphi-5-Version des TreeDesigner 3.0 fehlen im Vergleich mit den in diesem Buch besprochenen Teilen lediglich die XML-Funktionen und die anpassbaren Symbolleisten.

Die verschiedenen Delphi-Ausgaben

Bezüglich der unterschiedlichen Ausgaben von Delphi gilt, dass selbst die kleinste Version (Personal-Ausgabe; bis Delphi 5: Standard-Ausgabe) denselben hochentwickelten Compiler, dieselbe Object-Pascal-Version und dieselbe VCL verwendet wie die vielfach teurere Enterprise-Ausgabe. Daher können Sie schon mit dieser Einstiegsversion alle Beispielprogramme auf der CD übersetzen, ausgenommen die Datenbankbeispiele, fast alle Beispiele von Web-Server-Anwendungen und eine spezielle mit dem Aktionsmanager arbeitende Version des TreeDesigners (die »normale« TreeDesigner-Version funktioniert natürlich auch mit der Personal-Ausgabe; einige Datenbank-Beispiele funktionieren mit den Standard-Ausgaben von Delphi 4 und früheren Versionen).

Allerdings ist es mit den Personal/Standard-Versionen nicht möglich, alle Beispiele »nachzubauen«. Hierzu fehlen für einige COM-Beispiele der Typenbibliotheks-Editor (verwendet in Kapitel 8.7) und der Experte zur Generierung von ActiveX-Controls (Kapitel 6.8). Für den Einsatz des OpenTools-APIs in Anhang A fehlen die als Dokumentation dienenden Quelltexte der entsprechenden Units. Weitere Einschränkungen der Personal/Standard-Versionen wie etwa bei der Funktionalität von Editor und Debugger wirken sich nicht auf die Verwendung der Beispiele aus.

Kapitel-Übersicht

Im Folgenden finden Sie einen kurzen Überblick über die acht Kapitel des Buchs:

- Kapitel 1 beschreibt die Entwicklungsumgebung (IDE) und den grundlegenden Aufbau sowie die Funktionsweise einer Delphi-Anwendung. Sie können wählen zwischen einem Schnelleinstieg, der direkt bei einem kleinen Beispielprojekt beginnt, und einer ausführlicheren Einführung, die auch Hintergründe erläutert.
- Kapitel 2 widmet sich der Programmiersprache Object Pascal. Obwohl sein Schwerpunkt auf der objektorientierten Programmierung liegt, werden auch die Grundlagen von Pascal an geeigneter Stelle so zusammengefasst, dass sie sich auch Umsteiger aneignen können.
- Kapitel 3 beschäftigt sich intensiv mit der VCL und reicht von der Untersuchung wichtiger VCL-Interna über die Beschreibung von Komponenten bis hin zu konkreten Beispielprogrammen. Besonderer Wert wurde dabei auf allgemeine Klassen wie *TComponent*, *TControl* und *TWinControl* gelegt, denn wenn Sie diese kennen, kennen Sie bereits einen großen Teil der Eigenschaften aller speziellen visuellen Komponenten.
- Kapitel 4 setzt die in Kapitel 3 begonnene Behandlung der VCL fort, beschäftigt sich aber nicht mehr in erster Linie mit den visuellen Komponenten, sondern mit nicht-visuellen Klassen wie etwa Listenklassen und Klassen für die Windows-Registry, für Menüs und Threads sowie für die Grafikausgabe.

- Kapitel 5 demonstriert anhand der Beispielanwendung *TreeDesigner 3.5* unter anderem die Entwicklung größerer, dokumentbasierter MDI-Anwendungen mit moderner Benutzerschnittstelle, Dokument-View-Konzept, XML-Unterstützung und mit der Ausgabe geräteunabhängiger Grafik auf Bildschirm und Drucker. Die CD-ROM enthält bereits einige mit dem TreeDesigner angefertigte Hierarchiegrafiken der VCL.
- Thema von Kapitel 6 ist die Entwicklung eigener Komponenten. Das Kapitel beschreibt die hinter Delphis Komponenten stehenden Konzepte und demonstriert die Entwicklung eigener Komponenten anhand von nützlichen Beispielen wie einer History-Kombobox, einer Echtfarben-Farbpalette und einem Tastenkürzeleditor für Menüs und Aktionslisten.
- Kapitel 7 befasst sich mit Datenbankanwendungen, unter anderem auch mit der neuen Zugriffsmethode *dbExpress*. Es erläutert den Aufbau von Datenbanken, die Komponenten der VCL, über die Sie diese ansprechen, sowie die allgemeine Funktionsweise einer Datenbankanwendung in Delphi und es zeigt verschiedene Beispiele der Programmierung mit Feldern und Indizes, SQL-Anfragen und Datenbank-Updates.
- Im Mittelpunkt von Kapitel 8 steht der Daten- und Code-Austausch mit anderen Anwendungen oder Bibliotheken, z.B. über die Zwischenablage, mit Hilfe von DLLs und OLE. Der größte Teil des Kapitels ist dem großen Bereich des Component Object Models (COM) gewidmet, von der Nutzung einfacher COM-Objekte bis zum Selbstentwickeln eines DCOM-Servers am Beispiel des TreeDesigners aus Kapitel 5. Der letzte Teil des Kapitels befasst sich schließlich mit Web-Server-Anwendungen, speziell mit den Delphi-Features *WebBroker*, *WebSnap* und mit der Unterstützung für Web-Dienste.
- Anhang A gibt einen Überblick über Delphis OpenTools-API, über das Sie die Delphi-IDE mit eigenen Tools erweitern können. Während zur Kompilierung der Beispielexperten die Professional-Version erforderlich ist, können Sie die fertigen Experten von der CD auch in der Standard-Version installieren, wodurch diese übrigens durch eine Alternative zum CodeExplorer der Professional-Ausgabe erweitert wird.

Rezeptverzeichnis *R175*

R-Nummern in Überschriften wie die oben gezeigte beziehen sich auf die Nummern des Rezeptverzeichnisses in Anhang C. Dieses Verzeichnis gibt eine Übersicht über viele der in diesem Buch beschriebenen Lösungen und weist dabei eine zum Inhaltsverzeichnis alternative Gliederung auf. Es beschränkt sich auf die nach meiner Einschätzung interessantesten Rezepte und verzichtet auf den Verweis auf grundlegende Beschreibungen, die schon durch das Inhaltsverzeichnis schnell gefunden werden kön-

nen. Aufgrund dieses strengeren Rezept-Auswahlverfahrens enthält das aktuelle Verzeichnis weniger Einträge als das des Vorgänger-Buchs, obwohl wieder einige neue Einträge hinzugekommen sind.

Feedback und Homepage

Ihr Feedback zu diesem Buch, auch Fragen und Korrekturhinweise, sind unter der Adresse *leserkontakt@ewlab.de* herzlich willkommen. Im Internet bewohnt dieses Buch eine Homepage unter der Adresse *http://ewlab.de/delphi6/buch-homepage.html*. Gegebenenfalls werden dort Fehlerkorrekturen, Erweiterungen der Beispielprogramme oder Antworten auf häufig gestellte Fragen veröffentlicht. Und nun wünsche ich Ihnen nützliche Erkenntnisse aus diesem Buch sowie viel Spaß und viel Erfolg mit Delphi.

Elmar Warken, Bonn, im September 2001

1 Die visuelle Programmierumgebung

Dieses Kapitel soll Ihnen nicht nur einen praktischen Einstieg bzw. Umstieg in Delphis Programmierumgebung ermöglichen, sondern auch ein fundiertes Grundwissen über die Arbeitsweise der neuen Umgebung und die Funktionsweise von Delphi-Anwendungen vermitteln. Daher wechselt sich die praktische Einführung mit Grundlagenbeschreibungen ab.

Damit Sie mit diesem Kapitel auch einen Schnelleinstieg finden können, wurde das Kapitel so organisiert, dass Sie es auf zwei verschiedenen Routen durchqueren können (das Buch begrüßt die *Two-Way-Tools* von Delphi also mit einem *Two-Way-Tutorial*):

- Die große Route: Sie lesen das Kapitel von Anfang bis Ende durch und überspringen dabei nur wenige Themen, die Sie später nachschlagen wollen. Auf diese Weise werden Ihre praktischen Erfahrungen mit Delphi von umfangreichem Grundlagenwissen untermauert und Sie erhalten gleichzeitig einen guten Überblick über die gesamte Delphi-IDE, ohne dass Sie sich mit jedem einzelnen Menüpunkt beschäftigen müssen.
- Die kleine Route: Sie beginnen sofort mit Kapitel 1.3. Nachdem Sie dieses über den Entwicklungszyklus für eine Delphi-Anwendung informiert hat, finden Sie dort einen kurzen Überblick über die Entwicklung eines Beispielprogramms. Sie besuchen daraufhin die in diesem Überblick beschriebenen Stationen des Kapitels, um die vollständige Entwicklung dieses Programms beobachten oder mitmachen zu können.

Sie können beide Routen auch mit älteren Versionen von Delphi nachvollziehen: Zwar beziehen sich sämtliche Abbildungen auf Delphi 6, die Bedienung der verschiedenen Delphi-Versionen ist jedoch sehr ähnlich. Falls sich einmal ein Menüpunkt oder ein Dialogfeld zwischen den Versionen unterscheiden sollte, weist Sie zumindest eine Fußnote auf diesen Unterschied hin. Alle in diesem Kapitel entwickelten (und auf der CD befindlichen) Programmversionen können mit allen Delphi-Versionen ausgeführt werden, mit Ausnahme der in Kapitel 1.9.4 und 1.9.5 behandelten, für die mindestens Delphi 3 erforderlich ist.

Wir wenden uns nun den wichtigen Grundlagen zu, auf denen jede Delphi-Anwendung aufbaut – Willkommen auf der großen Route! Wenn Sie Delphi zum ersten Mal

starten, zeigt es Ihnen ein leeres Formular, das Sie durch Hinzufügen von Komponenten zur Oberfläche Ihres Programms ausbauen können. An der Oberfläche dreht sich also alles um Formulare und Komponenten. Für die Entwicklung von Anwendungen ist jedoch die grundsätzliche interne Funktionsweise einer Delphi-Anwendung am wichtigsten, weshalb wir auch genau dort beginnen.

1.1 Delphi-Anwendungen sind ereignisorientiert

Moderne Anwendungen auf grafischen Oberflächen arbeiten ereignisorientiert, das heißt, sie laufen nicht nach einer festgelegten Reihenfolge ohne Eingriffsmöglichkeiten für den Benutzer ab wie etwa eine Kommando-Datei für die Windows-Eingabeaufforderung.

Eine optimal entworfene Delphi-Anwendung tut nichts anderes, als auf Ereignisse (Events) zu reagieren. Diese Ereignisse können vom Betriebssystem, von anderen Anwendungen oder von der Delphi-Anwendung selbst (beispielsweise von der Komponentenbibliothek VCL) erzeugt werden.

Die Ereignisse haben zunächst je nach ihrem Entstehungsort unterschiedliche Erscheinungsbilder: Von Windows generierte Ereignisse erreichen die Anwendung beispielsweise in Form von Nachrichten, und Ereignisse der VCL hängen oft mit *virtuellen Methoden* zusammen. Als Verwender von Delphi macht dies für Sie jedoch keinen Unterschied, denn die VCL formt fast alle Nachrichten in gleichartige (aber nicht identische) *Events* um, die Sie im Objektinspektor sehen und bearbeiten können.

Sie kommen zwar normalerweise mit den von der VCL bereitgestellten Events aus, jedoch steht Ihnen unter Delphi auch der Weg zu den ursprünglichen Windows-Nachrichten offen. Wenn Sie beispielsweise selbst Komponenten programmieren oder auf Windows-Nachrichten reagieren wollen, für die es im Objektinspektor kein Event gibt, können Sie andere Möglichkeiten der Ereignisbearbeitung nutzen. Eine detaillierte Darstellung des Nachrichtenflusses in einer Delphi-Anwendung finden Sie in Kapitel 3.1.4.

1.1.1 Zu jedem Anlass ein Ereignis

Falls Sie bisher nicht ereignisorientiert programmiert haben und nicht davon überzeugt sind, dass Ereignisse genügen, um Ihre Anwendung zu steuern, stellen Sie sich alle Aktionen vor, die Ihnen bekannte Windows-Anwendungen ausführen und die Ihre eigene Anwendung möglicherweise ausführen könnte. Jede dieser Aktionen muss durch irgendetwas ausgelöst werden. Dieser Auslöser wird in ereignisgesteuerten Anwendungen als *Ereignis* bezeichnet. Jeder Auslöser bzw. jedes Ereignis sollte in einer der Kategorien zu finden sein, die im Folgenden aufgezählt sind.

- *Befehlsereignisse*: Die wichtigsten Operationen wie das Öffnen und Speichern von Dateien oder das Aufrufen von Dialogboxen werden vom Anwender über Menüs oder Schalter gestartet. Windows sendet Ihrer Anwendung eine spezielle Nachricht, wenn der Anwender auf diese Art einen Befehl aufgerufen hat. Die VCL wandelt diese Nachricht in ein normales Event um – Events für Befehlsereignisse haben meistens den Namen *OnClick*, der darauf zurückzuführen ist, dass der Benutzer ein Element (Menü, Schalter etc.) angeklickt hat. Es spielt jedoch keine Rolle, ob das Element tatsächlich angeklickt oder mit der Tastatur ausgewählt wurde.

- *Eingabeereignisse*: Weitere Ereignisse sind die Eingaben, die der Benutzer mit der Maus oder der Tastatur vornimmt, doch nicht alle dieser Eingaben erreichen Ihre Anwendung. So wird z.B. die gesamte Menüsteuerung per Tastatur von Windows übernommen. Sobald der Benutzer einen Menüpunkt aufruft, erhält die Anwendung eine Befehlsnachricht des oben genannten Typs. Eingabeereignisse, die Sie vielleicht direkt bearbeiten möchten, sind Aktionen der Maus in einem Zeichenbereich Ihrer Anwendung. Zu den Mausereignissen gehören nicht nur Klicks, sondern jede kleine Bewegung der Maus ist ein eigenes Ereignis, das Sie bearbeiten können. Auch Tastatureingaben in den Arbeitsbereich Ihres Fensters bzw. Formulars gehören zu den Eingabeereignissen.

- *Fensterereignisse*: Um beim Öffnen und Schließen des Fensters (bzw. beim Starten und Beenden des Programms) bestimmte Initialisierungs- oder Freigabeaktionen durchführen zu können, müssen Sie die zum Öffnen und Schließen gehörenden Ereignisse abfangen.

- *Timer-Ereignisse*: Soll Ihr Programm beispielsweise in regelmäßigen Abständen automatische Sicherheitskopien anlegen, kann es hierzu sicher keine Eingaben des Benutzers als auslösendes Ereignis verwenden. Für derartige Zwecke finden Sie in der Komponentenpalette eine Timer-Komponente, deren *OnTimer*-Events in frei wählbaren konstanten Intervallen generiert werden.

- *Andere Systemereignisse*: Im Zusammenhang mit der Kommunikation zwischen verschiedenen Anwendungen sowie zwischen Windows und einer Anwendung kommt es ebenfalls zu vielen Ereignissen: Wenn Sie z.B. in der Systemsteuerung eine Systemfarbe ändern, informiert Windows alle Anwendungen darüber, damit diese ihre Farben entsprechend anpassen können; Drag&Drop-Operationen mit dem Windows-Explorer basieren ebenfalls auf Nachrichten.

- Die Menge der Ereignisse ist keineswegs festgelegt: Unter den Events, die Sie im Objektinspektor aufgelistet finden, sind auch einige, die nicht auf Ereignisse der grafischen Oberfläche zurückzuführen sind, sondern direkt von der VCL generiert werden (z.B. alle Events der Datenbankkomponenten, wie sie etwa anlässlich der verschiedensten Datenbankoperationen entstehen). Wenn Sie eigene Komponenten entwickeln, können Sie selbst neue Events definieren.

Selbst eine einfache DOS-Stapeldatei können Sie in dieses Ereignisschema pressen. Danach ist die Stapeldatei ein Programm, das nur auf das Ereignis, gestartet zu werden, reagiert und als Reaktion darauf alle Anweisungen der Reihe nach ausführt und schließlich endet.

Die benutzerabhängigen Ereignisse können jederzeit stattfinden, nur die Timer-Ereignisse müssen Sie in Ihrem Programm selbst anfordern. Es macht jedoch nichts, wenn Sie für ein Ereignis keine spezielle Antwort vorsehen: Das Ereignis wird dann entweder ignoriert oder es wird eine Standardaktion ausgeführt – von Windows oder von der VCL. Die Standardbearbeitung von Ereignissen sorgt z.B. dafür, dass Sie ein leeres Formular, für das Sie noch keine einzige Zeile Code geschrieben haben, wie jedes andere Fenster verschieben, manipulieren und schließen können und dass die Befehle in seinem Systemmenü funktionieren.

Hinweis: Im Folgenden wird die strenge Unterscheidung zwischen Ereignissen im Allgemeinen und den Events, die im Objektinspektor aufgelistet werden, nicht mehr weiter durchgeführt. Die folgenden Abschnitte des ersten Kapitels befassen sich nur noch mit den Events des Objektinspektors, die von nun an ebenfalls als »Ereignisse« bezeichnet werden.

1.1.2 Ereignisse in Delphi

Um einen ersten praktischen Eindruck von Delphis Ereignissen zu bekommen, ohne gleich ein Programm zu schreiben, können Sie die Seite *Ereignisse* des Objektinspektors aufschlagen. Wenn Sie Delphi neu gestartet haben und sich wie in Abbildung 1.1 ein leeres Formular auf dem Bildschirm befindet, dann zeigt der Objektinspektor die Ereignisse, die Sie für dieses Formular bearbeiten können. Dazu gehören beispielsweise die folgenden:

- *OnActivate* tritt auf, wenn das Formular in den Vordergrund geholt wird.
- *OnClick* zeigt Ihnen an, dass der Benutzer des Programms in das Innere des Formulars geklickt hat.
- *OnCreate* benachrichtigt Sie davon, dass das Fenster gerade erzeugt worden ist.
- *OnCloseQuery* tritt auf, bevor das Formular geschlossen wird. Wenn die Datei, die im Formular bearbeitet wird, noch nicht gespeichert wurde, können Sie bei dieser Gelegenheit den Benutzer fragen, ob er die Datei speichern oder ob er den Vorgang abbrechen will.

Jede Komponente, die Sie in das Formular einfügen, verfügt über eine individuelle Auswahl an Ereignissen. Um das auszuprobieren und eine Komponente in das Formular einzufügen, klicken Sie mit der Maus auf irgendein Symbol der Komponenten-

palette und dann einmal in das Formular. Haben Sie sich beispielsweise für die Schalterkomponente (*TButton*) entschieden, so finden Sie im Objektinspektor nun eine andere Ereignisauswahl. Die meisten Ereignisse, wie z.B. *OnClick*, sind auch beim Formular vorhanden, insgesamt haben Schalter aber weit weniger Ereignisse. Die Ereignisse *OnEnter* und *OnExit* finden Sie dagegen nicht im Formular. Sie werden ausgelöst, wenn der Schalter den Tastaturfokus erhält (*OnEnter*) bzw. verliert. (Ein normaler Windows-Schalter zeigt seine Fokussierung dadurch an, dass er den Schaltertext mit einer gepunkteten Linie umrandet. Mit den Pfeiltasten und der Tabulatortaste können Sie in einer Dialogbox normalerweise jeden Schalter ansteuern. Weitere Informationen zum Tastaturfokus liefert Kapitel 3.3.2.)

Während das Konzept der Ereignisse weit verbreitet ist, gibt es doch viele verschiedene Ansätze, es in einer Programmiersprache zu implementieren. Auf Betriebssystem-Ebene werden Ereignisse z.B. häufig in standardisierten Nachrichtenpaketen an eine bestimmte, für die Nachrichtenbearbeitung vorgesehene Programmfunktion übermittelt. Diese Programmfunktion muss nun zunächst einmal feststellen, um welches Ereignis es sich überhaupt handelt, und dann gegebenenfalls andere, spezielle Funktionen aufrufen.

Objektorientierte Systeme verwenden ausgefeiltere Konzepte und können die Ereignisse beispielsweise zunächst nach dem Objekt (z.B. Fenster, Steuerelement), das für das Ereignis zuständig ist, unterscheiden. Dieses Objekt könnte wiederum für jeden Ereignistyp eine eigene Methode bereitstellen.

Vorteile der Ereignisbearbeitungsweise in Delphi

Das auffälligste Merkmal der Ereignisbearbeitung unter Delphi ist sicher der komfortable Verknüpfungsmechanismus von Ereignis und Ereignisbearbeitungsmethode. Um Ereignisse zu bearbeiten, müssen Sie zwar immer noch eine Ereignisbehandlungsmethode (einen Event-Handler) schreiben, Delphi erstellt jedoch automatisch ein Gerüst dafür und erspart Ihnen, an anderen Stellen des Programmcodes für den korrekten Aufruf dieses Ereignisses zu sorgen.

Theoretisch können Sie die Ereignisse in Delphi von beliebigen Objekten bearbeiten lassen, der Objektinspektor ist jedoch darauf beschränkt, die Ereignisse mit Methoden des Formulars zu verknüpfen. Dies ist in der Praxis keine große Einschränkung, bedeutet dafür allerdings eine große Vereinfachung beim Erlernen der Delphi-Programmierung, insbesondere für Entwickler, denen die objektorientierte Programmierung noch fremd ist. Wenn Sie etwa durch die bisherigen Ausführungen über »verschiedene Objekte, die ein Ereignis bearbeiten können«, verwirrt sein sollten, so können Sie diese Bemerkungen beruhigt vergessen, denn Sie brauchen sich bei der grundlegenden Ereignisbearbeitung gar nicht bewusst zu sein, dass das Ereignis von

einem »Objekt« bearbeitet wird, es genügt für den Anfang, wenn Sie die Methode (alias Funktion, Unterprogramm) kennen, die (das) das Ereignis bearbeitet.

Für Eingeweihte der OOP sei aber schon gesagt: Verknüpfungen zu Methoden anderer Objekte als des Formulars können Sie auch in Delphi durch Anweisungen im Programmcode vornehmen (einfache Zuweisungen genügen, siehe z.B. R57). Die Tatsache, dass alle Ereignisse eines Fensters vom Formular bearbeitet werden, führt dazu, dass die Deklaration der Formularklasse, ja die ganze Quelltextdatei mit den Methodenimplementationen etwas unübersichtlich werden kann. Falls Sie hier eine stärkere Kapselung / Modularisierung wünschen, können Sie möglicherweise das Konzept der Frames nutzen und ein Formular mit verschiedenen eigenständigen Teilbereichen in Frames aufspalten (siehe Kapitel 3.7.2). Jedes Frame verfügt über eine eigene Klasse, über einen eigenen Satz von Methoden und die Ereignisse der Komponenten eines Frames können sogar mit zwei Methoden gleichzeitig verknüpft werden: mit Methoden des Frames und mit Methoden des Formulars.

1.2 Von Formularen und Komponenten

Neben den Ereignissen spielen Formulare und Komponenten die zweite Hauptrolle in einer Delphi-Anwendung. Doch werfen wir zunächst einen Blick auf die Fensterstruktur einer allgemeinen Windows-Anwendung, die sich natürlich auch bei einer Delphi-Anwendung wiederfinden lässt.

1.2.1 Die Fensterhierarchie einer Windows-Anwendung

In den meisten Programmierumgebungen sind Fenster die Hauptbausteine einer Anwendung. Haben Sie schon mit dieser Sichtweise gearbeitet, werden Sie sich vielleicht fragen, welcher Zusammenhang zwischen Formularen, Komponenten und Fenstern besteht. Für die Leser, die mit der Fensterhierarchie von Windows noch nicht vertraut sind, sei diese noch einmal kurz erläutert, denn sie ist natürlich auch bei Delphi vorhanden:

Unter Windows gelten nicht nur diejenigen Rechtecke als Fenster, die eine Titelzeile und einen mit der Maus manipulierbaren Rahmen besitzen. Auch kleinere Bauteile, die meistens als Steuerelemente in Dialogboxen vorkommen, wie verschiedenartige Schalter, Eingabefelder, Listen usw., werden unter Windows als Fenster behandelt. Alle Fenster sind in einer Eltern-Kind-Fensterhierarchie angeordnet, d.h., ein Fenster kann mehrere Kindfenster haben und jedes Fenster kann ein Elternfenster haben. Die Hauptfenster einer Anwendung haben unter Windows offiziell kein Elternfenster, obwohl der Windows-Hintergrund (der *Desktop*) auch als Fenster ansprechbar ist. Um eine spätere Verwirrung zu vermeiden, sollten Sie diese Beziehung der Fenster immer als Eltern-Kind-Beziehung ansehen, denn das *Besitzen* von Fenstern ist ein anderes

Thema, das sich in diesem Buch speziell auf einen Mechanismus von Delphis VCL bezieht und in Kapitel 3.2.1 erörtert wird.

Drei der am häufigsten vorkommenden Fensterkonstruktionen sind:

- *Hauptfenster*: Das Hauptfenster einer einfachen Anwendung hat als Unterfenster oft eine Mauspalette und eine Statuszeile. Zwischen diesen befindet sich ein Bereich, in dem das Programm Informationen darstellt, wie z.B. den Inhalt der Festplatte oder einer Datenbank. Oft kann in diesem Bereich auch ein Dokument bearbeitet werden (z.B. Text oder Grafik). Der freie Bereich zwischen Mauspalette und Statuszeile heißt auch *Arbeitsbereich* oder *Client-Bereich* (*Client-Area*).
- *Dialogboxen*: Eine Dialogbox ist das Elternfenster all seiner Steuerelemente, und auch Elemente, die nur der optischen Gestaltung dienen, wie z.B. eine dreidimensionale Hervorhebung unter einer Schaltergruppe, können als eigene Unterfenster realisiert sein.
- *MDI-Anwendungen*: Diese zeichnen sich dadurch aus, dass mehrere Dokumente (oder andere Informationen) in verschiedenen Fenstern gleichzeitig bearbeitet werden können. Statuszeile und Symbolleiste bleiben im Hauptfenster und der Client-Bereich wird zu einer Fläche, auf der die MDI-Kindfenster angeordnet werden (jedes dieser Kindfenster hat einen eigenen Client-Bereich).

Mit Delphi können Sie Ihre Anwendung auf die drei oben genannten und auf viele weitere Arten strukturieren. Ein Beispiel für eine relativ lockere Verknüpfung einzelner Fenster zu einer Anwendung (ohne diese Fenster in einem gemeinsamen Hauptfenster darzustellen) ist die Delphi-Oberfläche selber. Zwar gibt es auch hier ein Hauptfenster mit Menü und Schalterleisten, die darunter angeordneten Fenster Objektinspektor, Objekt-Hierarchie, Formular und Quelltexteditor (in Abbildung 1.1 verdeckt) sind jedoch davon getrennt und können unabhängig voneinander auf dem gesamten Bildschirm bewegt werden.

1.2.2 Formulare und Fenster

Beim Entwurf einer Anwendung mit Delphi tritt der Begriff des Fensters zunächst einmal in den Hintergrund. Anwendungsfenster und Dialogboxen werden hier gleichermaßen als *Formulare* behandelt und können im Formular-Editor entworfen werden, ebenso wie MDI-Haupt- und Kindfenster. Alle kleineren Bauteile in diesen Fenstern sind *Komponenten*.

Wie Sie am Beispiel einer MDI-Anwendung sehen können, ist es nicht sinnvoll, jedes Fenster, das bei Verwendung der Anwendung benötigt wird, einzeln zu entwerfen, denn bei MDI-Anwendungen ist die Zahl der Kindfenster nicht vorhersehbar und theo-

retisch unbegrenzt. Daher ist es wichtig, die Formulare, die Sie in Delphi entwerfen, als Formulare oder Schablonen zu verstehen, nach denen später beliebig viele »echte« Fenster produziert werden können.

Da Delphi dafür sorgt, dass beim Programmstart automatisch ein Fenster nach Muster des Hauptformulars angelegt und aktiviert wird, kann dieses Fenster leicht mit dem Formular selbst verwechselt werden. Im objektorientierten Sprachgebrauch ist das Fenster jedoch eine *Instanz* einer Fenster*klasse*, die Sie durch das Formular definiert haben.

Bei Untersuchung der VCL-Klassenhierarchie werden Sie schnell feststellen, dass die Klasse für die Formulare, *TForm*, von der Komponenten-Klasse, *TComponent*, abgeleitet ist. Ein Formular *ist* also eine Komponente. Sie taucht jedoch nicht in der Komponentenpalette auf, das heißt, Formulare können nicht aus »Unterformularen« zusammengesetzt werden. Wenn Sie aus irgendeinem Grund gerne ein bestehendes Formular wie eine Komponente beim Entwurf eines neuen Formulars verwenden wollen, bieten sich jedoch Frames als Lösung an (siehe Kapitel 3.7.2).

1.2.3 Komponenten

Die Bestandteile des Fensters, die in anderen Programmiersprachen auch als Steuerelemente, Dialogelemente, Controls etc. bezeichnet werden, tauchen in Delphi als Komponenten wieder auf. Komponenten sind jedoch ein noch viel weiter gehendes Konzept, denn Komponenten müssen nichts mit Windows-Steuerelementen zu tun haben und sie können auch unsichtbar sein.

Wie Kapitel 3 genauer erläutern wird, entsprechen die verschiedenen Komponententypen einem Teil der Klassenhierarchie der Delphi. Tatsächlich handelt es sich bei den Komponententypen um Object-Pascal-Klassen, die speziell zur Bearbeitung mit den Delphi-Tools (z.B. dem Objektinspektor) vorbereitet sind. Mehr dazu erfahren Sie in Kapitel 6.

Komponenten und Fenster

Wie erwähnt, gelten auf Programmier-Ebene auch kleinere Fensterteile wie Schalter und Listen als eigene Fenster. Unter Delphi müssen Sie sich zunächst keine Gedanken darüber machen, ob die Komponenten, mit denen Sie Schalter, Listen und andere Elemente erzeugen, auch »echte« Windows-Fenster sind. Meistens ist das zwar der Fall, manche Komponenten wie z.B. die *Label*-Komponente sind jedoch keine Windows-Fenster. Als fortgeschrittener Delphi-Programmierer finden Sie in Kapitel 3.1.3 genaue Informationen zu diesem Thema.

1.3 Der Entwicklungszyklus

(Ich darf an dieser Stelle herzlich die Leser begrüßen, die sich für die kleine Route durch Kapitel 1 entschieden haben und hier mit dem Lesen beginnen.)

1.3.1 Der Entwicklungszyklus in der Delphi-IDE

Die Entwicklung einer einfachen Delphi-Anwendung besteht meistens aus den folgenden Schritten (wobei Sie jederzeit zu einem früheren Schritt zurückkehren können, um beispielsweise neue Komponenten hinzuzufügen):

1. Auswahl und Hinzufügen der Komponenten
2. Einstellung der Komponenten-Properties
3. Schreiben der Ereignismethoden
4. Starten der Anwendung (Testen, Fehlersuche etc.)

Zwar kann auch ein leeres Fenster Ereignisse empfangen, jedoch werden Sie wahrscheinlich zuerst Kontrollelemente wie Menüs, Schalter und Eingabefelder darin unterbringen wollen. Dazu wählen Sie in Schritt 1 die gewünschten Elemente aus der Komponentenpalette und zeichnen sie in das Formular ein. Je komplexer die Anwendung ist, desto mehr ist es übrigens notwendig, die Benutzerschnittstelle schon hier sorgfältig zu planen, damit sie trotz der Komplexität übersichtlich und logisch bleibt.

Sobald Sie eine Komponente hinzugefügt haben, können Sie als zweiten Schritt im Objektinspektor zahlreiche Feineinstellungen vornehmen, indem Sie die Attribute der Komponenten verändern, z.B. die Hintergrundfarbe, den Schaltertext usw.

Die hinter der Oberfläche befindliche Funktionalität bringen Sie im dritten Schritt als Reaktion auf die passenden Ereignisse unter. Dazu wählen Sie aus der zweiten Seite des Objektinspektors die Ereignisse aus, auf die Ihre Anwendung reagieren soll. Delphi führt Sie in den Code Editor an die Stelle, an der Sie die Anweisungen für das Ereignis eingeben können.

An jedem Punkt dieses Prozesses können Sie die vorläufige Anwendung von Delphi ausführen lassen, sofern sich keine Fehler im Programmcode befinden. Wenn Sie noch keine Ereignisbearbeitung festgelegt haben, erhalten Sie ein nutzloses, aber funktionsfähiges Fenster: Sie können darin schon die Standardaktionen ausprobieren (so lässt sich in ein Editierfeld Text eintragen, der dort allerdings ein völlig sinnloses Dasein fristen muss, es sei denn, Sie kopieren ihn mit dem Standard-Tastenkürzel oder über das Kontextmenü des Editierfelds in die Zwischenablage).

Darüber hinaus können Sie nahezu beliebig zwischen den Schritten wechseln. Bei großen Anwendungen ergeben sich sicher noch während der Entwicklung neue Aufgaben

für weitere Komponenten, die Sie problemlos nachträglich hinzufügen können. Auch alle Eigenschaften und Ereignisse im Objektinspektor bleiben veränderbar. Das Einzige, was im aufgelisteten Entwicklungszyklus festgelegt ist, ist, dass Sie eine Komponente in das Formular eingefügt haben müssen, bevor Sie deren Eigenschaften und Ereignisbehandlung über den Objektinspektor ändern können.

Ein fester Bestandteil des Entwicklungszyklus bei einem größeren Programm ist natürlich auch das Testen der Anwendung und die Korrektur von Fehlern. Zum dafür vorgesehenen integrierten Debugger kommen wir in Kapitel 1.7.

Nicht weiter behandelt werden soll in diesem Buch der Entwurf von Software mit Hilfe von Methoden des Software-Engineerings, hierfür gibt es eigene umfangreiche, Programmiersprachen-unabhängige Literatur. Der oben beschriebene Entwicklungszyklus lässt sich jedenfalls problemlos mit verschiedenen Techniken des Software-Entwurfs kombinieren: Auch wenn der Funktionsumfang und das Aussehen einer Anwendung sorgfältig im Voraus geplant wurden, können sich durch die praktische Arbeit mit einem ersten Delphi-Formular-Prototyp leicht neue Ideen ergeben, die ein nachträgliches Hinzufügen von Komponenten empfehlenswert erscheinen lassen.

1.3.2 Übersicht über das Beispielprogramm

Im nächsten Kapitel beginnen wir mit einem kleinen Beispielprogramm, das bis zum Ende von Kapitel 1.8 weiterentwickelt wird. Das Beispielprogramm soll zunächst einmal als Uhr funktionieren und die Aufgabe erhalten, ständig die aktuelle Uhrzeit in Ziffern anzuzeigen. Da dies mit Delphi eine extrem einfache Aufgabe ist, haben wir Zeit, eine einfache Weckfunktion zu integrieren, dem Benutzer einen Schriftartwechsel zu ermöglichen und verschiedene andere Dinge daran zu demonstrieren.

Sie können aus den nun folgenden Abschnitten auch eine kurze Praxiseinführung machen, wenn Sie nur die Abschnitte lesen, die sich mit dem Beispielprogramm beschäftigen, und die anderen Abschnitte später bei Bedarf lesen. Die »kurze Route« durch dieses Kapitel verläuft über die folgenden Stationen, die sich mehrheitlich mit dem Beispielprogramm beschäftigen:

- Kapitel 1.4.3, *Entwerfen von Formularen*. Im Abschnitt *Zum Beispielprogramm* (ab Seite 44) bauen Sie das Formular des Beispielprogramms auf, noch ohne die Properties zu setzen.
- Kapitel 1.4.5, *Der Objektinspektor*. Im Abschnitt *Properties für das Beispielprogramm* werden die Properties des Beispielprogramms eingestellt und erläutert.
- In Kapitel 1.5.1 geht es zwar nicht um das Beispielprogramm, die hier praktisch demonstrierten Techniken der Ereignisbearbeitung sind aber Voraussetzung für die übernächste Station.

- Für Pascal-Neulinge: Die Schnellübersicht in Kapitel 1.5.2 soll Ihnen, falls Sie sich mit Object Pascal oder einer ähnlichen Sprache wie C++ noch nicht auskennen, einen Überblick über die Aktionsmöglichkeiten im Innern einer Methode geben.
- In Kapitel 1.5.4 wird das Beispielprogramm durch Hinzufügen zweier Ereignisbearbeitungsmethoden zum ersten Mal zu einer funktionsfähigen Uhr.
- Kapitel 1.5.5 erweitert diese Uhr um die Weckfunktion und einen Schriftwahldialog.
- In Kapitel 1.6 und 1.7 gibt es erst einmal eine Pause vom Beispielprogramm, denn diese Kapitel widmen sich Einrichtungen der IDE wie Editor, Projektverwaltung, Objektablage, Browser und Debugger. Pascal-Neulinge brauchen nicht alle der dort beschriebenen Dinge sofort zu verstehen, sondern können sich anhand dieser Kapitel einen Überblick darüber verschaffen, was ihnen die Delphi-Entwicklungsumgebung noch alles zu bieten hat.
- Das gesamte Kapitel 1.8 ist schließlich einer neuen Version des Beispielprogramms gewidmet, die Gebrauch von der *StringGrid*-Komponente macht und bereits intensiver auf die Programmierschnittstelle der VCL zugreift.

Empfehlenswerter, als das komplette Beispielprogramm nachzubauen, ist es, das Beispielprogramm von der CD-ROM zu laden (DATEI | PROJEKT ÖFFNEN... wählen und das Projekt *Wecker1* bzw. *Wecker2* laden), die zugehörigen Kapitel zu lesen und dann das Beispielprogramm zu verändern und eigene Experimente zu starten.

1.4 Die IDE und die visuellen Werkzeuge

Wir beschäftigen uns nun konkret mit den Bestandteilen der Delphi-Oberfläche (IDE, Integrated Development Environment) und beginnen dabei mit dem Beispielprogramm des Kapitels. Für den Rundgang durch die IDE und das Beispielprogramm benötigen wir zuerst ein neues Projekt mit einem leeren Formular. Wenn Sie ein neu installiertes Delphi starten, legt dieses automatisch ein solches an. Um später nach beliebigen anderen Arbeitsschritten zu diesem Zustand zurückzukehren, wählen Sie aus dem Hauptmenü den Punkt DATEI | NEUE ANWENDUNG. Sie erhalten dann eine Bildschirmanzeige ähnlich der in Abbildung 1.1 gezeigten.

1.4.1 Der Aufbau der IDE

Der Grundaufbau der IDE von Delphi 6 stimmt noch immer mit dem der ersten Delphi-Version überein; so befindet sich im oberen Teil des Bildschirms ein Hauptfenster, das nur aus dem Menü, ein paar Symbolleisten und der Komponentenpalette besteht. Andere Fenster, darunter die Formulare, werden unter diesem Hauptfenster angeordnet.

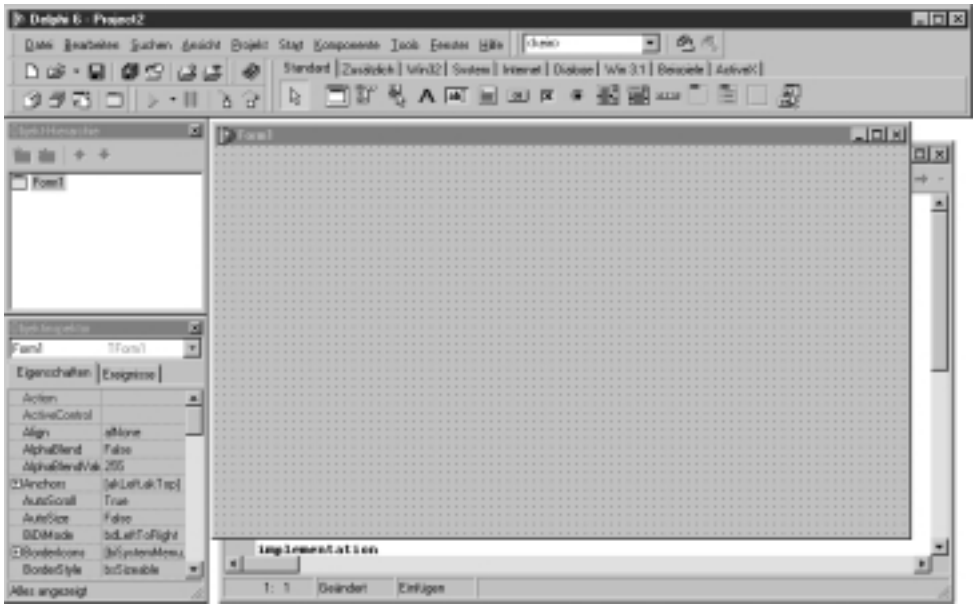


Abbildung 1.1: Ein neues Projekt (hier in der Personal-Ausgabe von Delphi)

Seit Delphi 4 besteht auch die Möglichkeit, die Symbolleisten und die Komponentenpalette aus dem Hauptfenster herauszuziehen und als separate Leisten frei auf dem Bildschirm zu platzieren. Auch können Sie die Fenster, die sich unter dem Hauptfenster befinden, flexibel aneinander docken und übereinander stapeln. Wenn Sie die weiter unten beschriebene Autospeichern-Option einschalten, speichert Delphi für jedes Projekt ein separates Fenster- und Docking-Layout, das bei jedem Öffnen des Projekts wiederhergestellt wird. Seit Delphi 5 können Sie sogar verschiedene Layouts mit Namen benennen und über die neben der Menüleiste befindliche aufklappbare Liste beliebig zwischen ihnen wechseln.

Lokale Menüs

Das Hauptmenü steht nicht nur mit der Symbolleiste in Konkurrenz, sondern auch mit den lokalen Menüs (auch als Popup- oder Kontextmenüs bezeichnet). Alle Delphi-Fenster verfügen über derartige Menüs, die Sie mit der rechten Maustaste aufrufen. Delphi listet im Popup-Menü nur Funktionen auf, die sich auf das angeklickte Fenster beziehen. Jedes Fenster verfügt über ein völlig eigenes Popup-Menü.

Falls Sie schon mehrere andere Programme mit Kontextmenüs verwendet haben, fragen Sie sich sicher, welches Konzept bei Delphi zugrunde liegt. Sie werden feststellen, dass der Inhalt der lokalen Menüs nicht streng logisch begründet ist, es sind also weder alle möglichen Aktionen für das Fenster darin enthalten, noch gibt es für alle

lokalen Menüpunkte einen gleichwertigen globalen Hauptmenüpunkt (z.B. nicht für das Konfigurieren der Symbolleiste).

Am besten ist es, die verschiedenen lokalen Menüs einfach auszuprobieren, denn sie enthalten viele bequeme Abkürzungen: Statt beispielsweise die Dialogseite TOOLS | UMGEBUNGSOPTIONEN | PALETTE über das Hauptmenü und das Seitenregister aufzurufen, können Sie aus dem lokalen Menü der Komponentenpalette den Punkt EIGENSCHAFTEN aufrufen. Schließlich lassen sich manche besonders selten benötigte Funktionen nur in lokalen Menüs finden, z.B. HINWEISE ZEIGEN im Menü der Komponentenpalette und NUR LESEN im Menü des Editors.

Fenstermanagement und Docking

Aufgrund des Verzichts auf das MDI-Konzept gibt es in der IDE kein Menü, mit dem sich Fenster automatisch anordnen lassen (etwa mit Anordnungsbefehlen wie NEBENEINANDER oder ÜBEREINANDER). Jedoch ordnet das Editorfenster standardmäßig alle Quelltextdateien in einem Seitenregister an, in dem Sie auch mit Shift+Strg+Tab blättern können, und Objektinspektor sowie Objekt-Hierarchie-Fenster bleiben normalerweise sowieso an ihrer Position. Unübersichtlich kann die Lage dann aber noch durch viele geöffnete Formulare werden. Zwischen diesen können Sie über die Liste, die Sie mit Shift+F12 erhalten (ANSICHT | FORMULARE), wechseln. ANSICHT bietet noch einige weitere Optionen zur Organisation der IDE wie etwa die Erzeugung eines neuen Editorfensters mit eigenem Seitenregister, aber diese Optionen können Sie sicher leicht selbst erforschen.

Eine sehr willkommene Neuerung in Delphi 6 ist das FENSTER-Menü, das eine Liste aller momentan geöffneten Fenster enthält, inklusive der geöffneten Formulare (im Unterschied dazu enthält die Liste, die Sie unter ANSICHT | FORMULARE erhalten, die Formulare des aktuellen Projekts). Wenn Sie einmal mit einigen der zahlreichen speziellen Editoren arbeiten, die in der IDE zu den verschiedensten Gelegenheiten eingesetzt werden (z.B. Menüeditor, Datenbank-Feldereditor, Aktionslisteneditoren für GUI-Aktionslisten und Webmodule), und diese Editoren von anderen Fenstern verdeckt werden sollten, werden Sie dieses Menü schnell zu schätzen lernen.

Was wieder eher selbst zu erforschen bzw. am besten durch Praxisexperimente zu erlernen ist, ist das Fenster-Docking. Abbildung 1.2 zeigt ein Beispiel einer durch Docking umkonfigurierten Delphi-IDE. Zum Andocken in Frage kommen alle Hilfsfenster mit einer schmalen Titelzeile wie etwa der Objektinspektor.

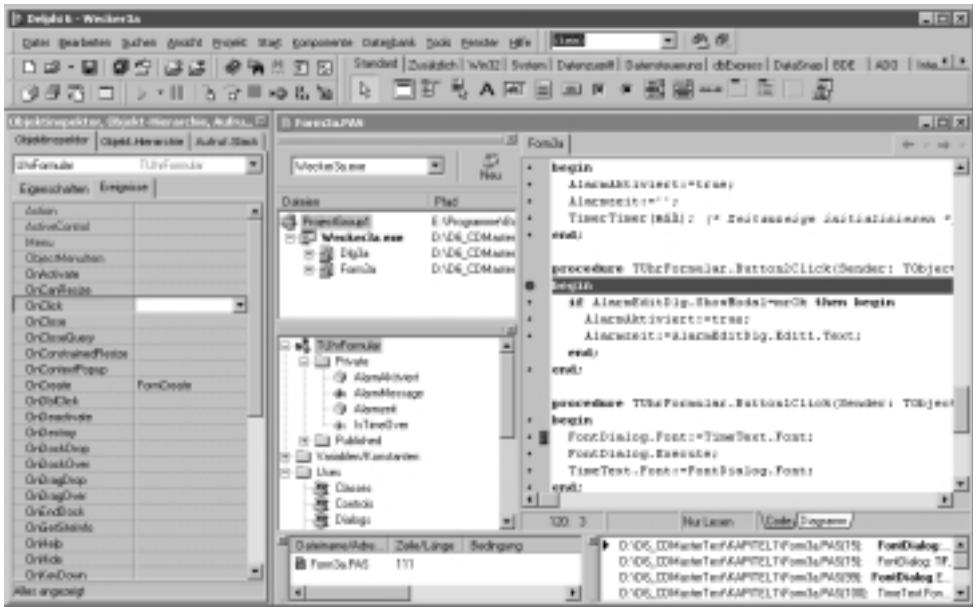


Abbildung 1.2: Nur eine von unzähligen Möglichkeiten, die Fenster der Delphi-IDE zu verbinden und den Platz rechts von den Menüpunkten mit einer frei konfigurierbaren Symbolleiste zu füllen.

Angedockt werden kann zum einen an allen Seiten des Editorfensters außer an der Oberseite, zum anderen können Hilfsfenster gegenseitig aneinander oder sogar übereinander andocken. Zum Andocken (oder zum Wiederabtrennen) eines Fensters ziehen Sie es einfach mit der Maus an die gewünschte neue Position bzw. auf die gewünschte Seite des Editorfensters.

Während dieses Ziehvorgangs erhalten Sie durch einen grau gezeichneten Vorschaurahmen Rückmeldung darüber, was mit dem Fenster passieren würde, ließen Sie die Maustaste jetzt los. Hierfür gibt es grob gesagt die folgenden Möglichkeiten:

- Das Fenster wird den Bildschirm als freies Fenster betreten. In diesem Fall nimmt der Vorschaurahmen genau die vom normalen Verschieben eines Fensters gewohnten Ausmaße an.
- Das Fenster wird an der Seite eines anderen Fensters angedockt: Der Vorschaurahmen markiert in diesem Fall die betreffende Fensterseite.
- Das Fenster wird mit einem anderen Fenster zu einem mehrseitigen Fenster gekoppelt. Für diesen Fall kennzeichnet der Vorschaurahmen den mittleren Teil des anderen Fensters.

Delphi 6 bringt hier eine sehr sinnvolle Erweiterung für das Fensterdocking mit: Wenn Sie in den Umgebungsoptionen auf der Seite PRÄFERENZEN den Schalter AUTOM. ANDOCKEN BEIM ZIEHEN abschalten, werden Fenster nur angedockt, wenn Sie beim Ziehen der Maus auch noch Strg gedrückt halten. Bei normalen Mausziehoperationen können Sie somit die Fenster unbehelligt von hin- und herspringenden Vorschaurahmen auf die herkömmliche Art verschieben. (In älteren Delphi-Versionen funktioniert es nur umgekehrt: Docking ist standardmäßig aktiviert und lässt sich durch Strg unterbinden.)

Unabhängig von der Delphi-Version können Sie das Docking außerdem über die Kontextmenüs das Docking für bestimmte Fenster deaktivieren (Menü-Option ANDOCKBAR).

Die Konfiguration der IDE speichern

Sehr empfehlenswert ist es, im Umgebungsdialog (TOOLS | UMGEBUNGSOPTIONEN) auf der Seite PRÄFERENZEN in der Gruppe OPTIONEN FÜR AUTOSPEICHERN die Option (PROJEKT-)DESKTOP einzuschalten, bei der Delphi den Aufbau der IDE (offene Fenster, Position dieser Fenster) und vor allem die Liste der im Editorfenster geöffneten Dateien speichert. Wenn Sie in einem Projekt viele Dateien, vielleicht auch einige der VCL-Quelltexte gleichzeitig offen haben, müssten Sie diese Dateien ohne diese Option mühsam neu laden, wenn Sie das Projekt erneut öffnen. Auch die Docking-Konfiguration wird beim erneuten Laden des Projekts wiederhergestellt (verändert sich aber eventuell, wenn Sie zu einem anderen Projekt wechseln).

Falls Sie Auto-Speichern einschalten, legt Delphi die Daten zu jedem Projekt jeweils in einer Datei mit der Endung `.dsk` ab. Nicht zur IDE-Konfiguration zählen beispielsweise die Compiler- und Linker-Optionen, die automatisch in `.dof`-Dateien gespeichert werden, ohne dass Sie dazu einen Schalter aktivieren müssten. Die projektübergreifenden Einstellungen der IDE speichert Delphi übrigens in der Windows-Registry unter dem Schlüssel `HKey_Current_User\software\borland\delphi\6.0`.

Desktop-Konfigurationen speichern

Delphi bietet eine bequeme Möglichkeit, verschiedene Projekt-Desktops zu speichern, die auch innerhalb desselben Projekts einfach abgerufen werden können. Hierzu gibt es eine kleine, standardmäßig neben der Menüleiste angeordnete Toolbar mit einer Liste von gespeicherten Desktops (zu Beginn leer), einem Schalter zum Speichern eines Desktops sowie einen Schalter zum Festlegen des Debug-Desktops.

Die Speichern-Funktion fordert Sie zum Angeben eines Namens für den Desktop-Aufbau auf und fertigt dann einen Schnappschuss des Desktops an, den Sie sich gewissermaßen auch als Schablone vorstellen können. Denn er lässt sich von da an in allen

Projekten abrufen und stellt dann dort die Fenster so ein wie zum Zeitpunkt der Speicherung, ändert aber ihren Inhalt nicht. Sie rufen einen gespeicherten Desktop ab, indem Sie einfach den selbst gewählten Namen aus der aufklappbaren Liste auswählen.

Die gespeicherten Desktops beeinträchtigen eine eventuell eingeschaltete Desktop-Autospeichern-Funktion nicht. Diese veranlasst Delphi weiterhin, beim Öffnen eines Projekts den letzten Desktop dieses Projekts wiederherzustellen, egal ob es sich dabei um einen separat abgespeicherten Desktop handelt oder nicht. Die Desktops aus der Desktop-Liste werden also nur dann aktiviert, wenn Sie sie manuell auswählen.

Eine Ausnahme dazu bildet der Debug-Desktop. Dies ist ein über den dritten Schalter der Desktop-Symbolleiste (oder über ANSICHT | DESKTOPS | DEBUG-DESKTOP EINSTELLEN) speziell festgelegter Desktop, der automatisch aktiviert wird, sobald Sie eine Anwendung in Delphi starten und damit den integrierten Debugger aktivieren.

Hinweis: Delphi speichert die projektübergreifenden Desktops in seinem BIN-Verzeichnis jeweils unter dem Desktop-Namen, erweitert mit der Dateiendung `.dst`. Beim Start sucht es einfach alle existierenden `.dst`-Dateien zusammen, so dass Sie die Desktop-Auswahl durch Hineinkopieren neuer `.dst`-Dateien erweitern oder Sicherungskopien durch Duplizieren der `.dst`-Dateien anfertigen können.

Die Komponentenpalette

Alle in ein Formular einfügbaren Komponenten sind in der Komponentenpalette versammelt. Aufgrund ihrer großen Zahl mussten sie auf mehrere Seiten verteilt werden. Einen Überblick über einzelne Komponenten werden Sie in Kapitel 1.9 finden, an dieser Stelle soll ein Überblick über die wichtigsten Seiten der Palette genügen (nicht alle Seiten sind in allen Delphi-Versionen verfügbar):

- Auf der Seite *Standard* befinden sich die grundlegenden Kontrollelemente, die in allen Betriebssystemen bzw. grafischen Oberflächen schon seit Jahren zum – nun: eben zum Standard gehören, z.B. Listenfenster, Markierungsfelder, Eingabefelder. Auch die Menüs sind mit einer Haupt- und einer Popup-Menükomponente auf dieser Seite vertreten, und passend dazu die Aktionslisten-Komponente *TActionList*, die kein eigenständiges Steuerelement ist und in Kapitel 4.6.4 genauer erläutert werden wird. Schließlich finden Sie ganz links noch einen Schalter für *TFrame*: Dies ist eigentlich gar keine Komponente, jedenfalls keine sofort verwendbare. Um Frames verwenden zu können, müssen Sie zuerst selbst eines definieren. Wie das geht, beschreibt Kapitel 3.7.2.
- *Zusätzlich*: Hier befinden sich weitere Elemente, die in vielen Anwendungen vorkommen; von einfachen Speedbuttons bis zu komplexen Eingabekomponenten wie z.B. zwei Tabellenkomponenten.

- Die Seite *Win32* enthält Komponenten für Steuerelemente, die im 32-Bit-Zeitalter zu den Standardsteuerelementen von Windows hinzugefügt wurden. Die meisten davon stehen schon seit Windows 95 zur Verfügung, für einige davon ist unter Windows 95 zumindest ein Update der `comctl32.dll` erforderlich (diese Komponenten gibt es erst ab Delphi 4, das Update der `comctl32.dll` wird mit Delphi mitgeliefert).
- *System* enthält mit *Timer* eine Komponente für Systemereignisse, dazu Komponenten, die mit OLE und DDE in Zusammenhang stehen, sowie die Multimediakomponente und vor allem die Zeichenfläche, die zur Ausgabe von Grafiken gedacht ist.
- *Dialoge*: Diese Seite enthält acht Komponenten, die die von Windows vordefinierten Standarddialoge »verpacken«, sowie zwei zusätzliche Dialoge, die auf das Öffnen und Speichern von Bilddateien spezialisiert sind.
- *Datenzugriff*: Hier befinden sich klassische, von Delphi eingeführte Datenbankkomponenten, mit denen Sie Verbindungen zu Datenbanken und Datenbankservern aufbauen können (seit Delphi 5 erst ab der Professional-Ausgabe).
- *dbExpress* ist Borlands neue Datenbankarchitektur, die anlässlich der Cross-Platform-Datenbankentwicklung mit Kylix eingeführt wurde. In Delphi für Windows kommen diese Komponenten als Alternativen zu den BDE-Komponenten hinzu und Kapitel 7 beschreibt, warum sich die Verwendung dieser neuen Komponenten in jedem Fall vorteilhaft auswirken kann.
- *Datensteuerung* könnte in dieser Übersetzung mit dem *Datenzugriff* verwechselt werden. Was aber wirklich gemeint ist, sind *Datensteuerelemente*, das sind überwiegend Standardkomponenten, die um eine Datenbankanbindung erweitert wurden. Mit Hilfe dieser Komponenten können Sie den Inhalt der Datenbanken auf vielfältige Weise anzeigen und editieren lassen (diese Seite ist nur in den Produktversionen verfügbar, in denen auch die Seite *Datenzugriff* vorhanden ist).
- *Win3.1* enthält einige von Delphi 1 stammende Komponenten, die in den aktuellen Windows-Versionen nicht mehr unbedingt benötigt werden, die aber unter Umständen interessante Alternativen zu den *Win32*-Komponenten bieten, in jedem Fall aber für die Kompatibilität zu 16-Bit-Delphi wichtig sind, etwa für die Portierung alter Delphi-1-Anwendungen. Wer heute eher auf Kompatibilität zu Kylix und Linux Wert legt, sollte diese Komponenten nicht mehr verwenden. Unter Kylix stehen die *Win3.1*-Komponenten nämlich *nicht* zur Verfügung.
- *Internet* enthält hauptsächlich nicht-visuelle Komponenten, die in Internet-Server-Anwendungen benötigt werden, aber auch Komponenten für Clients wie *ClientSocket* und natürlich die *WebBrowser*-Komponente, die den Microsoft Internet Explorer kapselt.

Die übrigen Seiten setzen sich je nach Delphi-Ausgabe aus Beispielkomponenten und aus Komponenten für die Bereiche Datenbanken, Internet/Netzwerk und ActiveX/COM zusammen.

Neben den schon vordefinierten Komponenten und Registerseiten befindet sich nahezu unbegrenzter Platz zum Installieren neuer Komponenten (dank der beiden kleinen Schalter zum Scrollen, falls nicht alle Registerzungen auf den Bildschirm passen). Darüber hinaus können Sie die schon installierten Komponenten beliebig zwischen den einzelnen Seiten austauschen und neue Seiten erstellen, denn auch die Komponentenpalette gibt in ihrem lokalen Menü den Punkt EIGENSCHAFTEN an, der Sie aber nicht zu einem eigenen Editor, sondern lediglich auf eine Seite führt, die es auch im Optionsdialog des Menüpunktes TOOLS | UMGEBUNGSOPTIONEN gibt.

Eine alphabetische Auflistung aller Komponenten finden Sie unter dem Menüpunkt ANSICHT | KOMPONENTENLISTE. Diese Liste kann Ihnen bei der Suche nach einer bestimmten Komponente, von der Sie nur den Namen kennen, behilflich sein.

Installation neuer Komponenten

Eine wichtigere Möglichkeit, die Komponentenpalette anzupassen, stellt die Installation neuer Komponenten dar. Diese ist Thema des Kapitels 6.2.3.

1.4.2 Hilfe zu IDE und Sprachreferenz

Abgesehen davon, dass hier der Platz fehlt, um alle Möglichkeiten der IDE zu beschreiben, wäre eine Referenz über alle Menübefehle, Schalter und Optionen sicher wenig interessant. Im restlichen Kapitel sind daher die für den grundlegenden Arbeitsablauf wichtigen und einige weitere besonders interessante Fähigkeiten der IDE herausgegriffen. Zu weiteren Details gibt Ihnen Delphi mit den üblichen Mitteln Hilfestellung (Hinweisfenster über den Schaltern der Symbolleisten, Hilfe mit F1 in den Menüs und über Hilfeschalter in Dialogboxen). Wenn Sie sich also irgendwann für die Compileroptionen interessieren, sollten Sie unter den Projektoptionen (PROJEKT | OPTIONEN) die Seite COMPILER aufschlagen und dort die Hilfsinformation abrufen. In den meisten Fällen empfiehlt es sich, als Erstes im lokalen Menü des aktuellen Fensters nachzusehen, ob dieses bereits die gewünschte Funktion enthält.

VCL-Referenz

Um bei der Programmierung schnelle Hilfe zu einem Bestandteil der VCL zu erhalten, können Sie Borlands Online-Sprachreferenz aus dem Editor heraus mit F1 aufrufen. Delphi sollte dann die Seite der Hilfedatei aufschlagen, auf der das Wort an der aktuellen Editorposition beschrieben ist. In den Hilfeseiten der einzelnen Klassen (und Komponenten, wie etwa *TButton*, *TMainMenu*) können Sie ein eigenständiges Nebenfenster

öffnen, das eine Übersicht über alle Eigenschaften, Methoden oder Ereignisse der Klasse gibt und sogar noch eine alphabetische Sortierung erlaubt. Wenn Sie dann z.B. eines der Ereignisse auswählen, wird das Übersichtsfenster nicht geschlossen, sondern bleibt neben der detaillierten Beschreibung dieses Ereignisses sichtbar und erlaubt Ihnen ein schnelles Weiterblättern zu den anderen Ereignissen oder einen Sprung zurück zur Hauptseite der Klasse.

Lücken in der Online-Referenz

Neue Versionen von Borlands Softwareentwicklungssystemen sind immer heiß begehrt und nur wenige möchten auf die neuen und eigentlich fertigen Produkte warten, bis auch die Online-Hilfe perfekt ist. Das führt dazu, dass die Online-Hilfen eines neuen Produkts fast immer einige auffällige Lücken aufweisen wie undokumentierte neue Properties oder gar ganze undokumentierte Komponentenklassen. Doch Abhilfe ist auf verschiedene Weise möglich:

- Borland stellt Updates der Hilfedateien im Internet bereit, die oft schon kurz nach Erscheinen des Produkts in den Newsgroups angekündigt werden.
- Ultimative Information über alle verfügbaren Properties und Methoden einer Klasse erhalten Sie aus dem Quelltext der VCL, in dem Sie schon durch wenige Tastendrücke die richtige Position finden können: Zunächst können Sie ohne Zuhilfenahme eines Dateiauswahldialogs und ohne das Browsen durch tiefe Verzeichnisstrukturen die gewünschte Unit öffnen, indem Sie im Editor der IDE die Eingabemarke auf den Namen der Unit setzen und Strg+↵ drücken. (Falls Sie den Namen der Unit nicht kennen: Die für eine bereits im Formular vorkommende Komponente in Frage kommenden Units befinden sich in der *uses*-Anweisung der Formular-Unit.) Im zweiten Schritt bringt Sie eine Suchfunktion des Suchen-Menüs schnell zur gewünschten Klasse. Direkt zur Klassendeklaration von *ClassName* kommen Sie z.B. bei der inkrementellen Suche spätestens nach Eingabe von `TClassName = class(`. Von dort können Sie nun bei Bedarf mit Hilfe von Strg+Shift+Pfeil↓ sogar direkt in die Implementationen von Methoden springen.

  Natürlich weisen die wenigsten Klassen im Quelltext eine Dokumentation auf, daher ist Erfahrung in der Komponentenentwicklung sehr hilfreich, um mit dem Quelltext der VCL etwas anfangen zu können. (Falls Sie noch nicht über solche Erfahrung verfügen, wäre ein erster Tipp, dass Sie die mit *private* und *protected* überschriebenen Bereiche der Klassendeklaration überspringen und sich gleich den *public*- und *published*-Bereichen zuwenden.)

Hinweis: Units können nur dann mit der oben erwähnten Tastenkombination Strg+↵ geöffnet werden, wenn sie in einem Pfad enthalten sind, der in den Umgebungsoptionen unter BIBLIOTHEK / SUCHPFAD aufgeführt ist.

1.4.3 Entwerfen von Formularen

Beim visuellen Entwurf einer Delphi-Anwendung liegt der Mittelpunkt des Interesses auf dem Formular, das in Abbildung 1.1 unter der Überschrift *Form1* bereits einen großen Teil des Bildschirms einnimmt.

Entwurfszeit und Laufzeit

Bei der Arbeit mit einem Formular wird zwischen zwei sehr verschiedenen Arbeitsbedingungen unterschieden:

- Zur *Entwurfszeit* entwerfen Sie das Formular mit den visuellen Werkzeugen der Delphi-IDE, platzieren Komponenten darin und stellen Eigenschaften im Objektinspektor ein.
- Wenn Sie eine Anwendung über Delphis START-Menü starten, erscheint die *Laufzeit*-Version des Fomulars. Zur Laufzeit haben Sie von Delphi aus keine Möglichkeit mehr, das Formular zu beeinflussen. Der Programmcode, mit dem Sie auf die verschiedenen Ereignisse des Formulars und seiner Komponenten reagieren, ist nur zur Laufzeit aktiv.

Sie können das Formular zur Entwurfszeit unter verschiedenen Blickwinkeln betrachten:

- *Als passives Objekt*, das Sie mit der Maus und mit den Werkzeugen gestalten, die um das Formular herum angeordnet sind: Verändern Sie im Objektinspektor Attribute des Formulars und seiner Bestandteile, so passen diese meistens ihr Aussehen entsprechend an (z.B. die Farbe, die Schriftart etc.). Wählen Sie aus der Komponentenpalette eine Komponente aus, können Sie diese zum Formular hinzufügen. Die *Ausrichtungspalette* (Menü ANSICHT) gibt Ihnen weiteren Einfluss auf das Formular.
- *Als teilweise aktiver Prototyp* des Fensters, das zur Laufzeit des Programms erscheinen wird. Tatsächlich laufen schon beim Entwurf des Formulars große Teile des Codes der Komponenten ab, hauptsächlich diejenigen Teile, die die Komponente auf den Bildschirm zeichnen. So können beispielsweise *DBGrid*-Komponenten den Inhalt einer Datenbank schon zur Entwurfszeit darstellen.

Die zweite Sichtweise wird besonders dann wichtig, wenn Sie eigene Komponenten entwerfen, daher erfahren Sie in Kapitel 6 Näheres dazu, *wie* aktiv der Programmcode schon zur Entwurfszeit des Formulars ist.

Hinweis: Die obigen Erläuterungen zeigen bereits, dass es in der Delphi-IDE keine klar erkennbare Einrichtung gibt, die sich als Formular-Editor oder *Formular-Designer* bezeichnen lässt. Vielmehr arbeiten verschiedene Einrichtungen der IDE (Objektinspektor, Komponentenpalette, Ausrichtungspalette, BEARBEITEN-Menü) mit der Entwurfszeit-Version des Formulars zusammen und bilden mit dieser quasi einen Formular-Designer. Delphi selbst hat intern eine klare Definition des Formular-Designers und stellt diese den Komponentenentwicklern teilweise über Schnittstellen des OpenTools-API zur Verfügung. Es ist jedoch im Folgenden nicht erforderlich, von einem eigenständigen Formular-Designer zu sprechen.

Hinzufügen von Komponenten

Delphi befindet sich nach dem Start in einem »Manipulationsmodus« (erkennbar an der Hervorhebung des Zeigersymbols links neben den Komponenten), in dem Sie mit der Maus bestehende Komponenten verrücken oder in der Größe ändern können. Um eine Komponente hinzuzufügen, müssen Sie den Typ der gewünschten Komponente zuerst in der Komponentenpalette auswählen. Delphi behandelt daraufhin die einzuzeichnende Komponente ähnlich wie ein Grafikprogramm die Elemente einer Grafik. Sie können die Komponenten also mit einem Mausklick positionieren und bei gedrückter Maustaste auf die gewünschte Größe bringen. Es genügt jedoch auch ein einziger Klick, um die Komponente in ihrer vordefinierten Größe einzuzeichnen.

Nachdem Sie ein Exemplar des gewählten Komponententyps eingefügt haben, aktiviert Delphi automatisch wieder das Zeigersymbol (also den »Manipulationsmodus«). Wenn Sie mehrere Komponenten desselben Typs hintereinander frei einzeichnen wollen, müssen Sie den Komponentenschalter besonders fest eindrücken, jedoch nicht mit zusätzlichem Druck auf die Maustaste, sondern durch gleichzeitiges Drücken von [Shift]. Der Komponentenschalter bleibt dann so lange aktiv, bis Sie ihn durch einen erneuten Klick auf irgendeinen Schalter wieder lösen.

Es gibt noch einen zweiten schnellen Weg. Mit einem Doppelklick auf die Komponente in der Palette erreichen Sie, dass Delphi ein Exemplar davon in die Mitte des Formulars setzt bzw. auf die gerade markierte Komponente (sofern diese in der Lage ist, untergeordnete Komponenten aufzunehmen). Diesen Doppelklick können Sie so oft wiederholen, wie Sie Komponenten benötigen, müssen diese aber danach per Maus in die richtige Lage und Position bringen.

Editieroperationen

Die folgende kurze Zusammenfassung soll Ihnen einen Überblick über die Möglichkeiten des Formularentwurfs geben, zunächst über die Manipulation der Komponenten mit Maus und Tastatur (die Tastatursteuerung ist nur möglich, wenn kein Element der Komponentenpalette gewählt ist):

- Mit einzelnen Mausklicks markieren Sie einzelne Komponenten des Formulars; halten Sie zusätzlich [Shift] gedrückt, werden dabei die schon bestehenden Markierungen nicht gelöscht, so dass Sie mehrere Komponenten gleichzeitig auswählen können. In Delphi 6 können Sie mit Hilfe des Objekt-Hierarchie-Fensters jede beliebige Komponente des Formulars markieren, selbst wenn sie vollständig verdeckt sein sollte (dies kommt häufig vor, wenn die zu markierende Komponente (z.B. ein *Panel*) als Basisfläche für eine andere Komponente dient, von dieser vollständig ausgefüllt wird und nicht über einen eigenen Rand verfügt, wenn also etwa *Panel.BorderWidth* gleich Null ist).

- Um von Komponente zu Komponente zu springen, verwenden Sie die Pfeiltasten oder die Taste [Tab] (mit »Springen« ist gemeint, dass jeweils eine andere Komponente markiert wird).

- Indem Sie mit der Maus ein Rechteck um mehrere Komponenten ziehen, wählen Sie alle Komponenten aus, die innerhalb des Rechtecks liegen. Wenn dies nicht möglich ist, weil sich die Komponenten innerhalb einer anderen Komponente befinden, die Sie durch die Maus verschieben würden, halten Sie [Strg] gedrückt, wenn Sie beginnen, das Rechteck aufzuziehen. Oder halten Sie [Shift] gedrückt und klicken Sie die auszuwählenden Komponenten einzeln an.

- Mehrere ausgewählte Komponenten können Sie mit der Maus gleichzeitig verschieben. Eine gemeinsame Größenänderung ist nur über den Menüpunkt GRÖSSE... (lokales Menü des Formulars und BEARBEITEN-Menü) und beim Skalieren möglich (Menüpunkt SKALIERUNG...). Beim Skalieren werden jedoch Höhe und Breite der gewählten Komponenten um denselben Faktor vergrößert.

- Beim Verschieben ändert sich lediglich die Position einer Komponente, es ist auf diese Weise nicht möglich, eine Komponente als »Kindkomponente« einer anderen Komponente unterzuordnen (etwa einem *Panel*). Hierzu müssen Sie die Komponente im Objekt-Hierarchie-Fenster auf die gewünschte Elternkomponente ziehen (oder die Komponente per Ausschneiden und Einfügen in die neue Elternkomponente einfügen, was bis Delphi 5 die einzige Möglichkeit eines Elternwechsels darstellt).

- Wenn sich Komponenten überlappen, können Komponenten ganz oder teilweise von anderen Komponenten verdeckt werden. Um verdeckte Komponenten wieder sichtbar zu machen, verwenden Sie einen der lokalen Menübefehle NACH HINTEN SETZEN und NACH VORNE SETZEN. Ersterer platziert die markierten Komponenten hinter alle nicht markierten, Letzterer verhält sich genau umgekehrt. Diese Verschiebung ist jedoch nicht immer wirksam. So können Sie *SpeedButtons* oder *Labels* beispielsweise nicht auf einen *Button* legen. Allgemein können Komponenten, die

keine Fenster sind, nicht auf Komponenten liegen, die Fenster sind. Da Sie sich fürs Erste nicht weiter mit diesem Unterschied zu beschäftigen brauchen, kommt erst Kapitel 3.1.3 wieder darauf zurück.

- Es gibt auch Tastaturkürzel zum Verschieben, Vergrößern und Verkleinern der aktuellen Komponente(n), diese bieten sich aufgrund ihrer Genauigkeit für Feintuning an: Mit Strg und den Pfeiltasten verschieben Sie die gesamte Komponente, mit Shift und den Pfeiltasten verschieben Sie nur die rechte untere Ecke. Mit den Pfeiltasten alleine bewegen Sie sich von Komponente zu Komponente.
- Die üblichen Ausschneiden/Einfügen-Operationen (Cut & Paste) stehen natürlich auch für Komponenten zur Verfügung. Ein Anwendungsbeispiel ist das schnelle Vervielfältigen einer gerade eingezeichneten Komponente: Diese ist nach dem Einzeichnen automatisch markiert, so dass sie mit BEARBEITEN | KOPIEREN schnell in die Zwischenablage kopiert werden kann. Mit dem EINFÜGEN-Befehl können Sie von dieser Vorlage nun beliebig viele Kopien anfertigen, die Delphi etwas verschoben über dem Original anordnet. Diese Methode ist natürlich nur dann schnell, wenn Sie statt der Menübefehle die Tastenkürzel verwenden. Diese hängen jedoch von Ihrer gewählten Tastatureinstellung ab und werden neben dem entsprechenden Menüpunkt angezeigt.
- Zum Ausrichten der ausgewählten Komponente(n) besitzt Delphi eine eigene Schalterleiste, die Sie mit ANSICHT | AUSRICHTUNGSPALETTE aktivieren. Die einzelnen Schalter erklären sich über automatisch erscheinende Hinweise selbst, sobald Sie den Mauszeiger kurze Zeit über das fragliche Symbol halten. In der Dialogbox AUSRICHTEN... (lokales Menü des Editors) finden Sie dieselben Optionen in schriftlicher Form wieder.
- Um die markierten Komponenten zu löschen, drücken Sie Entf. Sie löschen damit auch die Einstellungen, die Sie im Objektinspektor vorgenommen haben, können die Löschung aber mit BEARBEITEN | RÜCKGÄNGIG wieder ungeschehen machen.

Reihenfolgen der Komponenten

Die Komponenten eines Formulars liegen nicht einfach so im Formular herum, sondern sind in zweierlei Reihenfolgen angeordnet: Alle sichtbaren Elemente, die von der Tastatur angesteuert werden können, können Sie innerhalb der TABULATORREIHENFOLGE (lokales und globales Menü BEARBEITEN) umgruppieren – das ist die Reihenfolge, in der die Elemente durch Drücken von Tab angesteuert werden (wie in jedem üblichen Windows-Dialog). Eine andere Möglichkeit, die Tabulatorreihenfolge zu ändern, liegt in der Eigenschaft *TabOrder*, die Sie bei jeder ansteuerbaren Komponente im Objektinspektor finden (dazu später mehr).

In der zweiten Reihenfolge (ERSTELLUNGSREIHENFOLGE) brauchen Sie für den Einstieg noch keinen Sinn zu sehen: Sie bezieht sich nur auf die nicht-visuellen Komponenten, beispielsweise die Datenzugriffskomponenten. Sie können die Reihenfolge festlegen, in denen diese initialisiert und im Array *Components* angeordnet werden (siehe Kapitel 3.2.1).

Sichtbare Elemente, die nicht von der Tastatur angesteuert werden, können Sie in keiner der Reihenfolgen anordnen, da ihre Reihenfolge programm- und bedientechnisch keine Rolle spielt (zu diesen Komponenten gehören wieder die Komponenten, die sich schon weder nach vorne noch nach hinten schieben ließen, wie z.B. die *Label*-Komponente).

Eine weitere Möglichkeit, die Sie zum Entwurfszeitpunkt der Anwendung haben, ist, Komponenten zu gruppieren. Gruppierte Elemente sind einem gemeinsamen Gruppenelement, meistens einer Komponente des Typs *TPanel*, untergeordnet (auch als *Elternkomponente* bezeichnet), gehören aber weiterhin in den Besitz des Formulars. Mehr dazu finden Sie in *R*1 auf Seite 376.

Zum Beispielprogramm

Das Hauptfenster für unsere Uhr benötigt in der ersten Version die folgenden Komponenten von der Seite *Standard* (Delphi zeigt die Namen der Komponenten als Hinweis an, wenn Sie die Maus über die Mauspaletten-Schalter bewegen):

- ein Textfeld (*Label*) für die Anzeige der Uhrzeit
- einen einfachen Schalter (*Button*), über den Sie eine Dialogbox zum Ändern der Schriftart aufrufen können
- ein Hintergrundelement (*Panel*), das als dreidimensionale Plattform für das Eingabefeld und dessen Beschriftung dienen soll
- ein Eingabefeld (*Edit*), in dem Sie die Weckzeit eintragen können
- ein weiteres Textfeld (*Label*), das als Beschriftung für dieses Eingabefeld dient
- eine *FontDialog*-Komponente, die dem Programm Zugriff auf den Dialog zur Schriftauswahl ermöglicht

Später wird das Formular etwas spektakulärer, wenn es auch noch von einer Tabelle und einem Markierungsschalter bevölkert wird und sich außerdem auf- und zuklappen lässt.

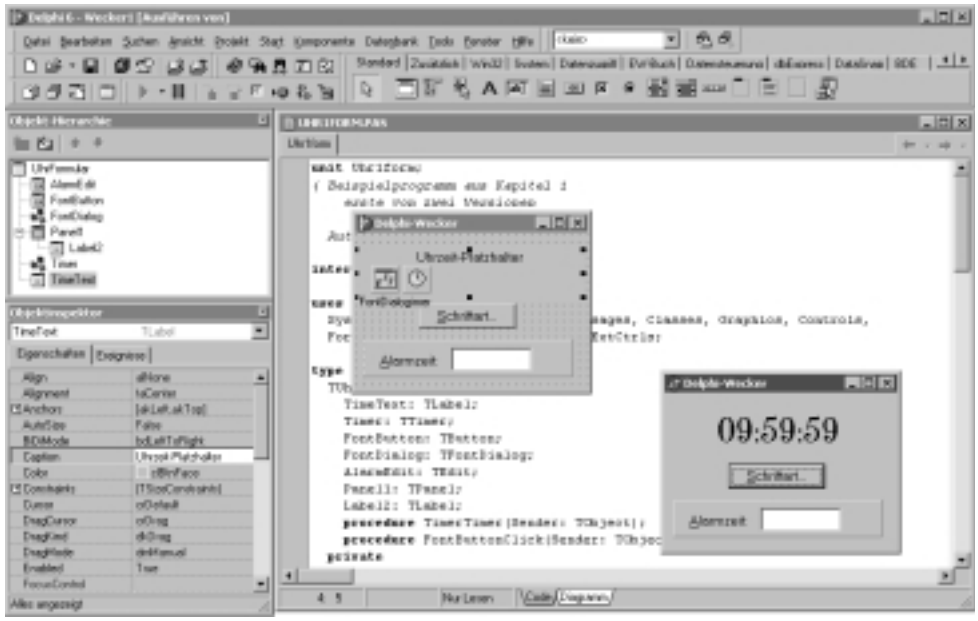


Abbildung 1.3: Das erste Formular des Beispielprogramms zur Entwurfszeit, rechts unten zur Laufzeit (nach Hinzufügen der Ereignisbearbeitungsmethoden in Kapitel 1.5.4)

Zeichnen Sie, wenn Sie die einzelnen Schritte in der Praxis mitverfolgen möchten, die genannten Elemente nach Belieben in das Formular ein, das Layout spielt für die Funktionsfähigkeit des Programms (noch) keine Rolle.

Damit die Beschriftung nicht unter dem Panel verschwindet, müssen Sie das Panel *vor* dem Label in das Formular einfügen und das Label auf dem Panel einzeichnen. So wird die Beschriftung zu einem Kindelement des Panels.

So weit an dieser Stelle zum Beispielprogramm; aufwändigere Aktionen sind für den nächsten Schritt zu erwarten, wenn in Kapitel 1.4.5 die Properties eingestellt werden. Vorher benötigen wir eine Möglichkeit, das gerade begonnene Projekt erstmals zu speichern.

1.4.4 Dateiverwaltung

Wenn Sie ein neues Projekt erzeugen (DATEI | NEUE ANWENDUNG), erhalten alle Dateien dieses Projekts zunächst einen vorgegebenen Namen, den Sie später auf Anfrage von Delphi ändern können. Die vorgegebenen Namen lauten *Project1* für das Projekt, *Form1* für das Formular und *Unit1* für die zugehörige Quelltextdatei, wobei Delphi die Ziffer 1 der Formulare und Units erhöht, wenn Sie mehrere neue davon unbenannt in ein Projekt einfügen.

Es folgt eine Zusammenfassung der wichtigsten Punkte des Menüs DATEI:

- Mit ALLES SPEICHERN speichern Sie sowohl die Projektdatei als auch alle Dateien, die zum Projekt gehören und seit der letzten Änderung noch nicht gesichert worden sind.
- Mit SPEICHERN und SPEICHERN UNTER... speichern Sie nur die aktuelle Datei bzw. das aktuelle Formular (Formular und zugehörige Unit werden immer gleichzeitig gespeichert, sofern sie verändert wurden).
- Mit PROJEKT SPEICHERN UNTER... geben Sie der obersten Datei des aktuellen Projekts (der Projektdatei) einen neuen Namen. Beim ersten Speichern eines Projekts, wenn das Projekt also noch keinen Namen hat, gelangen Sie automatisch immer in diese Dialogbox. Beim ersten Speichern erhalten Sie außerdem noch eine zweite *Speichern-unter*-Dialogbox, um die Formulardatei samt zugehöriger Quelltext-Unit zu benennen. Da Delphi bei jedem neuen Projekt mit den Namen *Project1* und *Unit1* neu zu zählen beginnt, kann es schnell zu Dateinamenskonflikten kommen, wenn Sie zwei Projekte gleichzeitig im selben Verzeichnis speichern wollen, ohne die vorgegebenen Namen zu ändern (abgesehen von den anderen offensichtlichen Nachteilen quasi namenloser Dateien).
- Das Laden von Projekten kann auf verschiedene Weise geschehen: Wenn Sie die Autospeichern-Option für den Desktop verwenden, lädt Delphi beim Start immer automatisch das zuletzt bearbeitete Projekt. Über den Menüpunkt ÖFFNEN... können Sie ein anderes Projekt laden, wenn Sie in der Dateiauswahlbox eine Projektdatei (erkennbar an der Endung `.dpr` und dem von `.pas`-Dateien unterschiedlichen Icon) auswählen. Außerdem finden Sie unter dem Menüpunkt NEU ÖFFNEN und beim entsprechenden DropDown-Schalter der Symbolleiste eine Liste der zuletzt bearbeiteten Projekte.

Hinweis: Obwohl Projektdateien und Units sich bereits durch die Endung unterscheiden, müssen Sie ihnen unterschiedliche Namen geben. Sie können also den Namen der Datei, die das Hauptformular enthält, nicht für den Namen des Projekts wiederverwenden.

Weitere Informationen über Dateien und Projekte finden Sie ab Kapitel 1.6.2.

1.4.5 Der Objektinspektor

Jede Komponente und jedes Formular verfügt über eine Vielzahl an veränderbaren Eigenschaften, den *Properties*, von denen Sie die meisten im Objektinspektor verändern können. Die Werte der Properties aller Komponenten des Formulars und des Formulars selbst gehören zur »Fensterschablone« (dem Formular), von der zur Laufzeit echte

Fenster gebildet werden (siehe Kapitel 1.2.2). Das heißt, dass jedes Fenster, das Sie auf Grundlage dieses Formulars erstellen, nicht nur die Komponenten enthält, die Sie in das Formular eingezeichnet haben, sondern dass auch die Properties der Komponenten und des Formulars selbst automatisch mit diesen Einstellungen initialisiert werden.

Es gibt einige Properties (bei manchen Komponenten sogar sehr viele), die Sie nicht zur Entwurfszeit, sondern nur zur Laufzeit des Programms ansprechen können, so z.B. die Zellinhalte einer String-Tabelle (*StringGrid*, Property *Cells*) oder die Zeichenfläche *Canvas* eines Formulars oder einer Zeichenfläche (*PaintBox*).

Die Daten der Komponenten verteilen sich auf die beiden Seiten *Eigenschaften* und *Ereignisse*. Sie bestehen jeweils aus einer zweispaltigen Liste, die Sie mit einer Bildlaufleiste bewegen können. Die erste Spalte gibt den Namen des Properties bzw. des Ereignisses an, die zweite dessen Wert. Wir beschäftigen uns in diesem Kapitel nur mit der ersten der beiden Seiten.

Hinweis: Dieses Buch bezeichnet Properties fast immer deutlich als Properties, während sie in der Online-Hilfe meistens als »Eigenschaft« bezeichnet werden, was den sehr engen Bezug zum klar definierten Object-Pascal-Schlüsselwort *property* (das ja auch nicht übersetzt wurde) verschleiert. Die Bezeichnung *Eigenschaft* ist in diesem Buch allgemeineren Zwecken vorbehalten, so dass keine Verwechslungsgefahr besteht.

Property-Kategorien

Wenn Ihnen die lange Liste von Properties einer Komponente zu unübersichtlich ist, können Sie sie von Delphi in Kategorien einteilen lassen. Wählen Sie dazu einfach ANORDNEN | NACH KATEGORIE aus dem Popup-Menü des Objektinspektors. Abbildung 1.4 zeigt die dadurch hervorgerufene Darstellung des Objektinspektors. Wenn Sie sich zum ersten Mal Kategorien anzeigen lassen, sehen Sie zunächst nur die Liste der Kategorien. Über das Plus-Symbol erhalten Sie dann die Liste der Properties, die zur jeweiligen Kategorie gehören, wobei viele Properties zu mehreren Kategorien gleichzeitig gehören. Delphi merkt sich, auch wenn Sie zwischendurch wieder zur alphabetisch sortierten Normaldarstellung zurückwechseln, welche Kategorien Sie angezeigt haben wollen.

Einen Vorteil hat die Kategorisierung der Properties auch, wenn Sie die normale alphabetische Sortierung im Objektinspektor gewählt haben: Sie können Properties von uninteressanten Kategorien ausblenden und erhalten so eine übersichtlichere alphabetische Liste. Für diese Filterfunktion ist das Popup-Untermenü ANSICHT des Objektinspektors zuständig.

Auswahl des Objekts für den Objektinspektor

Der Objektinspektor zeigt von sich aus immer die Einstellungen der im aktuellen Formular ausgewählten Komponente an. Ist keine Komponente ausgewählt, stellt er die Daten des Formulars selbst dar. Außer durch die Markierung am Bildschirm können Sie das ausgewählte Element aus der aufklappbaren Liste unter der Titelzeile des Objektinspektors ersehen. Aus dieser Liste können Sie jede andere in Ihrem Projekt befindliche Komponente auswählen, um sie zu bearbeiten. Auch im neuen Objekt-Hierarchie-Fenster ist die gerade markierte Komponente immer hervorgehoben bzw. auch über dieses Fenster können Sie wählen, welche Komponente im Objektinspektor dargestellt werden soll.

Um das Formular selbst auszuwählen, klicken Sie auf einen freien Bereich, nicht auf dessen Titelzeile. Wenn das gesamte Formular mit Komponenten belegt ist, können Sie es immer noch auswählen, indem Sie die Taste Shift gedrückt halten, während Sie auf eine beliebige Komponente klicken (alternativ dazu können Sie das Formular über die Objekt-Hierarchie bzw. die aufklappbare Liste des Objektinspektors wählen).

Properties für das Beispielprogramm

Die meisten Einstellungen für das Formular unseres Beispielprogramms beziehen sich auf die Beschriftung der einzelnen Komponenten, die jeweils im Property *Caption* gespeichert wird, und vor allem auf den Namen (Property *Name*). Da Delphi als Name standardmäßig den Namen des Komponententyps mit einer Nummerierung verknüpft, entstehen dadurch wenig aussagekräftige Namen wie *Button1*, *Button2*, *Button3* usw. Das Beispielprogramm belässt es nur dann bei dieser Vorgabe, wenn es sich um eine Komponente handelt, die im Programmcode nicht direkt angesprochen wird.

Die folgende Tabelle enthält alle Einstellungen, die Sie in den Eigenschaften der Komponenten und des Formulars selbst vornehmen können, um die Beispielanwendung nachzubauen. Es handelt sich ausschließlich um einfache Einstellungen, bei denen es genügt, den Wert in das Editierfeld einzutragen. Mehr zu den komplexeren Properties erfahren Sie im Abschnitt *Typen von Properties* weiter unten.

Wählen Sie also nacheinander das Formular selbst und die einzelnen Komponenten aus und geben Sie ihren Properties die folgenden Werte:

Komponente	Property	Wert
Label für die Uhrzeit	Name	TimeText
	Alignment	taCenter
	AutoSize	False
	Caption	Uhrzeit kommt gleich...

Komponente	Property	Wert
Edit	Name	AlarmEdit
zweites Label	Name	(unverändert)
	Caption	&Alarmzeit:
	FocusControl	AlarmEdit
Button	Name	FontButton
	Caption	&Schriftart:
Timer	Name	Timer
FontDialog	Name	FontDialog
Formular	Name	UhrFormular
	ActiveControl	AlarmEdit
	Caption	Delphi-Wecker

Alle Properties dieses Beispiels sind nicht nur in den hier verwendeten Komponenten, sondern auch in einigen anderen zu finden, doch die in der Tabelle folgende Beschreibung hält sich ausschließlich an das Beispielprogramm.

Property	Bedeutung
Alignment	positioniert den von einer Komponente (hier *TLabel*) angezeigten Text und sorgt im Beispiel mit dem Wert *taCenter* dafür, dass die Uhrzeit zentriert wird, da die *Label*-Komponente die gesamte Fensterbreite abdeckt. Mit den beiden anderen möglichen Werten für *Alignment* richten Sie Text linksbündig oder rechtsbündig aus.
ActiveControl	gibt die Komponente des Formulars an, die bei Programmstart über den Tastaturfokus verfügt, im Beispiel gehen die Eingaben also sofort an das Eingabefeld für die Alarmzeit, und nicht an den Schriftauswahl-Schalter.
AutoSize	besagt, ob das *Label*-Element seine Größe automatisch dem Platzbedarf der Beschriftung anpasst. Dies würde jedoch im Beispiel die Zentrierung über die *Alignment*-Einstellung unterlaufen, da das *Label* nach der automatischen Größenanpassung nicht mehr die gesamte Fensterbreite umfassen würde. Die automatische Größenanpassung muss hier daher ausgeschaltet, also auf *False* geändert werden.
Caption	enthält den Text, der die Komponente beschriftet (im Unterschied zum Property *Text*, das Sie statt dessen in Editierfeldern antreffen. Dieser grundsätzliche Unterschied zwischen *Caption* und *Text* findet sich bei allen Komponenten wieder: *Text* steht für editierbare Werte, während *Caption*-Text vom Benutzer zur Laufzeit nicht geändert werden kann. Indem Sie einem Zeichen ein '&' voranstellen, erreichen Sie, dass dieses Zeichen unterstrichen dargestellt wird.
FocusControl	siehe Beschreibung unter »Rücksichtnahme auf die Tastaturbedienung«
Items	Liste der Punkte, die in der aufklappbaren Liste erscheinen sollen.

Bemerkung: Die oben gedruckte Tabelle der Properties enthält nur die Werte, die nicht mit der Standardeinstellung übereinstimmen. Es stimmt jedoch nicht, dass der Autor des Beispielprogramms mühsam darüber Buch führen musste, welche Properties er geändert hat, denn Delphi speichert selbst nur die geänderten Properties in den Formulardateien ab. Zur Bearbeitung der Formulardateien in der aufschlussreichen Textdarstellung siehe Kapitel 3.5.6.

Rücksichtnahme auf die Tastaturbedienung

Auch wenn man beim visuellen Design eines Delphi-Formulars die Tastatur leicht vergessen kann, sollte man die Dialoge so gestalten, dass sie auch mit der Tastatur angenehm zu bedienen sind. Dabei kommt es besonders darauf an, dass der Benutzer jedes Dialogelement mit einer einzigen Taste ansteuern kann. Auf grafischen Oberflächen können Sie die Dialogelemente oft direkt anspringen, indem Sie Alt in Kombination mit dem Buchstaben drücken, der in der Beschriftung des Elements unterstrichen ist, wie das auch bei den Menüpunkten üblich ist. Wenn keine Zeichen unterstrichen sind, bleibt Ihnen nur die Möglichkeit, mit Tab oder mit den Pfeiltasten durch das Fenster zu navigieren.

Um nun ein einfach mit der Tastatur zu bedienendes Programm zu entwickeln, genügt es bei Schaltern bereits, dass Sie in der Beschriftung (*Caption*) vor das gewünschte Zeichen ein '&' setzen, um die Ansteuerungslogik automatisch zu aktivieren. Bei *Label*-Elementen verhält es sich anders, denn wenn Sie das in diesen hervorgehobene Zeichen eingeben, soll nicht die Beschriftung, sondern das beschriftete Element angesprungen werden. Da ein *Label* nicht wissen kann, zu welchem benachbarten Element es gehört, müssen Sie dieses im *Label*-Property *FocusControl* angeben.

Da die Komponente *AlarmEdit* des Beispielprogramms über keine integrierte Beschriftung verfügt, benötigen wir eine *Label*-Komponente, deren erster Buchstabe unterstrichen ist und deren *FocusControl*-Property auf *AlarmEdit* weist. Nachdem Sie die Komponente *AlarmEdit* in das Formular eingefügt haben, können Sie diese im *FocusControl*-Property der *Label*-Komponente einfach aus der aufklappbaren Liste auswählen.

Weitere Maßnahmen, die die Tastaturbedienung erleichtern, sind:

- Die Tabulatorreihenfolge der Dialogelemente sollte nachvollziehbar sein. Wenn Sie ein Dialogfenster umorganisieren, kann diese Reihenfolge ganz schön durcheinanderkommen, so dass Sie mit Tab kreuz und quer durch das Dialogfenster springen. Zum Umstellen der Reihenfolge verwenden Sie das Property *TabOrder* oder den Punkt TABULATORREIHENFOLGE... im lokalen oder im Bearbeiten-Menü.
- Dialoge sollten zwischen kleinen Sprüngen mit den Pfeiltasten und großen Tabulatorsprüngen unterscheiden. Hierzu verwenden Sie das Property *TabStop*. Hat dieses

den Wert *True*, erreichen Sie diese Komponente mit Tab, ansonsten überspringen Sie es mit dieser Taste und müssen statt dessen die Pfeiltasten verwenden.

- Nach dem Öffnen des Dialogs sollte das erste Element gewählt sein. Dies können Sie im Formular-Property *ActiveControl* sicherstellen.

Wegweiser

Der nächste Schritt beim Aufbau des Beispielprogramms ist die Bearbeitung der Ereignisse. Damit geht es in Kapitel 1.5 weiter, die Ereignisse des Beispielprogramms werden erst in Abschnitt 1.5.4 auf Seite 72 hinzugefügt. Die folgenden Abschnitte des Kapitels 1.4 befassen sich noch weiter mit der Programmierumgebung: mit dem Editieren von Properties, Menüs und Bildern.

Properties von mehreren Komponenten gleichzeitig verändern

Eine weitere Besonderheit des Objektinspektors sollte nicht unerwähnt bleiben: Sie können mehrere Komponenten auswählen und deren Properties gleichzeitig ändern. Der Objektinspektor zeigt dann natürlich nur noch die Properties an, die allen gewählten Komponenten gemein sind. So können Sie beispielsweise die Funktionen zum linksbündigen Ausrichten von Komponenten (siehe die Ausrichtungspalette im Menü ANSICHT) ersetzen, indem Sie alle Elemente wählen, die an einer Linie linksbündig ausgerichtet werden sollen, und dann das Property *Left* gemeinsam ändern.

Oft gibt es jedoch effektivere Möglichkeiten, wie in diesem Fall die Ausrichten-Funktion oder in einem weiteren Beispiel: Sollen alle Komponenten mit einer anderen Schriftart versehen werden, ist es ratsam, die Schriftart des Formulars zu wechseln und darauf zu achten, dass das Property *ParentFont* bei den Komponenten auf *True* gesetzt ist, damit sie die Schriftart des Formulars übernehmen.

Typen von Properties

Die Art und Struktur der Properties kann sehr verschieden sein. Im einfachsten Fall handelt es sich um einfache Zahlen (z.B. bei den Properties *Left/Top*, die die Position der Komponente angeben) oder Buchstabenfolgen (Strings, z.B. *Caption* für den Fenstertitel und *Name* für den Komponentenbezeichner). In manchen Fällen müssen Sie aus mehreren vorgegebenen Werten auswählen oder können ganze Listen eingeben. Um diese verschiedenen Typen bequem editierbar zu machen, wird jeder Typ auf eine spezielle Art editiert. In der Terminologie der Borland RAD-Tools wie Delphi bedeutet das: Es gibt für jeden Property-Typ einen *Property-Editor*.

Für Zahlen und Strings besteht dieser Editor aus dem Eingabefeld, in dem Sie den Wert einfach eingeben. Falls ein komplizierterer Property-Editor für ein Feld existiert, zeigt Ihnen ein Schalter neben dem Wert des Properties dies an. Mit dem Schalter rufen

Sie den Property-Editor auf. Der Schalter ist allerdings nur sichtbar, wenn das Property gerade ausgewählt ist. Die meisten anderen Property-Typen entsprechen einem der Datentypen von Object Pascal, die in Kapitel 2.4 behandelt werden.

Der Zahl der verschiedenen Property-Editoren sind jedoch keine erreichbaren Grenzen gesetzt, denn Komponentenentwickler können ihren Komponenten eigene Property-Editoren beifügen. Neben dem einfachen Property-Editor, der nur aus einem Eingabefeld besteht, gibt es in Delphi für die folgenden Property-Typen eigene Editoren:

- Aufzählungstypen: Hier gibt es mehrere Optionen zur Auswahl, jede Option wird über einen Namen angesprochen. Der Editor für Properties diesen Typs ist eine aufklappbare Liste, in der alle Optionen aufgeführt werden. Sie wählen eine neue Option, wie Sie es von einer solchen Liste gewöhnt sind. Die Properties eines Formulars geben viele Beispiele für diese Aufzählungstypen: Für den *Cursor* können Sie im gleichnamigen Property eine Form auswählen, die dieser dann annimmt, wenn er sich über der freien Fläche des Formulars befindet, z.B. *crCross* für ein Zeichenkreuz, *crHourGlass* für den Sanduhr-Zeiger und *crDefault* für den normalen Mauszeiger. Manchmal ist es auch möglich, die Listenvorgaben zu ignorieren: So kann Delphi für das Property *Color* nicht alle 16,7 Millionen möglichen Farbwerte, sondern nur die am häufigsten verwendeten in einer Liste anbieten (*Color* ist allerdings aus Object-Pascal-Sicht auch kein Aufzählungstyp). Sie können daher mit einem Doppelklick auf das Editierfeld dieses Properties einen Farbauswahldialog öffnen oder die Farbe direkt als Zahlenwert angeben, z.B. *$FFAF55* für einen seltenen Blauton. In einigen aufklappbaren Listen werden auch Grafiken angezeigt, so etwa bei *Color* die Farbe oder bei *Cursor* ein Bild der gewählten Mauszeigerform (ab Delphi 5).

- Referenzen auf Komponenten: In manchen Properties geben Sie eine Komponente des Formulars an, z.B. in den oben verwendeten Properties *FocusControl* und *ActiveControl*. Auch hier steht Ihnen eine aufklappbare Liste zur Verfügung, die Ihnen alle in Frage kommenden Komponenten zur Wahl stellt.

  Diese Referenz-Properties werden seit Delphi 6 standardmäßig im Farbton »Maroon« hervorgehoben; zusätzlich haben Sie hier die Möglichkeit, die Properties der verknüpften Komponente zu editieren, wie bei den nachfolgend besprochenen geschachtelten Properties.

- Objekte mit geschachtelten Properties (siehe Abbildung 1.4): Wenn manche Komponenten all ihre Optionen in der Objektinspektor-Liste angäben, würde diese sehr unübersichtlich werden. Daher sind manche Properties standardmäßig nicht sichtbar, sondern einem Eintrag untergeordnet, der durch ein kleines Plus-Zeichen links neben dem Namen markiert ist. Durch einen Doppelklick auf dieses Zeichen können Sie diesen Zweig der Liste expandieren, so dass alle Unterpunkte in der Liste dargestellt werden. Zu einem Zweig zusammengefasst werden meistens Proper-

ties, die zu einem gemeinsamen Aufgabengebiet gehören, oder einzelne Flags, falls es sehr viele davon gibt (siehe nächster Punkt).

Ein Beispiel für die erste Variante sind die *Font*-Properties vieler Komponenten wie auch des Formulars. Zu den Eigenschaften einer Schriftart gehören die Einstellungen, die Sie auch in einer Schriftauswahl-Dialogbox vornehmen können. Sie haben in diesem Fall sogar die Wahl zwischen einer manuellen Eingabe aller Attribute in die einzelnen Property-Felder und einer bequemen Auswahl aus einer Schriftauswahl-Dialogbox. Programmtechnisch handelt es sich bei den Properties wie *Font* um fest in die Komponente integrierte Objekte, die ihrerseits wieder Properties besitzen.

- Mengen von Flags (siehe Abbildung 1.4 rechts): Ebenfalls als expandierbare Zweige dargestellt werden Listen von Flags (ein- und ausschaltbare Properties), beispielsweise das Property *BorderIcons* des Formulars. In diesem Property wählen Sie aus, ob Systemmenü und Schalter zum Maximieren und Minimieren in der Titelleiste des Fensters erscheinen sollen, wobei Sie jedes Icon separat an- und abstellen können. Bei dieser Property-Art können Sie die Flags auch im Feld des übergeordneten Eintrags an- und ausschalten. In diesem werden die Flags so dargestellt wie eine Menge in Object Pascal, beispielsweise *[biMinimize, biMaximize]*, falls diese beiden Flags eingeschaltet, *biSystemMenu* aber inaktiv ist. Um Flags einzeln umzuschalten, genügt ein Doppelklick auf das entsprechende Unterfeld (das dann als aufklappbare Liste mit den Werten *True* (»wahr«, eingeschaltet) und *False* (abgeschaltet) realisiert ist).

- Properties, die mit speziellen Dialogen editiert werden können, manchmal *nur* mit Hilfe eines Dialogs. Zum Beispiel: Properties, die einen Dateinamen angeben, haben meistens eine Dateiauswahlbox als Editor; eine Schriftwahldialogbox erleichtert die Einstellung des Properties *Font*. Sie rufen diese Dialogboxen auf, indem Sie den Schalter mit den drei Punkten rechts neben dem Wert des Properties anklicken. Weitere Beispiele für derartige Dialogboxen sind der Stringlisten-Editor (z.B. für das *Tabs*-Property der *TabSet*-Komponente oder die *Items* einer *ListBox* oder *ComboBox*) oder der Grafik-Editor (z.B. für das *Picture*-Property der *Image*-Komponenten oder das *Glyph*-Property der Bitmapschalter). Als Entwickler von Komponenten können Sie Ihre eigenen Dialogboxen als Property-Editor in die IDE einbinden.

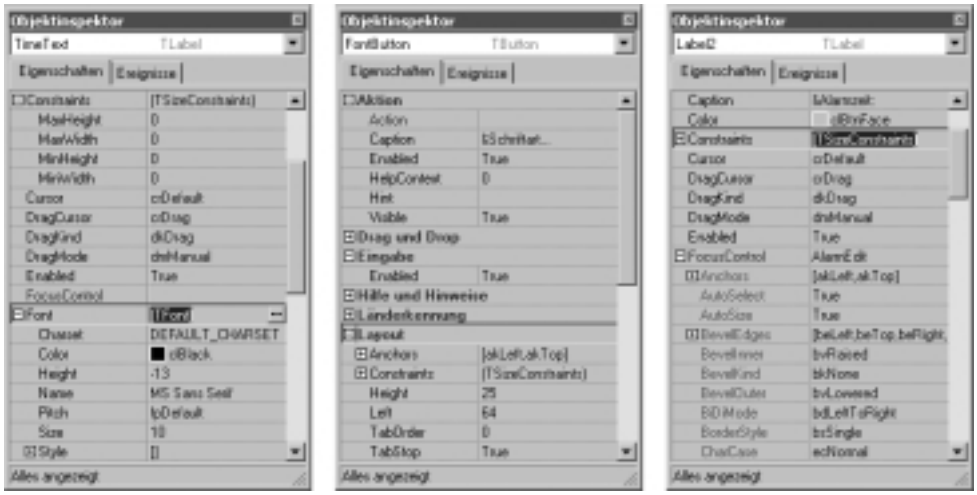


Abbildung 1.4: Geschachtelte Properties und nach Kategorien sortierte Properties, rechts die in Delphi 6 neu eingeführte Darstellung von Properties einer verknüpften Komponente (hier TLabel.FocusControl)

1.4.6 Das Objekt-Hierarchie-Fenster

Die wichtigste Neuerung von Delphi 6, die den Entwurf von Formularen betrifft, ist das Fenster *Objekt-Hierarchie* (Abbildung 1.5). Im Gegensatz zur »eindimensionalen« alphabetischen Liste aller Komponenten des aktuellen Formulars, die Sie im Objektinspektor aufklappen können, wird hier auch sichtbar, welche Komponenten in anderen Komponenten enthalten sind. Im bisherigen kleinen Beispielprogramm kommt bis auf die Tatsache, dass alle Komponenten im Formular enthalten sind, nur eine derartige Beziehungen vor: das Beschriftungselement *Label2* ist Kindelement von *Panel1*. Wenn Sie mit komplexeren Formularen mit vielen verschachtelten Komponenten arbeiten, hat das Objekt-Hierarchie-Fenster drei große Vorteile:

- Es verschafft Ihnen einen Überblick über die logische Struktur eines Formulars.
- Es erlaubt Ihnen, einzelne oder mehrere Komponenten, die nicht mit der Maus im Formular anklickbar sind, gezielt zu selektieren (eine *TPanel*-Komponente, die ganz durch andere Komponenten ausgefüllt wird, ist beispielsweise im Formular nicht per Maus selektierbar).
- Es gestattet Ihnen, Komponenten per Drag&Drop umzugruppieren, also etwa einen Schalter von einer Seite eines *PageControls* auf eine andere zu verschieben – ein Ziel, das Sie ohne die Objekt-Hierarchie nur relativ umständlich mit Ausschneiden und Einfügen erreichen würden.

Die in der Abbildung gezeigte Formular-Struktur stammt aus dem Beispielprogramm *NBDemo* aus Kapitel 3.5.4. Es enthält eine *TPageControl*-Komponente mit vier Seiten.

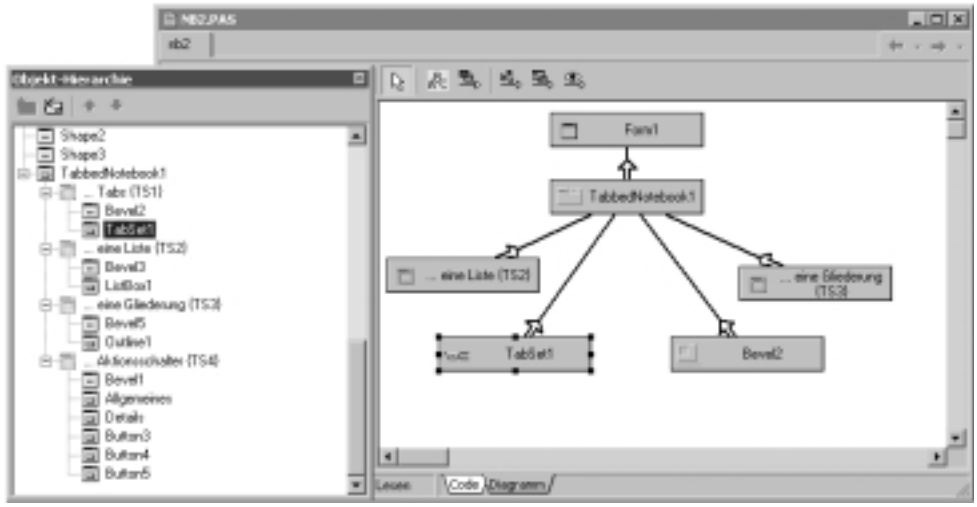


Abbildung 1.5: Eine verschachtelte Komponentengruppe im Objekt-Hierarchie-Fenster sowie eine dazugehörige grafische Darstellung im Diagramm (TabSet1 und Bevel2 sind Kindelemente von TabbedNotebook1 und sind daher im Diagramm direkt mit diesem verbunden).

Da die einzelnen Seiten nicht als eigenständige Komponenten zählen (Sie finden etwa eine einzelne Seite nicht als Komponente in der Komponentenpalette), werden sie in der Objekt-Hierarchie »halb-transparent« dargestellt. Die auf den Seiten befindlichen Komponenten werden wieder normal und eine Ebene tiefer dargestellt.

Hinweis: Das Beispielprogramm *NBDemo* zeigt auch eine kleine Beschränkung des Objekt-Hierarchie-Fensters, was die Zusammenarbeit mit älteren Komponenten betrifft: Die Delphi-1-Kompatibilitäts-Komponente *TNotebook* wird nicht adäquat behandelt, denn ihre einzelnen Seiten und die darin befindlichen Komponenten werden nicht hierarchisch, sondern lediglich als Unterknoten des Formulars dargestellt.

Objekt-Hierarchien jenseits von Formularen

Die weitere Funktionalität des Objekt-Hierarchie-Fensters erklärt sich aus seiner Entwicklungsgeschichte, die in die Datenmodule von Delphi 5 zurückführt. Um die logischen Beziehungen von Datenbank-Sessions, Datenmengen, *DataSource*-Komponenten etc. besser zu veranschaulichen, war nämlich in Delphi 5 eine Hierarchiedarstellung in die Datenmodul-Fenster integriert. Diese Funktion hat Borland nun verallgemeinert und in das Objekt-Hierarchie-Fenster ausgelagert, so dass sie nun auch für den Formularentwurf zur Verfügung steht.

Kapitel 7 wird bei mehreren Gelegenheiten auf die Objekthierarchie einer Datenbankanwendung bzw. auf die dafür erstellbaren Datendiagramme eingehen. An dieser Stelle sei angemerkt, dass Sie in Delphi 6 (jedoch erst ab der Professional-Ausgabe) Diagramme auch für die Komponenten eines Formulars erstellen können, wie in Abbildung 1.5 gezeigt. Das Diagramm zeigt zunächst eine Reihe von Eltern-Kind-Beziehungen (die Seiten des *PageControls* sind *Parent* der in ihnen enthaltenen Komponenten), die durch einen Pfeil symbolisiert sind. Eine andere Pfeilart (schwarzer ausgefüllter Pfeil) weist darauf hin, dass ein Property der einen Komponente die mit dem Pfeil verbundene Komponente referenziert (im Beispiel verweist das *ActivePage*-Property des *PageControls* auf die gerade aufgeschlagene Seite).

Das Diagramm wurde erstellt, indem alle darin vorkommenden Komponenten in der Objekthierarchie markiert (in Verbindung mit Shift können Sie mehrere Komponenten gleichzeitig markieren) und auf das Diagramm gezogen wurden. Danach wurden die einzelnen Komponenten im Diagramm manuell so angeordnet wie in der Abbildung zu sehen. (Dies war übrigens die einzige dem Autor begegnete Situation, bei der Delphi 6 einmal rettungslos im Nirvana verschwunden ist; eine Speicherung des Projekts vorher kann also nicht schaden ...)

1.4.7 Menüs

Zum Editieren stellt Delphi einen Menü-Designer zur Verfügung, der auf den Menükomponenten ganz links auf der ersten Seite der Komponentenpalette basiert. Um ein Hauptmenü zu erhalten, nehmen Sie eine Komponente des Typs *TMainMenu* in Ihr Formular auf.

Hinweis: Ab der Professional-Version von Delphi 6 werden Sie möglicherweise lieber die neuen Aktions-Menüleisten verwenden, die in Kapitel 4.6.5 erläutert werden. Der hier besprochene Menüeditor ist bei diesen Leisten nicht mehr erforderlich!

Der Aufbau eines Menüs verbirgt sich hinter dem Property *Items* im Objektinspektor. Um diese Eigenschaft zu verändern, d.h. also, den Menüeditor aufzurufen, können Sie entweder den zugehörigen Property-Editor aufrufen oder auf die Menükomponente im Formular doppelklicken.

Minikomponenten bilden ein Menü

Durch die Definition eines Menüs lösen Sie eine Schwemme von Minikomponenten aus: Jeder Menüpunkt ist eine eigene Komponente, die jedoch nicht mühsam mit der Maus aus einer Palette herantransportiert werden muss, sondern automatisch erstellt wird. Der Objektinspektor ist lediglich dazu da, dass Sie dort die Attribute eines

Menüpunkts (ob er markiert oder grau dargestellt ist, ob es sich um eine Trennlinie handelt etc.) einstellen können, und dazu, ein Eingabefeld für den Menütext zur Verfügung zu stellen. Parallel dazu wird das Menü im Menü-Designer visuell am Bildschirm aufgebaut, so, wie es später zur Programmlaufzeit aussieht (siehe Abbildung 1.6).

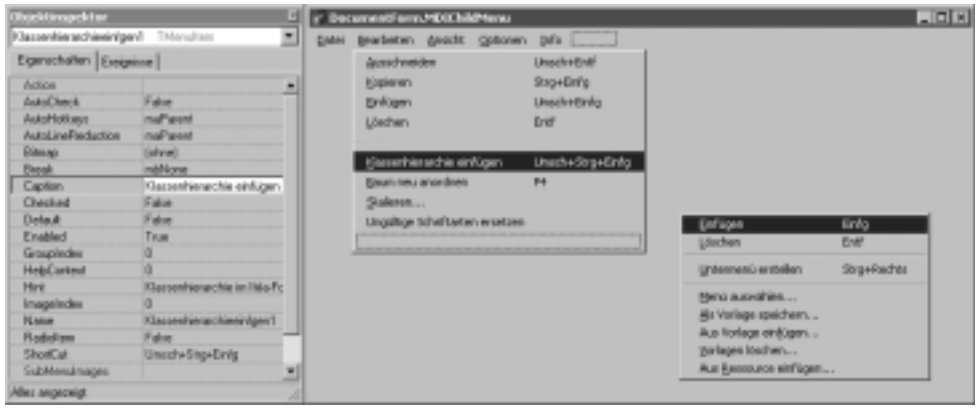


Abbildung 1.6: Der Menü-Designer

An den Stellen, wo neue Menüpunkte eingegeben werden können, zeigt der Menü-Designer zusätzlich einen leeren Menüpunkt als Platzhalter an (am Anfang gibt es nur links oben ein Exemplar davon). Im Menü-Designer können Sie später mit der Maus beliebige Punkte zur Bearbeitung auswählen oder sogar per Drag&Drop verschieben.

Effektive Bedienung des Menü-Designers

Um das gesamte Menü am schnellsten eingeben zu können, müssen Sie sich dem Wechselspiel von Objektinspektor und Menü-Designer anpassen. Angenommen, Sie haben gerade den Designer für ein neues Menü aufgerufen. Dieser ist dann standardmäßig das aktive Fenster. Geben Sie dort ein Zeichen ein, so leitet der Designer dieses automatisch an den Objektinspektor weiter, in dem praktischerweise gleich das *Caption*-Property aktiviert ist (sofern Sie vorher kein anderes Property ausgewählt haben). Wenn Sie in *Caption* den Text für den Menüpunkt eingeben und ↵ drücken, befördern Sie den eingegebenen Text automatisch in das Menü und erreichen, dass darunter ein neuer leerer Menüpunkt angelegt wird, dessen Text Sie sofort eingeben können. Nach dem Drücken von ↵ hat Delphi außerdem das Menüfenster wieder aktiviert, so dass Sie in diesem mit den Pfeiltasten zwischen den Menüpunkten navigieren können. Wollen Sie einfach nur unter dem zuletzt eingegebenen Menüpunkt weitermachen, genügt es wieder, einfach ein Zeichen einzugeben, Delphi aktiviert dann sofort wieder den Objektinspektor.

Um einen neuen Hauptmenüpunkt zu erstellen, tragen Sie einen Text in das *Caption*-Property des immer vorhandenen freien Platzhalter-Menüpunkts im Hauptmenü ein. Sobald Sie dies mit ↵ bestätigt haben, öffnet sich unter dem neuen Punkt ein neues Menü mit einem Platzhaltereintrag und der Platzhalter des Hauptmenüs wandert eine Position nach rechts.

Um verschachtelte Untermenüs zu erzeugen, wählen Sie zunächst den Untermenüpunkt aus, der zu einem weiteren Menü führen soll. Wählen Sie dann aus dem lokalen Menü des Menü-Designers den Punkt UNTERMENÜ ERSTELLEN.

Um ein bestimmtes Zeichen des Menütextes zu unterstreichen und gleichzeitig für das geöffnete Menü zu einem Tastenkürzel zu machen, geben Sie vor diesem das Zeichen '&' ein.

Trennlinien zur Gliederung eines Menüs erhalten Sie, indem Sie einfach als *Caption* ein »-« eingeben.

Menüpunkt-Properties

Die weiteren wichtigen Editieroperationen im Menü-Designer (Löschen und Verschieben von Menüpunkten, Einfügen von Schablonen und Menü-Ressourcen, siehe lokales Menü des Menü-Designers) sollen hier aufgrund ihrer Einfachheit nicht weiter erläutert werden. Interessant sind vor allem weitere Properties, die die Menüpunkte neben dem *Caption*-Property zu bieten haben:

- *Action*: Hier geben Sie optional ein Aktionsobjekt an, das beim Anwählen des Menüpunktes ausgeführt werden soll (siehe Kapitel 4.6.4).
- *AutoCheck* (ab Delphi 6) bewirkt, dass der Haken neben dem Menüpunkt (Property *Checked*) automatisch an- bzw. abgeschaltet wird, jedes Mal wenn der Benutzer den Menüpunkt ausführt.
- *AutoHotkeys* (ab Delphi 5) gibt es als Property für das gesamte Menu (*TMainMenu* oder *TPopupMenu*) und als Property für die einzelnen Menüpunkte (dann bezieht es sich auf ein unter diesem Menüpunkt befindliches Untermenü). *AutoHotkeys* sorgt, auf *maAutomatic* gesetzt, für eine automatische Zuweisung von Tastenkürzeln zu allen Menüpunkten des Menüs oder Untermenüs, denen Sie nicht manuell zu einem solchen Tastenkürzel verholfen haben (durch vorangestelltes '&' im *Caption*-Property). Falls ein Tastenkürzel mehrfach vorkommt, wird diese Situation durch Änderung der Duplikate bereinigt. Sie können das Property auch auf *maParent* setzen, um das Verhalten des übergeordneten Menüpunktes bzw. Menüs zu erben.
- *AutoLineReduction* (ab Delphi 5) verfährt wie *AutoHotkeys*, um doppelte Trennlinien (Linien, die unmittelbar auf eine andere Linie folgen) zu entfernen.

- *Bitmap* dient zur Speicherung eines Bildsymbols, das neben dem Menüpunkt eingeblendet werden soll (im Menü-Designer ist es allerdings noch nicht sichtbar). Alternative dazu: Property *ImageIndex*.
- *Break:* Hiermit können Sie Zeilenumbrüche in Menüzeilen und Spaltenumbrüche in den Spalten bewirken. Diese werden vor dem Menüpunkt eingefügt, der das entsprechende Property besitzt. Sie werden nicht im Menü-Designer, sondern im Formular dargestellt, wenn das bearbeitete Menü im *Menu*-Property des Formulars eingestellt ist.
- *Checked* sorgt, wenn es eingeschaltet ist, für die Markierung des Menüpunkts. Für die Umschaltung der Markierung zur Laufzeit müssen Sie im Programm sorgen, Beispiele finden Sie in Kapitel 5.2.6.
- *Default* macht einen Menüpunkt zur Voreinstellung eines Menüs. Sein Text wird dann durch Fettschrift hervorgehoben und er kann durch einen Doppelklick auf den übergeordneten Menüpunkt aufgerufen werden.
- *Enabled* sind per Voreinstellung alle Menüpunkte. Schalten Sie dieses Flag aus, ist der Menüpunkt nicht mehr anwählbar, aber normalerweise noch sichtbar und lesbar, falls die Farbeinstellung für deaktivierte Menüpunkte in der Windows-Systemsteuerung nicht anderweitig manipuliert wurde.
- *GroupIndex* spielt erst beim Verschmelzen von Menüs in MDI-Anwendungen eine Rolle und wird in Kapitel 5.7.3 erklärt.
- *HelpContext* verknüpft den Menüpunkt über eine Kennzahl mit einer Seite der Hilfedatei, die Sie Ihrer Anwendung zuordnen können (siehe Property *Application.HelpFile*).
- In *Hint* geben Sie einen Hinweistext für den Menüpunkt an, den Sie zur Laufzeit sichtbar machen können, wie in Kapitel 1.9.5 gezeigt.
- *ImageIndex* kann einen Index in eine Bilderliste (*TImageList*) angeben und so wie das *Bitmap*-Property dazu führen, dass das Menü zur Laufzeit mit einem Bildsymbol versehen wird. Bilderlisten werden im Zusammenhang mit der *TListView*-Komponente in Kapitel 1.9.5 und genauer in Kapitel 3.6.2 erläutert.
- *Name* wird von Delphi automatisch auf eine besondere Variation des *Caption*-Textes gesetzt, gefolgt von der obligatorischen Nummerierung. Für Änderungen des Namens gilt dasselbe wie bei den Namen aller anderen Komponenten.
- *RadioItem*: Wenn Sie diese auf *True* setzen, wird die Markierung des Menüpunktes nicht mit einem Häkchen, sondern mit einer an einen Radioschalter erinnernden Kreisfläche dargestellt. Auf diese Weise können Sie auch in einem Menü die Funktion von Radioschaltern simulieren (siehe Komponente *TRadioGroup* bzw. *TRadioButton*). Siehe auch Kapitel 5.2.6, Abschnitt *Menümarkierung und -deaktivierung*.

- *Shortcut* gibt das Tastenkürzel an, über das der Menüpunkt auch dann aufgerufen werden kann, wenn das Menü nicht gerade angezeigt wird. Wenn Sie hier eine Tastenkombination aus der Liste auswählen, erreichen Sie zweierlei: Die Tastenkombination wird im Menü angezeigt und zugleich als funktionsfähiges Tastenkürzel im Programm installiert.
- *SubMenuImages* (ab Delphi 5) ist für Menüpunkte gedacht, die zu einem Untermenü führen. Sie weist auf eine *TImageList*-Komponente, welche die Bildsymbole enthält, die in den *ImageIndex*-Properties der Untermenüpunkte referenziert werden.
- *Tag* ist keinem besonderen Zweck verpflichtet und steht Ihnen damit uneingeschränkt für eigene Pläne zur Verfügung. Ein Beispiel für den Einsatz der *Tag*-Properties anderer Komponenten finden Sie in Kapitel 3.5.2.
- *Visible* schaltet zwischen Sichtbarkeit und Unsichtbarkeit des Menüpunkts um.

Menübefehle ausführen

Das Einzige, was dem Menü jetzt noch fehlt, ist die Fähigkeit, die Befehle auch auszuführen. Dies geschieht über das *OnClick*-Event, das jeder einzelne Menüpunkt besitzt. Die beiden schnellsten Wege, Anweisungen einzugeben, die mit einem Menüpunkt verknüpft werden sollen, sind:

- Doppelklicken Sie im Menü-Designer auf den Menüpunkt.
- Oder wählen Sie den Punkt (zur Entwurfszeit) im Formular aus (falls das Formular das Menü nicht anzeigt, müssen Sie zuerst das Property *TForm.Menu* auf die Menükomponente setzen).

In beiden Fällen gelangen Sie direkt in den Code-Editor, in dem bereits ein automatisch erzeugter Methodenrumpf (oder eine früher schon erzeugte Methode) wartet. Mehr zum Verknüpfen von Ereignissen mit Methoden erfahren Sie im Kapitel über die Ereignisbehandlung (Kapitel 1.5).

Wichtige Menü-Themen sind auch das Erstellen, Verändern und Verschmelzen von Menüs zur Laufzeit des Programms. Sie werden in den Kapiteln 4.6, 5.2.6 und 5.7.3 besprochen.

1.4.8 Grafiken

Zum Erstellen von Icons (Symbolen), Bitmaps und Mauszeigerformen gibt es ein kleines Anhängsel an die Delphi-IDE: den *Bildeditor*. Er ist nur so weit in die IDE integriert, dass Sie ihn aus dem TOOLS-Menü heraus aufrufen können. Es ist jedoch auch in der neuesten Generation von Delphi noch nicht möglich, ihn direkt von einem Mauspalettenschalter im Formular aus aufzurufen, um die Grafik dieses Schalters anzupassen.

Dieses Kapitel beschreibt den Bildeditor nur insoweit, wie er für die in diesem Buch vorkommenden Beispielprogramme angewendet werden kann. Dabei konzentrieren sich die folgenden Abschnitte auf die Verwaltung der Ressourcendateien und deren Einbindung in die Delphi-Anwendung, nicht besprochen werden die Zeichenwerkzeuge, mit denen Sie die Bilder erstellen.

Verwaltung von Ressourcendateien

Der Bildeditor kann die folgenden Dateiformate bearbeiten:

- Icon-Dateien mit der Endung `.ico` enthalten ein Icon, das in mehreren Formaten vorliegen kann: als monochromes und als 16-farbiges Icon sowie in den Auflösungen 32*32 und 16*32 Pixel.
- Bitmap-Dateien mit der Endung `.bmp` enthalten eine Bitmap, deren Größe nahezu beliebig sein kann. Delphis Bildeditor kann Bitmaps mit bis zu 256 Farben erzeugen. Um Bitmaps in Properties von Komponenten, z.B. Bitmap-Schaltern, zu laden, müssen diese im .BMP-Format vorliegen, selbst wenn sie nicht größer sind als ein Icon.
- Cursor-Dateien mit der Endung `.cur` enthalten ein sehr spezielles Bitmap: eine Mauszeigerform, die aus schwarzer und weißer Farbe sowie aus invertierten und transparenten Bereichen besteht.
- Ressourcen-Dateien im Windows-Format tragen die Endung `.res`. Sie können grundsätzlich alle Ressourcentypen enthalten, der Bildeditor unterstützt jedoch nur Bitmaps, Icons und Cursor-Ressourcen. Dateien im .RES-Format lassen sich sehr einfach mit dem Compilerbefehl *$R* in ein Projekt einbinden.
- `dcr` ist die Abkürzung für *Delphi Component Resource*, einer Ressourcendatei, die Ressourcen für Komponenten enthält, welche nur zur Entwurfszeit benötigt werden, darunter vor allem ein Icon, das die Komponenten in der Komponentenpalette vertritt (siehe Kapitel 6.4.4).

Eine besondere Eigenschaft des Bildeditors ist seine Fähigkeit, mehrere Ressourcendateien gleichzeitig zu öffnen. Falls es sich um eine Datei mit verschiedenen Ressourcen handelt (`.res` oder `.dcr`), zeigt der Editor zu jeder Datei ein Übersichtsfenster wie in Abbildung 1.7 unter der Überschrift *Unbenannt1.RES* an. Dieses enthält für jeden der drei schon genannten Ressourcentypen einen Abschnitt, der alle Ressourcen des Typs auflistet. Über das Kontextmenü können Sie auf einfache Weise neue Ressourcen erstellen und alte bearbeiten, umbenennen oder löschen.

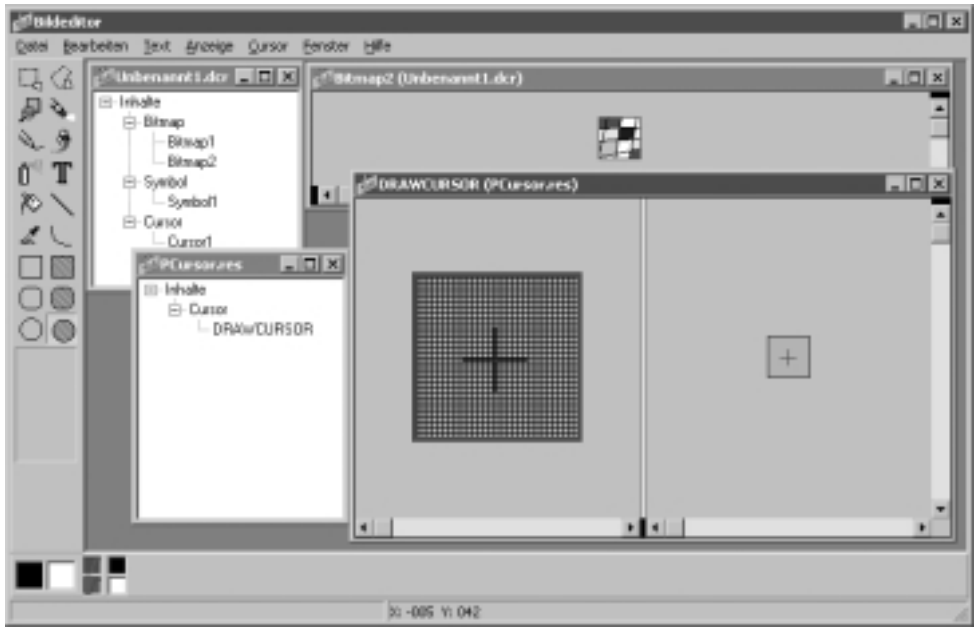


Abbildung 1.7: Der Bildeditor

Bitmaps

Die Beispielanwendung *TreeDesigner* aus Kapitel 5 verwendet mehrere Bitmaps für seine Mauspaletten-Schalter. Mauspaletten-Schalter haben ein Property *Glyph*, in das Sie eine Bitmap laden können, die auf der Oberfläche des Schalters gezeichnet wird. Drücken Sie im Objektinspektor den Schalter neben dem Editierfeld von *Glyph*, um eine Dialogbox zu erhalten, mit der Sie eine Bitmap-Datei laden können.

Wenn Sie in Ihrer Anwendung einfach so feste Bitmaps einbinden wollen (als Illustration, nicht als Schalter), bietet sich die Komponente *TImage* an. Wenn Sie Bitmaps selbst zeichnen wollen, können Sie dies mit *TCanvas* (Kapitel 4.4) und der Klasse *TBitmap* (Kapitel 4.5.2) tun.

Bitmaps im BMP-Format können Sie im Bildeditor erzeugen, indem Sie DATEI | NEU wählen und den Typ *Bitmap-Datei* auswählen oder indem Sie eine Bitmap aus einer Ressourcendatei mit DATEI | SPEICHERN UNTER... in eine separate Datei auslagern.

Zum Editieren der Bitmap stellt der Bildeditor ein Fenster zur Verfügung, das den eigentlichen Bildeditor darstellt, zu ihm gehören eine Farbpalette und eine Werkzeugleiste (siehe Abbildung 1.7). Wie schon angekündigt, kann und braucht die Bedienung dieses Editors hier nicht erklärt zu werden.

Ressourcendateien einbinden *R6*

Das TreeDesigner-Projekt aus Kapitel 5 verwendet eine Mauszeiger-Ressource, um die Form des Mauszeigers über der Zeichenfläche in ein Fadenkreuz zu verändern (das etwas kleiner ist als das vordefinierte Fadenkreuz). Zur Laufzeit muss die Mauszeiger-Ressource, wie in Kapitel 3.3.3 gezeigt, mit der Anweisung *LoadCursor* geladen werden. Dazu muss die Ressource jedoch vorher in die EXE-Datei aufgenommen worden sein. Damit Delphi Ressourcendateien automatisch zur EXE-Datei hinzubindet, fügen Sie eine Zeile wie die folgende in den Pascal-Quelltext ein:

```
{$R FCURSOR.RES}
```

Delphi erlaubt hier nur das Format `.res`. Falls Sie beispielsweise eine `.cur`-Datei mit einer einzelnen Mauszeigerform haben, müssen Sie diese erst in das `.res`-Format umwandeln (DATEI | SPEICHERN UNTER... im Bildeditor). Wichtig ist auch, dass der Name der Ressourcendatei *nicht* mit dem Namen des Projekts übereinstimmt, denn unter diesem Namen speichert Delphi bereits eine Standard-Ressourcendatei für das Projekt, in der sich meistens nur das Icon für die Anwendung befindet.

Die Ressourcendatei für den TreeDesigner heißt `FCursor.res` und wird mit der oben gezeigten Compileranweisung in das Projekt eingebunden. Wir werfen einen kurzen Blick auf ihren Inhalt.

Mauszeiger *R7*

Der Bildeditor stellt Mauszeiger in einem Editorfenster dar, das Sie schon von den Bitmaps her kennen. Der Unterschied liegt in der Farbpalette, die hier nur vier Farben enthält. Unter den Farben Schwarz und Weiß befinden sich

- die transparente »Farbe«. Sie ist die Hintergrundfarbe für Mauszeiger und deckt, wenn der Mauszeiger einmal aktiv ist, den Bildschirminhalt nicht ab. Nach dem Erzeugen einer neuen Mauszeigerform ist die gesamte Zeichenfläche mit dieser Transparenz gefüllt, was bedeutet, dass der Mauszeiger noch völlig unsichtbar ist.
- die invertierte »Farbe«. Pixel, die im Editor diese Farbe aufweisen, werden in der Praxis invertiert dargestellt. Ein Beispiel für einen Mauszeiger mit invertierten Pixeln ist der Standard-Cursor *crIBeam*, der erscheint, wenn Sie die Maus über Standardeditierfelder von Windows bewegen.

Neben diesen beiden Scheinfarben gibt es bei Mauszeigerformen noch eine weitere Kleinigkeit zu bedenken: Der *Kontaktpunkt* ist der Punkt, mit dem die Maus quasi den Bildschirm berührt. Immer, wenn Sie es im Programm mit der Position der Maus zu tun bekommen, ist damit die Position dieses Kontaktpunkts gemeint. Ein falsch eingestellter Kontaktpunkt kann dazu führen, dass die Maus an eine andere Stelle, als vom Benutzer beabsichtigt, klickt. Sie stellen den Kontaktpunkt über den Menüpunkt CURSOR | SENSITIVE ZONE FESTLEGEN... ein.

Der in der Abbildung gezeigte Mauszeiger für den TreeDesigner verwendet statt echten Farben nur die Invertierung. Wie der Cursor *crIBeam* besteht auch er nur aus ein Pixel breiten Linien, die unsichtbar werden würden, wenn sie farbig wären und sich auf einem Hintergrund mit der gleichen Farbe bewegen würden.

Spezielle Delphi-Ressourcendateien

Abgesehen davon, dass Sie normalerweise nur Icons für Komponenten darin speichern, sind DCR-Dateien genauso zu handhaben wie RES-Dateien. Tatsächlich handelt es sich sogar um RES-Dateien, bei denen lediglich die Endung geändert wurde, damit Delphi sie eindeutig von den Ressourcendateien unterscheiden kann, die auch zur Laufzeit und nicht nur wie die DCR-Dateien zur Entwurfszeit geladen werden sollen.

1.5 Ereignisse

In diesem Kapitel geht es darum, wie Sie die Ereignisse bearbeiten und wie Delphi die Erstellung und Verwaltung der Ereignisbearbeitungsmethoden vereinfacht. Wie schon in Kapitel 1.1 beschrieben, sind Ereignisse die Elemente, die eine Delphi-Anwendung antreiben. Für dieses Kapitel schlagen wir also die Seite *Ereignisse* des Objektinspektors auf, auf der sich alle häufig benötigten Ereignisse befinden, die in Zusammenhang mit der gerade bearbeiteten Komponente auftreten können.

1.5.1 Einführung in die Ereignisbearbeitung

Die Ereignisse, die nur das Formular, also keine seiner untergeordneten Komponenten betreffen, finden Sie auf der Ereignisseite des Objektinspektors, wenn Sie das Formular auswählen (Abbildung 1.8). Dazu gehören Ereignisse, die der Benutzer mit seinen Eingaben direkt auslöst (z.B. *OnClick*, *OnDblClk*, *OnKeyPress* für Mausklicks und Tastatureingaben) und solche, die auch auf andere Weise hervorgerufen werden können: *OnDeactivate* tritt beispielsweise als Nebeneffekt auf, wenn der Benutzer ein anderes Fenster aktiviert, *OnPaint* wird vom System aufgerufen, wenn Fensterteile neu gezeichnet werden müssen, und das *OnDestroy*-Ereignis informiert Sie darüber, dass das Fenster gerade geschlossen wird. Das Programm selbst kann diese Ereignisse ebenfalls auslösen, indem es beispielsweise ein anderes Fenster in den Vordergrund holt, ein Fenster schließt oder andere Aktionen durchführt.

Verknüpfung von Ereignissen und Methoden

Die Spalten der Ereignisseite des Objektinspektors sind so aufgebaut wie die der Eigenschaftsseite. Am Anfang eines Projekts ist die rechte Spalte der Ereignisse leer, d.h., dass in Ihrer Anwendung bisher auf kein Ereignis eine spezielle Antwort vorgesehen ist.

Grundsätzlich gehört in die rechte Spalte der Name eines Programmteils, der das Ereignis bearbeitet (eine *Ereignisbearbeitungsmethode*). Sie können die Ereignisse schon bearbeiten, ohne sich vorher tiefer gehend mit dem Aufbau von Methoden und der objektorientierten Programmierung beschäftigt zu haben, denn für einfache Aufgaben benötigen Sie nur sehr geringe Pascal-Kenntnisse. Um schnell eigene Experimente machen zu können, genügt es, die formlose Zusammenfassung dessen, was Sie in einer Methode tun können, in Kapitel 1.5.2 (*Schnellübersicht: Pascal für Ereignisbearbeitungsmethoden*) zu lesen. Das gesamte Potenzial der Ereignisbearbeitungsmethoden können Sie mit den Object-Pascal- und Delphi-Kenntnissen ausschöpfen, die ab Kapitel 2 vermittelt werden.

Eine Beispielmethode

Nehmen wir an, Ihre Anwendung soll auf jeden Mausklick ins Innere des Hauptfensters reagieren. Sie benötigen also Code, der das Ereignis *OnClick* bearbeitet. Durch einen Doppelklick im Objektinspektor auf das leere Feld neben dem Ereignisnamen *OnClick* erreichen Sie, dass Delphi eine leere Bearbeitungsmethode für das Mausklick-Ereignis anlegt (Abbildung 1.8). Delphi gibt dieser Methode einen Namen und trägt ihn in das Feld des Objektinspektors ein, wo Sie ihn auch ändern können. Darüber hinaus holt Delphi den Quelltexteditor in den Vordergrund und zeigt den vorgegebenen Methodenrumpf an:

```
procedure TForm1.FormClick(Sender: TObject);
begin

end;
```

Delphi setzt auch die Eingabemarkierung bereits an die geeignete Stelle zwischen *begin* und *end*, so dass Sie sofort Anweisungen einfügen und von den mächtigen Mitteln der Sprache Object Pascal Gebrauch machen können. Da diese erst in Kapitel 2 besprochen werden, müssen wir uns an dieser Stelle etwas einschränken. Auf die (als *Parameter* bezeichnete) Angabe »*Sender: TObject*«, die Sie im so genannten *Methodenkopf* finden können, kommen wir in Kapitel 1.8.6 zu sprechen.

An dieser Stelle nur ein kurzes Beispiel: Angenommen, die Farbe des Fensters soll bei jedem Mausklick zwischen Silbergrau und Weiß wechseln. Ergänzen Sie dazu den Methodenrumpf mit dem folgenden Inhalt:

```
procedure TForm1.FormClick(Sender: TObject);
begin
  if Color <> clSilver
    then Color := clSilver
  else Color := clWhite;
end;
```

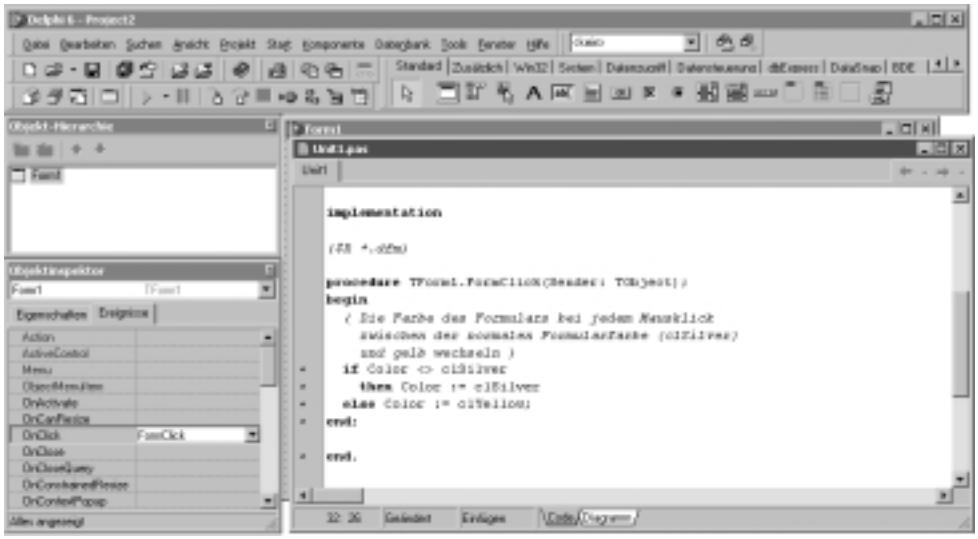


Abbildung 1.8: Eine Methode für das OnClick-Ereignis

(Erklärung: Die *if*-Anweisung überprüft, welche Farbe das Fenster zurzeit hat. Dazu vergleicht sie den Wert des Properties *Color* mit dem Wert *clSilver*. Wenn das Fenster nicht die Farbe *clSilver* hat, gibt die zweite Zeile ihm diese Farbe, ansonsten wechselt die dritte Zeile zur Farbe Weiß.)

Der erste Programmstart

Sie können die Anwendung sofort starten, indem Sie F9 (Menüpunkt START | START) drücken. Ohne am Bildschirm durch Meldungen großes Aufsehen zu erzeugen, übersetzt Delphi nun Ihr Programm und schreibt den Entwurf Ihres Formulars auf die Festplatte. Da dieser Vorgang bei kleinen Programmen wie diesem sehr schnell geht, fällt er gar nicht auf (falls es doch zu merklichen Verzögerungen kommt, ist Speicherknappheit oder hohe anderweitige Prozessorauslastung der Grund).

Wenn alles funktioniert hat, können Sie testen, ob das Fenster wie gewünscht auf Mausklicks reagiert. Sie werden feststellen, dass sich die Farbe nach dem ersten Mausklick ändert, aber erst dann, wenn Sie die Maustaste losgelassen haben. Wenn Sie versuchen, eine schnelle Folge von Mausklicks auf das Fenster loszulassen, scheint es, als habe dieses Reaktionsprobleme.

Dies liegt daran, dass *OnClick* ein relativ komplexes Ereignis ist. Es definiert einen Mausklick als das Niederdrücken der Taste und das anschließende Loslassen im selben Fenster. Es berücksichtigt darüber hinaus, dass zwischen zwei Klicks keine zu kurze Zeitspanne liegen darf, denn sonst handelt es sich um einen Doppelklick, und statt

eines zweiten *OnClick*-Ereignisses wird ein *OnDblClk*-Ereignis hervorgerufen (dabei kann das schon gesendete *OnClick*-Ereignis natürlich nicht mehr rückgängig gemacht werden).

Hinweis: Im Menü PROJEKT der IDE befinden sich mit *Name* COMPILIEREN, *Name* ERZEUGEN und SYNTAXPRÜFUNG VON *Name* (Dabei ist *Name* der Name des aktuellen Projekts) drei Punkte, mit denen Sie den Compiler aufrufen können, ohne das Projekt gleichzeitig zu starten. Beim Aufrufen des START-Menüpunkts verhält sich der Compiler wie beim Menüpunkt COMPILIEREN, übersetzt also nur die Dateien, die nicht mehr in einer aktuellen Übersetzung vorliegen. Genauere Informationen dazu gibt Ihnen die Online-Hilfe.

Wahl eines passenderen Ereignisses

Wir ändern das Programm nun so, dass das Fenster bei jedem Drücken der Maustaste sofort reagiert. Dazu müssen wir ein anderes Ereignis suchen, denn *OnClick* ist dazu wie gesehen nicht geeignet. Das reaktionsschnellere Ereignis heißt *OnMouseDown*. Erzeugen Sie durch einen weiteren Doppelklick im Objektinspektor einen neuen Methodenrumpf für dieses Ereignis. Dieser Methodenrumpf unterscheidet sich vom bisherigen außer im Namen durch die Parameter: Neben *Sender* finden Sie hier beispielsweise *X* und *Y*, die die Mausposition anzeigen.

Um den Rumpf zu füllen, verschieben Sie nun die Zeilen, die Sie oben in *FormClick* eingefügt haben, in die Methode *FormMouseDown* (*FormClick* sollte sich im Quelltexteditor direkt über der neuen Methode befinden), oder versuchen Sie es mit der folgenden Variation, deren neue Object-Pascal-Teile nur demonstrativen Zwecken dienen:

```
procedure TForm1.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
const Farbtafel: array[0..9] of TColor =
  (clMaroon, clGreen, clOlive, clTeal, clSilver,
   clRed, clBlue, clLime, clFuchsia, clAqua);
begin
  Color := Farbtafel[random(10)];
end;
```

Diese Methode wählt per Zufallszahl eine aus zehn vorgegebenen Farben aus und weist sie dem *Color*-Property zu. Und egal, wie schnell Sie klicken, die Reaktion des Programms sollte jetzt immer sofort auf das Niederdrücken einer Maustaste erfolgen.

Wegweiser

Mit dem Beispielprogramm und den Ereignissen der Komponenten geht es in Kapitel 1.5.4 weiter, zunächst folgen ein für Umsteiger interessantes Kapitel und ein Blick hinter die Kulissen der gerade besprochenen Delphi-Automatik.

1.5.2 Schnellübersicht: Pascal für Ereignisbearbeitungsmethoden

Falls Sie noch nicht in objektorientiertem Pascal, in C++ oder in einer ähnlichen Sprache programmiert haben, sollen Ihnen die beiden folgenden Aufzählungen einen groben Überblick über die Operationen geben, die Sie innerhalb einer Ereignisbearbeitungsmethode erledigen können.

Eine kleine Vorbemerkung: Sie können zwischen beliebigen Anweisungen, Wörtern und Zahlen Kommentare in den Quelltext einfügen. Kommentare müssen auf eine von zwei Arten vom Rest des Programms abgetrennt werden:

```
{ dies ist die erste Möglichkeit, einen Kommentar einzuschließen. }
(* auch so kann ein Kommentar geklammert werden *)
```

Die erste der beiden folgenden Listen beschreibt die drei verschiedenen Arten von Anweisungen, die Sie verwenden können, die zweite Liste zeigt, auf welche Elemente diese Anweisungen wirken bzw. mit welchen sie arbeiten. Die Anweisungsarten sind:

- *Zuweisung*: Eine sehr häufige Operation ist die Zuweisung eines Wertes an ein Property oder an eine Variable; für die Zuweisung verwenden Sie den Zuweisungsoperator »:=« wie im letzten Beispiel. Auf der rechten Seite der Zuweisung können Sie beliebige Ausdrücke angeben, solange diese sich mit der linken Seite vertragen (z.B. *Breite := 10*SpaltenBreite;*). Oft werden Sie sich dabei auf Properties beziehen wollen. Dies geschieht meistens wie in der gleich folgenden zweiten Liste beschrieben.
- *Methodenaufruf*: Neben der Zuweisung spielt der Aufruf von Methoden (Prozeduren und Funktionen) eine wichtige Rolle. Ein Methodenaufruf ist beispielsweise: *MessageBeep(MB_OK);*
- *Kontrollstrukturen*: Zuweisung und Methodenaufruf wären bereits das Einzige, was Sie innerhalb einer Methode tun könnten, wenn es nicht noch die Kontrollstrukturen und Schlüsselworte gäbe. So können Sie z.B. mit der *if*-Anweisung wie im letzten Beispiel Alternativen abfragen, mit der *for*-Schleife bestimmte Operationen wiederholen lassen oder gar mit *asm* Assemblercode einfügen.

Die obige Aufstellung ist zwar grob, aber umfassend (in dieser Einteilung sind dies die einzigen Arten von Anweisungen, die es gibt). Anders verhält es sich mit der folgenden Liste: Sie zeigt die Elemente, mit denen Sie am häufigsten arbeiten. Alle im folgenden genannten Elemente haben als Gemeinsamkeit, dass sie einen Namen, den so genannten *Bezeichner*, benötigen:

- Um sich auf Properties des Formulars zu beziehen, schreiben Sie einfach deren Namen hin (z.B. *Color := clSilver;* für das *Color*-Property des Formulars).

- Um Properties von Komponenten anzusprechen, stellen Sie den Namen der Komponente, gefolgt von einem Punkt, vor den Namen des anzusprechenden Properties, so weisen Sie z.B. mit *Panel1.Color := clWhite;* nur der Komponente mit dem Namen *Panel1* die weiße Farbe zu.
- Ebenso können Sie zwischen Methoden des Objekts und einer Komponente unterscheiden: Um beispielsweise eine Komponente in den Vordergrund zu holen (entsprechend dem Befehl NACH VORNE SETZEN im lokalen Menü des Formulars), schreiben Sie *Komponente.BringToFront;*. Um das Formular / Fenster selbst vor die anderen Formulare / Fenster zu bringen, schreiben Sie nur *BringToFront;*
- Um innerhalb einer Methode Werte speichern zu können, deklarieren Sie lokale Variablen. Diese sind nur in der Methode gültig und behalten ihre Werte nicht zwischen zwei verschiedenen Aufrufen dieser Methode.
- Um Werte zwischen zwei Ereignissen zu speichern, geben Sie dem Formular eine neue Variable. So kann beispielsweise die *MouseDown*-Methode speichern, zu welcher Zeit die Maustaste gedrückt wurde. Wenn die *MouseUp*-Methode ihre eigene Zeit feststellt, kann sie mit Hilfe der gespeicherten Zeit der *MouseDown*-Methode die Differenz zwischen beiden Zeiten ermitteln und damit errechnen, wie lange die Maustaste gedrückt war.
- Sie könnten in Object Pascal auch jederzeit globale Variablen deklarieren, jedoch ist meistens die Deklaration einer Formularvariablen empfehlenswerter. Object Pascal macht globale Variablen nahezu unnötig.

Ein wichtiger und selbstverständlicher Bestandteil des Programmcodes sind Konstanten wie Zahlen, Zeichenketten ('In Hochkommas') und Bezeichner, die für einen konstanten Wert stehen (z.B. die Namen für die Farben *clBlack* und *clWhite*), insbesondere die Pascal-Namen für »wahr« und »falsch« : *True* und *False*.

Soweit zum Inhalt einer Methode und damit zum Mikrokosmos eines Object-Pascal-Programms. Eine Übersicht über den gesamten Quelltext, den Delphi für das letzte *OnClick/OnMouseDown*-Beispielprogramm fast vollständig automatisch erzeugt hat, finden Sie in der folgenden Beschreibung des Makrokosmos.

1.5.3 Eine Übersicht über die Formular-Unit

Bisher haben Sie nur im Inneren der Ereignisbearbeitungsmethoden mit dem eigentlichen Programmcode zu tun gehabt. Wir sehen uns hier nun den vollständigen Quelltext des Formulars an, allgemeine und weitere Informationen zu Units finden Sie in Kapitel 2.1.7.

Delphi verwaltet für jedes Formular eine Unit, die dafür vorgesehen ist, alles, was mit dem Formular untrennbar logisch verbunden ist, in einer Datei zusammenzufassen

(Delphi selbst erweitert diese Unit nur um Komponenten und Methoden des Formulars; Sie können weitere Hilfsklassen, -typen, -funktionen und andere Dinge darin deklarieren). Wir betrachten zuerst die Unit, die erzeugt wird, wenn Sie einem leeren Formular eine *FormMouseDown*-Methode zuweisen wie im letzten Beispiel.

Die Unit zum Formular

Im oberen Teil der Unit befindet sich die so genannte *Deklaration* des Formulars. In ihr sind alle Bestandteile (Komponenten und eventuell weitere Variablen) und Fähigkeiten (Ereignisbearbeitungsmethoden) des Formulars aufgeführt. Das zuletzt bearbeitete Formular enthält als einziges Element die Methode für das Ereignis *FormMouseDown*:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs;

type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
   if Color <> clSilver
      then Color := clSilver
   else Color := clWhite;
end;

end.
```

Im zweiten Abschnitt der Unit, der durch die Überschrift *implementation* eingeleitet wird, ist die konkrete Implementierung der oben angegebenen Methode(n) untergebracht. Alle Methoden, die in der Deklaration aufgelistet sind, kommen hier ein zweites Mal vor, jedoch diesmal mit dem zugehörigen Code, der in einem von *begin* und *end* eingeschlossenen Block steht.

Weitere Bestandteile dieser Unit sind:

- Die Zeilen *unit...*, *interface*, *implementation* und *end.* bilden das Gerüst einer jeden Object-Pascal-Unit.
- Der *uses*-Abschnitt ist notwendig, um – grob gesagt – die VCL verwenden zu können. Alle Namen, die hier aufgelistet werden, sind ebenfalls *Units*, die zum größten Teil die Funktionalität der VCL bilden.
- Die Deklaration der Variable *Form1: TForm1*. Auf diese Weise wird von der Formularschablone ein Fensterobjekt hergestellt. Warum und wie ein solches nach dem Starten des Programms automatisch angezeigt wird, beschreibt das Kapitel zur Projektverwaltung (1.6.3).
- Die *$R*-Compiler-Anweisung gehört quasi zum Protokoll und sorgt für die Verbindung der Formulardatei mit der EXE-Datei, wenn das Programm übersetzt wird.

Sie können übrigens das Unit-Gerüst, das Delphi für alle neuen Formular-Units verwendet, ändern, indem Sie eine geänderte Formular-Unit in die Objektablage aufnehmen und beim Erstellen eines neuen Formulars kopieren (zum Thema Objektablage siehe Kapitel 1.6.4).

Komponenten in der Formulardeklaration

Wenn Sie dem Formular Komponenten hinzufügen, nimmt Delphi auch diese in die Formulardeklaration auf. Der folgende Ausschnitt ist der Unit des Beispielprogramms entnommen und zeigt die Deklaration der einzelnen Komponenten seines Formulars:

```
type
  TUhrFormular = class(TForm)
    TimeText: TLabel;
    Timer: TTimer;
    FontButton: TButton;
    FontDialog: TFontDialog;
    AlarmEdit: TEdit;
    Panel1: TPanel;
    Label2: TLabel;
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;
```

Die Bereiche *private* und *public* sind für Ihre Formularvariablen und für Nicht-Ereignisbearbeitungsmethoden vorgesehen, die Sie beide manuell an diesen Positionen eintragen müssen.

1.5.4 Die Ereignisse des Beispielprogramms

Wir können nun die Ereignisse der Komponenten in der Uhr-Applikation bearbeiten. Zwei Ereignisse genügen bereits, um die für die erste Version gewünschte Funktionalität zu erhalten: ein Timerereignis, bei dem die Uhrzeit aktualisiert wird, und das Ereignis, das auftritt, wenn der Schriftwahlschalter gedrückt wird (daraufhin wird natürlich die Schriftwahl-Dialogbox aufgerufen).

SysUtils-Funktionen für die Uhrzeit

Zunächst ist die Frage zu klären, woher das Programm weiß, wie viel Uhr es ist und wie es die in nackten Zahlen vorliegende Uhrzeit in eine menschenlesbare Ziffernkette im Format 00:00:00 umwandeln kann. Dies ist mit den Funktionen aus Delphis Standard-Unit *SysUtils* überhaupt kein Problem mehr: Die Funktion *Time* liefert die aktuelle Zeit in einem Format mit der Bezeichnung *TDateTime*. Dieses brauchen Sie nicht zu kennen (ungewöhnlicherweise handelt es sich übrigens um eine Fließkommazahl), denn andere Funktionen derselben Unit sind in der Lage, die *TDateTime*-Angabe in eine Zeichenkette oder in einzelne Zahlenwerte für Stunden, Minuten und Sekunden (und sogar Hundertstelsekunden) umzuwandeln. Um eine Zeichenkette (*String*) zu erhalten, wenden Sie die Funktion *TimeToStr* auf das Ergebnis der Funktion *Time* an:

```
UhrzeitAlsString := TimeToStr(Time);
```

Das Timer-Ereignis

Es kommt jetzt darauf an, die Ausgabe dieses Strings jede Sekunde zu wiederholen. Dazu ist lediglich ein Ereignis erforderlich: das Timer-Ereignis, das nach den vordefinierten Einstellungen genau einmal pro Sekunde ausgelöst wird (um dies zu ändern, können Sie das *Timer*-Property *Interval* anpassen). Wählen Sie die Timer-Komponente aus dem Formular aus, schlagen Sie die Ereignisseite des Objektinspektors auf und doppelklicken Sie neben das *OnTimer*-Feld. In der dadurch entstehenden Methode brauchen Sie nur eine Zeile einzufügen, um die Uhr funktionsfähig zu machen:

```
procedure TUhrFormular.TimerTimer(Sender: TObject);
begin
  TimeText.Caption := TimeToStr(Time);
end;
```

Für die Textausgabe sorgt die Komponente *TimeText*: Der in dieser *Label*-Komponente angezeigte Text befindet sich im Property *Caption* und lässt sich dort beliebig ändern. Die oben gezeigte Anweisung sorgt also dafür, dass die Komponente die neue Uhrzeit

am Bildschirm anzeigt. Es genügt, die aktuelle Zeitangabe einmal pro Sekunde an das Property zu übergeben, wie es die obige Methode tut.

Schon mit dieser ersten Ereignismethode ist die Uhr funktionsfähig, so dass Sie sie mit START | START aktivieren können; nach einem kleinen Einschub über die neuen Programmierhilfen werden wir sie mit Hilfe eines zweiten Ereignisses auch noch interaktiv machen.

Die Programmierhilfen im Editor

Wenn Sie den oben gezeigten Text selbst eingeben und die Delphi-Version 3 oder höher verwenden, erhalten Sie per Voreinstellung kleine Hilfestellungen bei der Eingabe. Selbst bei der winzigen Quelltextzeile im obigen Beispiel werden bereits die beiden wichtigsten der so genannten Programmierhilfen aktiviert, vorausgesetzt, dass Sie an den entsprechenden Stellen des Quelltextes die kurzen Pausen machen, durch die diese Funktionen erst aktiviert werden.

- Im obigen Beispiel öffnet der Editor eine DropDown-Liste, sobald Sie den Punkt hinter *TimeText* eingegeben haben. Die Liste enthält alle in *TimeText* ansprechbaren Properties und Methoden, darunter auch das Property *Caption*, das wir hier benötigen. Sie brauchen in diesem Fall *Caption* nicht ganz auszuschreiben, sondern können schon nach der Eingabe von »C« mit der Eingabetaste den dann in der Liste ausgewählten Text *Caption* in den Editor übernehmen.
- Wenn Sie nach der öffnenden Klammer hinter *TimeToStr* wieder einen Moment zögern, erscheint ein kleines Hinweisfenster, das die Namen und Typen der Parameter der Funktion *TimeToStr* auflistet. Dabei hebt es immer den Parameter, den Sie seiner Meinung nach als Nächstes eingeben sollten, durch Fettschrift hervor. Bei Funktionen mit mehr Parametern als in diesem Beispiel kann dies zu einer sehr nützlichen Hilfe werden. Es ist sogar häufig möglich, auf diese Weise eine Funktion aufzurufen, die man gerade erst durch die eben besprochene DropDown-Liste kennen gelernt hat, ohne die Verwendung der Funktion in der Online-Hilfe nachschlagen zu müssen (eine sinnvolle Benennung der Parameter vorausgesetzt).

Der Schriftwahlschalter

Als zweites Beispiel für eine Ereignisbearbeitungsmethode implementieren wir die Funktion für den Schriftwahlschalter, also ein Ereignis, das vom Benutzer ausgelöst wird. Unter den Ereignissen des Schalters befindet sich, genau wie bei den Formular-Events, ein Ereignis mit dem Namen *OnClick*. Da der Schriftwahldialog nun wirklich erst dann geöffnet werden soll, wenn die Maustaste wieder losgelassen wurde, ist dies das richtige Ereignis. Lassen Sie von Delphi eine Methode dafür anlegen und fügen Sie die folgenden Zeilen ein:

```
procedure TUhrFormular.FontButtonClick(Sender: TObject);
begin
  FontDialog.Font := TimeText.Font;
  FontDialog.Execute;
  TimeText.Font := FontDialog.Font;
end;
```

Auch dieses Mal ist die Komplexität des erforderlichen Codes nicht gerade Furcht erregend. Damit der Schriftdialog beim ersten Aufruf die Schrift anzeigt, die gerade für die Uhrzeitanzeige verwendet wird, setzt die erste Zeile das Schrift-Property des Dialogs auf das der *Label*-Komponente. Daraufhin ruft die zweite Zeile den Dialog über dessen Methode *Execute* auf. Während des Ablaufs von *Execute* wird das *Font*-Property des Dialogs gemäß der Auswahl des Benutzers gesetzt, falls dieser den Dialog mit dem *OK*-Schalter bestätigt. Beendet er den Dialog mit *Abbrechen* oder schließt er ihn über sein Systemmenü, so bleibt *FontDialog.Font* unverändert.

Am Schluss braucht die (möglicherweise geänderte) *FontDialog.Font*-Einstellung nur noch in das Property *Font* der *Label*-Komponente übertragen zu werden, und sofort stellt diese die Uhrzeit in der neuen Schrift dar (sie wartet nicht erst bis zum nächsten *OnTimer*-Ereignis).

Starten des Programms

Sie können das Programm jetzt wieder über den Menüpunkt START | START oder die zugehörige Tastenkombination starten. Wenn Sie zur Laufzeit die Schriftart entsprechend anpassen, können Sie ein Fenster erhalten, das aussieht wie in Abbildung 1.3 gezeigt.

Profitieren von den VCL- und Delphi-Mechanismen

Nach diesen beiden kurzen Methoden werfen wir einen Blick auf die im Hintergrund wirkenden komfortablen Mechanismen, die es uns bisher erspart haben, die Tastatur durch das Schreiben von Programm-Code allzusehr abzunutzen:

- Der Umgang mit den Einstellungen vom Schriftart-Dialog ist extrem einfach, da die *FontDialog*-Komponente die erforderlichen Daten in einem einzigen Property namens *Font* zusammenfasst, das Sie immer ansprechen können, egal, ob der Dialog gerade am Bildschirm sichtbar ist oder nicht. Die anderen Dialogkomponenten von Delphi arbeiten übrigens ähnlich.
- Wie bei allen Komponenten, die Sie zum Entwurfszeitpunkt in das Formular einfügen, wird die Initialisierung (per Konstruktor) und die Freigabe der Komponenten (per Destruktor) von Delphi automatisch durchgeführt.

- Einigen unbequemen Verwaltungsaufwand übernimmt auch die *Timer*-Komponente. Sie trägt die volle Verantwortung für den ordnungsgemäßen Umgang mit den Timer-Ressourcen.
- Ein wichtiger Punkt an den beiden obigen Ereignismethoden ist, dass diese Ereignisse an das Formular gesendet werden, obwohl sie es nicht direkt selbst, sondern nur eines seiner Elemente betreffen. Diese Vorgehensweise ist unter Delphi für alle Komponenten dieselbe: Alle Ereignisse der Komponenten werden an das übergeordnete Formular gesendet.
- Je nachdem, was Sie bisher für Erfahrungen mit der Windows-Programmierung gemacht haben, kann Ihnen das ungewöhnlich vorkommen, da zum Beispiel ein Mausklick von Windows selbst grundsätzlich an das Unterfenster gesendet wird, in dem der Mausklick stattgefunden hat. Da auch Schalter von Windows als Unterfenster verwaltet werden, erhalten sie den Mausklick auch in einer Delphi-Anwendung zuerst. Die VCL sorgt jedoch in jedem Fall dafür, dass das Ereignis auf die bisher besprochene Weise zum Formular gelangt, nachdem es von der Komponente selbst angemessen bearbeitet wurde. Sie haben aber auch immer die Möglichkeit, Nachrichten auf Komponentenebene abzufangen, indem Sie eigene Klassen dafür schreiben oder ableiten, allerdings kann Ihnen der Objektinspektor in diesem Fall nicht behilflich sein.

1.5.5 Methoden für die Weckfunktion

Wir kommen nun wieder zu unserer Beispielanwendung zurück. Wir werden die Weckfunktion implementieren und dem Formular dabei zwei weitere (Nicht-Ereignisbearbeitungs-)Methoden spendieren.

Die Weckfunktion soll in zwei Stufen implementiert werden: In diesem Kapitel soll die Programmierung so einfach wie möglich sein; wir werden dann mit insgesamt nur acht selbst geschriebenen Programmzeilen eine Digitaluhr mit fast sekundengenauer Weckfunktion und benutzerdefinierbarer Schriftart erhalten. Kapitel 1.8.3 wird dann eine verbesserte Weckmethode vorstellen.

Eine unsichere Weckmethode

Die einfachste Art, die Weckzeit vom Benutzer eingeben zu lassen, ist über das Eingabefeld *AlarmText*, das wir schon in Kapitel 1.4.3 in das Formular aufgenommen haben. Der Text, den der Benutzer eingibt, ist jederzeit über das Property *AlarmText.Text* abfragbar (und auch änderbar), sogar schon dann, wenn der Benutzer die Eingabe noch nicht beendet hat.

Die einfachste und zugleich unsicherste Art, den Wecker auszulösen, ist, den Text der Zeitanzeige (*TimeText.Caption*) mit dem der Benutzereingabe (*AlarmEdit.Text*) zu ver-

gleichen und bei Übereinstimmung ein Meldungsfenster zu öffnen. Unsicherheitsfaktoren unserer ersten Weckfunktion werden sein:

- In Zeiten hoher Systemauslastung kann es dazu kommen, dass der Windows-Timer eine Sekunde überspringen muss, so dass einige Sekunden nie angezeigt werden und damit auch die Weckzeit übersehen werden kann, denn (und das ist ein weiterer Nachteil):
- Der Benutzer muss die Weckzeit sekundengenau angeben. Lässt er die Sekunden weg, führt der simple Stringvergleich nicht zum Erfolg.
- Ein dritter Nachteil ist: Der Vergleich findet auch dann jede Sekunde statt, wenn der Benutzer noch mit dem Schreiben beschäftigt ist.

Diese Nachteile werden in Kapitel 1.8.3 ausgeräumt werden.

Black-Box-Design

Überlegen wir nun, was die Uhr in Zukunft noch alles leisten soll: Sie soll eine Liste von Alarmzeiten verwalten und muss dann nicht nur die *primäre Alarmzeit* im Eingabefeld, sondern auch alle Listeneinträge mit der Uhrzeit vergleichen.

Es liegen somit zwei Gründe vor, den Vergleich einer Alarmzeit mit der aktuellen Uhrzeit in einer eigenen Methode unterzubringen:

- Der Vergleich soll als Black Box gesehen werden, so dass wir die oben genannten Nachteile ausräumen können, ohne die Ereignismethoden ändern zu müssen.
- Der Vergleich soll mehrfach im Programm ausgeführt werden.

Eine zweite Methode soll dazu dienen, den Benutzer zu »wecken« , also ein Meldungsfenster anzuzeigen oder einen Piepston auszustoßen. Die beiden Methoden erhalten die Namen *IsTimeOver* und *AlarmMessage*.

Das Black-Box-Konzept wird deutlich, wenn wir beide Methoden schon aufrufen, bevor wir sie implementiert haben. Die im letzten Kapitel geschriebene Timer-Methode für die Komponente *Timer* wird also wie folgt ergänzt:

```
procedure TUhrFormular.TimerTimer(Sender: TObject);
begin
  TimeText.Caption := TimeToStr(Time);
  if IsTimeOver(AlarmEdit.Text)
    then AlarmMessage('Alarm zur Zeit '+AlarmEdit.Text);
end;
```

So kann das Programm nicht mehr übersetzt werden, weil die beiden Methoden *IsTimeOver* und *AlarmMessage* noch nicht definiert sind. Um dies nachzuholen, fügen Sie den folgenden Quelltext zwischen, vor oder hinter die anderen Methoden im Implementationsteil der Unit ein:

```
function TUhrFormular.IsTimeOver(Alarm: string): Boolean;
begin
  Result := Alarm = TimeText.Caption;
{ entspricht den folgenden Zeilen:
  if (Alarm = TimeText.Caption) then Result := True
    else Result := False; }
end;

procedure TUhrFormular.AlarmMessage(Alarm: string);
begin
  Application.BringToFront;
  MessageBeep(MB_ICONEXCLAMATON); { um den PC-Lautsprecher
     zu benutzen, ersetzen Sie MB_ICONEXCLAMATION durch $FFFF}
  ShowMessage(Alarm);
end;
```

... und ergänzen Sie die Deklaration des Formulars im *private*-Bereich, wie im folgenden Ausschnitt deutlich gemacht:

```
type
  TUhrFormular = class(TForm)
    ...
  private
    { Private-Deklarationen }
    function IsTimeOver(Alarm: string): Boolean;
    procedure AlarmMessage(Alarm: string);
  public
    { Public-Deklarationen }
  end;
```

Ablaufdetails

Zum Ablauf des Programms einige besonders ausführliche Hinweise:

Die Methode *TimerTimer*, die ihren von Delphi erzeugten Namen der Tatsache verdankt, dass sie das Ereignis On*Timer* der Komponentenklasse T*Timer* bearbeitet, liest die vom Benutzer eingestellte Weckzeit aus dem Property *AlarmText.Text* aus und gibt sie als Parameter an die Funktion *IsTimeOver*. In dieser tritt die Weckzeit unter einem neuen Namen auf, der in der Parameterdeklaration festgelegt wird. Es handelt sich um den Namen *Alarm*. Der Stringvergleich zwischen *Alarm* und der aktuellen Zeit, die in *TimeText.Caption* angezeigt wird, ergibt einen Wert für wahr oder falsch (*True* oder *False*), den Sie direkt als Funktionsergebnis zurückgeben können. Dieses sprechen Sie in Object Pascal über den Bezeichner *Result* an. *IsTimeOver* gibt also ebenfalls *True* zurück, wenn Alarmzeit und aktuelle Zeit übereinstimmen. (Eine andere Möglichkeit zum Zeitvergleich wäre gewesen, den String *Alarm* mit dem Wert von *TimeToStr(Time)* zu vergleichen.)

So gelangt das Ergebnis des Vergleichs also zurück zur *if*-Abfrage in der Methode *TimerTimer*. Im Falle von *True* ruft *TimerTimer* die Methode *AlarmMessage* auf.

Die Anweisung *Application.BringToFront* in *AlarmMessage* sorgt dafür, dass das Beispielprogramm auch dann in den Vordergrund rückt, wenn es inzwischen von Fenstern anderer Anwendungen verdeckt worden ist. Nur dann erscheint nämlich das Meldungsfenster, das den Alarm anzeigen soll, ebenfalls im Vordergrund und kann vom Benutzer gesehen werden. Das Meldungsfenster wird mit Hilfe der von Delphi vordefinierten Funktion *ShowMessage* angezeigt (Unit *Dialogs*). Zusätzlich versucht die Funktion mit der Windows-Funktion *MessageBeep* ein akustisches Signal zu geben, zu dessen Wahrnehmung der Benutzer eventuell seine Multimedia-Lautsprecher aktiviert haben muss.

Die Uhr im Vordergrund

Normalerweise bieten Uhren und ähnliche Utilities auch die Möglichkeit an, dass sie immer im Vordergrund sichtbar bleiben, also auch, wenn Sie eine andere Anwendung in den Vollbildmodus schalten. Seit Windows 95 existiert mit der Task-Leiste bereits ein Fensterbereich, der standardmäßig immer sichtbar ist und der praktischerweise schon über eine Zeitanzeige verfügt. Daher ist es nicht mehr erforderlich, dass die hier entwickelte Uhr ebenfalls ständig im Vordergrund liegt.

Trotzdem ist es interessant zu wissen, wie einfach diese Option mit Delphi nachgerüstet werden könnte: Sie müssten lediglich das Formular-Property *FormStyle* auf den Wert *fsStayOnTop* einstellen (siehe hierzu auch Kapitel 3.4.2). Allerdings nimmt das Uhrformular in der jetzigen Größe wohl zu viel Platz weg, als dass es ständig im Vordergrund gelassen werden könnte.

Wegweiser

Hiermit ist die erste Stufe der Entwicklung unseres Beispielprogramms abgeschlossen. Sie finden den derzeitigen Entwicklungsstand auf der CD im Projekt `wecker1.dpr`. In Kapitel 1.8 werden wir einige Erweiterungen vornehmen, darunter auch die angekündigte Verbesserung der unrühmlichen Methode *IsTimeOver*.

1.5.6 Ereignisverknüpfung: Navigieren, verändern, lösen

Das einfache Anlegen einer Ereignisbearbeitungsmethode ist nicht die einzige Aufgabe, bei der der Objektinspektor Ihnen behilflich ist. Auch Änderungen der Verknüpfung, das Löschen von Ereignissen und das Navigieren zwischen Ereignissen werden von Delphi besonders unterstützt. Dazu kommt noch, dass jede Komponente über ein *Standardereignis* verfügt.

Das Standardereignis

Ein Standardereignis ist ein Ereignis, das besonders häufig bearbeitet wird. Zu diesem Ereignis können Sie in Delphi besonders schnell eine Methode erstellen – ohne Umweg über den Objektinspektor gelangen Sie mit einem Doppelklick auf die Komponente direkt vom Formular in den Quelltexteditor zu einer Methode für das Standardereignis. Einige Beispiele:

- Das Standardereignis des Formulars ist *OnCreate*, das bei der Öffnung des Fensters zur Laufzeit aufgerufen wird.
- Ein Doppelklick auf eine *Edit*-Komponente befördert Sie zu einer Methode für das Ereignis *OnChange*, das bei jeder Änderung des Benutzers im Editierfeld erzeugt wird.
- *OnClick* ist naheliegenderweise das Standardereignis der *Button*-Komponente und einiger anderer Komponenten.

Navigieren zwischen Ereignissen

Wenn bereits eine Methode für ein Ereignis existiert, bringt Sie ein Doppelklick in das entsprechende Methodenfeld des Objektinspektors schnell zur bestehenden Methode im Code-Editor. Das erspart Ihnen viel Sucharbeit im Quelltext, wenn Sie viele Komponenten mit vielen Ereignissen haben: Während die Methoden dort völlig unsortiert und ungegliedert quasi auf einer einzigen langen Seite aufgezählt sind, sind sie im Objektinspektor logisch und schnell auffindbar nach Komponenten geordnet.

Die Ereignisverknüpfung ändern und löschen

Sie können den Namen der Ereignismethode im Objektinspektor verändern. Wenn Sie einen noch nicht existierenden Namen eingeben, ändert Delphi den Namen der Methode an den automatisch erzeugten Stellen des Quelltextes (in der Deklaration des Formulars und in der Definition der Methode im Implementationsteil). Haben Sie den alten Namen an weiteren Stellen selbst verwendet, müssen Sie ihn dort selbst anpassen.

Statt eines neuen Namens können Sie auch eine schon existierende Methode angeben oder besser noch aus der aufklappbaren Liste des Objektinspektors auswählen. Der Vorteil dieser Liste ist auch, dass sie nur die Methoden anzeigt, die mit dem Ereignis kompatibel sind (die also die erforderlichen Parameter für dieses Ereignis aufweisen). Wenn Sie eine existierende Methode angeben, ändert Delphi die Verknüpfung, löscht aber die bisher verknüpfte Methode nicht.

Auf diese Weise ist es möglich, mehrere Ereignisse aus einer oder aus verschiedenen Komponenten mit einer einzigen Methode zu verknüpfen. Um dann innerhalb der

Methode zwischen den möglichen Komponenten zu unterscheiden, fragen Sie den Parameter *Sender* ab. Mehr Erläuterungen dazu finden Sie in Kapitel 1.8.6, weitere Beispiele für Methoden, die mehrere Ereignisse bearbeiten, finden Sie in Kapitel 3.5.2.

Es ist übrigens auch nachträglich (zur Programmlaufzeit) möglich, Methoden mit Ereignissen zu verknüpfen oder Verknüpfungen zu verändern, genauso, wie Sie nachträglich die anderen Properties im Programm ändern können, denn die Information, welche Methode zu einem bestimmten Ereignis gehört, *ist* ein Property.

Ebenso einfach, wie Sie die Verknüpfung ändern können, können Sie sie auch löschen. Entfernen Sie dazu einfach den Methodennamen aus dem Feld des Objektinspektors. Wie schon beim Ändern der Verknüpfung wird dadurch die bisher verknüpfte Methode nicht gelöscht.

Automatisches Entfernen leerer Ereignismethoden

Es gibt einen besonders schnellen Weg, Ereignisbearbeitungsmethoden vollständig zu löschen. Löschen Sie dazu einfach alle Zeilen zwischen dem *begin* und dem *end* der zu löschenden Methode, so dass nur noch der von Delphi automatisch erzeugte Methodenrumpf übrig bleibt, und speichern Sie die Datei. Beim Speichern entdeckt Delphi solche leeren Methoden und entfernt den leeren Rumpf, die Deklaration der Methode in der Formulardeklaration und alle Ereignisverknüpfungen im Objektinspektor, die auf diese Methode weisen. Selbstverständlich würde es Delphi niemals wagen, andere Methoden, die Sie nicht über den Objektinspektor verknüpft haben, zu entfernen, nur weil sie leer sind.

1.5.7 Drei Blicke hinter die Kulissen

Dieses Kapitel beschäftigt sich mit den Zusammenhängen zwischen visuellem Formular und Quelltext und klärt dabei die folgenden Fragen:

- Welche Änderungen nimmt Delphi automatisch an der Formular-Unit vor und welche Voraussetzungen müssen erfüllt sein, damit Delphi dazu in der Lage ist?
- Was passiert, wenn Sie Komponenten nachträglich löschen oder umbenennen?
- Wie behandelt Delphi die automatisch erzeugten Bereiche bei der Übersetzung und können manuelle Änderungen an diesen zu Fehlern führen?

Wenn ein Programmierwerkzeug die Quelltextdateien des Programmierers verändert, besteht immer dann die Gefahr eines Konflikts, wenn das Programmierwerkzeug bestimmte Bereiche für sich reserviert, die der Programmierer nicht verändern soll. Delphi verändert die zu den Formularen gehörenden Units in vorsichtiger Weise, und falls Sie automatisch erzeugte Zeilen ändern, können Sie Fehler hervorrufen, die aber schnell behoben werden können.

Änderungen mit den visuellen Tools

Das Einfügen neuer Komponenten läuft normalerweise problemlos ab. Wenn Sie Formular und Quelltexteditor nebeneinander sichtbar machen und dann eine Komponente ins Formular einfügen, sehen Sie sofort, welche Änderungen Delphi am Quelltext vornimmt (in diesem Fall fügt es ein Feld zur Formularklasse hinzu). Vor jeder dieser Änderungen durchsucht Delphi den Quelltext nach der Deklaration des Formulars. Solange Sie also die Formularklasse im Quelltext nicht entfernen oder umbenennen, dürften Sie von Delphi keine Beschwerden erhalten. Andererseits können Sie ausprobieren, den Namen des Formulars im Quelltext zu ändern und dann eine Komponente einzufügen. Delphi wird Ihnen mitteilen, dass die Deklaration der Formularklasse fehlt oder fehlerhaft ist, und es wird die Komponente nicht einfügen. Beruhigend bei diesem Test ist, dass Delphi es beim nächsten Einfügen einer Komponente wieder von neuem versucht und dass alles wieder funktioniert, wenn Sie die Formulardeklaration wieder in den Ursprungszustand zurückversetzt haben.

Es kommt wohl selten vor, dass man auf Anhieb allen Komponenten zufrieden stellende Namen gibt und genau die Komponenten und Ereignisse auswählt, die man braucht. Nun wäre es sehr unbequem, wenn diese Änderungen – nachträgliches Umbenennen und Löschen von Komponenten und Ereignissen – von Hand im Code-Editor vollzogen werden müssten. Sehen wir uns zuerst das Löschen an.

Löschen von Komponenten

Um herauszufinden, wie Sie eine Komponente aus einem Formular löschen können, brauchen Sie sich nur auf die üblichen Handgriffe zu besinnen: Markieren Sie die Komponente und drücken Sie die Taste Entf. Was geschieht nun, wenn Sie für diese Komponente bereits Ereignismethoden geschrieben haben? Selbstverständlich kann Delphi diese nicht einfach mitlöschen, denn Sie könnten viel Arbeit darin investiert haben und wollen Teile des Codes vielleicht an anderer Stelle wiederverwenden. Delphi führt daher nur die folgenden Aktionen aus:

Es löscht die Komponente aus dem Formular und aus der Formulardeklaration. Wenn Sie beispielsweise das Feld für die Zeitangabe löschen, entfernt Delphi die Zeile

```
TimeText: TLabel;
```

Damit sind auch alle Einstellungen, die Sie im Objektinspektor vorgenommen haben, gelöscht, also auch die *Verknüpfung* zwischen Ereignissen und Methoden, aber wie schon gesagt nicht die Methoden selbst. Sie können die Löschung schnell über BEARBEITEN | RÜCKGÄNGIG komplett rückgängig machen.

Umbenennen von Komponenten und Methoden

Zum Umbenennen von Ereignisbearbeitungsmethoden im Objektinspektor siehe Kapitel 1.5.6, *Die Ereignisverknüpfung ändern und löschen*. Was vielleicht häufiger vorkommt als das Umbenennen von Methoden, ist das Umbenennen von Komponenten, denn während man die von Delphi vorgegebenen Komponentennamen (*Button1*, *Edit1*,...) am Anfang vielleicht für ausreichend hält, hätte man nach einigen Erweiterungen doch gerne eine bessere Unterscheidungsmöglichkeit für *Button1*, *Button2*, *Button3* und *Button4*.

Wenn Sie nun den Namen einer Komponente im Objektinspektor verändern, bemüht sich Delphi nicht sonderlich, den Code der Unit entsprechend anzupassen. Falls Sie es allerdings beim vorgegebenen Namen für die mit den Komponentenereignissen verknüpften Methoden belassen haben (dieser Name enthält den Namen der Komponenten, z.B. *Button1Click*), so ändert Delphi den Namen im Objektinspektor und im Code entsprechend. Aus *Button1Click* könnte so z.B. der Name *SucheLaserDiscClick* werden. Wie schon beim eigenhändigen Ändern der Methodennamen (siehe Kapitel 1.5.6) müssen Sie die Vorkommnisse des Komponentennamens im Programmtext manuell ändern, da der Editor die Syntax zu wenig kennt, als dass er die Bezeichner selbst austauschen könnte, ohne gleiche Bezeichner aus verschiedenen Gültigkeitsbereichen zu verwechseln.

Wenn Sie den Namen eines Formulars ändern, passt Delphi auch den Namen des Formulars in den Kopfzeilen der selbst definierten Methoden an:

```
  TForm1.PrivateMethode;
{ wird zu }
  TArtikelAuswahlDialog.PrivateMethode;
```

Innerhalb dieser Methoden ersetzt Delphi die Vorkommnisse des Namens *TForm1* und *Form1* genauso wenig wie die Namen der Komponenten, wenn Sie diese ändern.

Änderungen im reservierten Bereich

Der reservierte Bereich der Formulardeklaration beginnt direkt nach der ersten Zeile, die das Schlüsselwort *class* enthält. Er beginnt mit der Deklaration aller Komponenten, die Sie zur Entwurfszeit in das Formular eingefügt haben, und endet mit der Deklaration der Ereignisbearbeitungsmethoden, die Delphi automatisch erzeugt hat. Der reservierte Bereich ist mit der ersten *private*-Überschrift in der Formulardeklaration beendet. Bei jeder Übersetzung überprüft Delphi diesen reservierten Bereich in besonderer Weise:

- Delphi testet, ob dieser reservierte Bereich Variablendeklarationen enthält, die zu keiner der Komponenten gehören. Falls das der Fall ist, fragt Delphi Sie, ob diese Deklaration entfernt werden soll. Falls Sie aus Versehen eine Formularvariable im

reservierten Bereich anstatt in den Bereichen *private* und *public* deklariert haben, wählen Sie *Nein* und verschieben Sie die Deklaration an die richtige Stelle. Ansonsten handelt es sich womöglich um eine Komponentendeklaration, zu der das Formular keine Komponente mehr enthält. Ein solcher Fall wäre dann vorstellbar, wenn Sie das Formular nicht visuell ändern, sondern die `dfm`-Datei in den Quelltexteditor laden und dort eine Komponente löschen.

- Wenn es umgekehrt im Formular eine Komponente gibt, die in der Formulardeklaration nicht aufgeführt ist, macht das meistens überhaupt nichts, solange Sie nicht versuchen, diese Komponente im Programm anzusprechen. Es ist auch möglich, eine einmal entfernte Komponentenvariable wieder in den reservierten Teil der Formulardeklaration einzufügen. Solange Sie dieser einen Namen geben, der mit dem *Name*-Property der Komponente übereinstimmt, funktioniert die Verbindung von Komponente und Variable zur Laufzeit problemlos. (Es ist jedoch wichtig, dass mindestens eine Variable von jeder verwendeten Komponentenklasse deklariert wird, da der Linker die Klasse sonst nicht in die EXE-DATEI bindet, was zur Laufzeit zu einem Fehler beim Laden des Formulars führt.)
- Bei den Methoden ist es umgekehrt wie bei den Komponenten: Überzählige Methoden werden von Delphi ignoriert, fehlende aber reklamiert. Fehlende Methoden sind die, die im Objektinspektor mit einem Ereignis verknüpft sind, die aber »heimlich« (d. h. nicht automatisch von Delphi) aus der Unit gelöscht worden sind.

Wie gesagt kann es zu solchen Differenzen nur kommen, wenn Sie per Hand die reservierten Bereiche ändern. Bei Änderungen im Formular passt Delphi die Unit automatisch an.

1.6 Bearbeiten von Projekten

Dieses Kapitel beschäftigt sich mit den Werkzeugen, die Sie bei der Arbeit mit Projekten beliebiger Größe unterstützen. Das bisherige Beispielprogramm liefert hierfür noch kein besonders gutes Beispiel, denn mit lediglich einem Formular ist es nur ein Mini-Projekt; daher müssen wir das Beispielprogramm an dieser Stelle erst einmal zurückstellen (bis Kapitel 1.8). In diesem Kapitel werden die folgenden Werkzeuge behandelt:

- der integrierte Quelltexteditor,
- die Projektverwaltung,
- die Objektablage,
- der Projekt-Browser (ab Delphi 5),

To-Do-Listen

Falls Sie die Professional- oder Enterprise-Version von Delphi benutzen (ab Delphi 5), sei bei dieser Gelegenheit noch auf die To-Do-Listen hingewiesen, die beim Management großer Projekte ebenfalls sehr hilfreich sein können. In ihnen können Sie Notizen über alle erdenklichen Arten von ausstehenden Arbeiten an Ihrem Projekt speichern. Das Praktische an den To-Do-Listen ist, dass ihre Einträge als normale Kommentare im Quelltext gespeichert werden und damit kompatibel mit externen Editoren sind. Ein Beispiel für einen To-Do-Eintrag im Quelltext:

```
{ TODO 2 -oEW -cPhantasieKategorie : Hier das muss noch geändert werden! }
```

In diesem Beispiel stellt »2« die Priorität dar, der Name hinter »-o« die zuständige Person und hinter »-c« folgt eine Kategorie, der der Eintrag zugeordnet ist.

Natürlich bietet die Delphi-IDE auch zu diesem Thema einen komfortablen Zugang: Neben der textuellen Eingabe haben Sie die Möglichkeit, die Einträge aus dem Editor heraus per Dialog hinzuzufügen (Kontextmenü des Editors oder Tastenkürzel [Shift]+[Strg]+[T]). Eine Übersicht über alle Einträge des Projekts erhalten Sie unter ANSICHT | TO-DO-LISTE, wo Sie auch projektweite To-Do-Einträge definieren können, die in einer eigenen `.todo`-Datei gespeichert werden. Die Handhabung von To-Do-Eingabe-Dialog und To-Do-Liste ist intuitiv und soll hier nicht näher erläutert werden. Erwähnenswert ist allerdings die Möglichkeit, eine To-Do-Liste als HTML-Tabelle zu exportieren – mit zahlreichen benutzerdefinierbaren Optionen z.B. für Farbe und Tabellenaufbau.

1.6.1 Der Quelltext-Editor

In der grundsätzlichen Bedienung des Editors hält sich Delphi weitgehend an den Windows-Standard. Dieser Abschnitt befasst sich nicht mit einfachen Editieroperationen, sondern konzentriert sich auf zwei Bereiche:

- die Programmierhilfen, die mit Delphi 3 eingeführt wurden
- die in Delphi 4 eingeführten AppBrowser-Funktionen wie CodeExplorer und Hyper-Links im Quelltext (erst ab der Professional-Version).

Genauere Information zur Tastenbelegung und zu Menüpunkten des Editors (Kontext-Menü und BEARBEITEN-Menü) gibt Ihnen die Online-Hilfe. Eine Übersicht über die Tastenbelegungen finden Sie beispielsweise, wenn Sie die Hilfe zur Dialogseite *Editor* unter TOOLS | EDITOROPTIONEN[1] aufrufen. Um keine versteckten Fähigkeiten des Editors zu übersehen, sollten Sie sich auf dieser Seite auch die einzelnen Optionen des Editors genauer ansehen.

1 Bis Delphi 4 ist diese Seite unter den UMGEBUNGSOPTIONEN angesiedelt.

Hinweis: Mit den in Anhang A beschriebenen und auf der CD enthaltenen Experten können Sie auch eine Standard- bzw. Personal-Ausgabe von Delphi mit einem CodeExplorer-Ersatz und einer Funktion ähnlich der Quelltext-Hyper-Links (allerdings nur innerhalb einer Datei) nachrüsten.

Die AppBrowser-IDE

Mit der Einführung von Delphi 4 Professional hat Borland mehrere damals neue Funktionen unter dem marketingtauglichen Begriff des *AppBrowsers* zusammengefasst. Die Vorteile dieser Funktionen beginnen schon bei ihrer leichten Erlernbarkeit, aufgrund derer sie hier nur kurz aufgezählt werden müssen:

- Beim *CodeExplorer* handelt es sich um das Baumansichtsfenster, das nach einer Neuinstallation von Delphi per Voreinstellung am linken Rand des Editorfensters angedockt wird. Dieses Fenster gibt Ihnen einen Überblick über die in der aktuellen Quelltextdatei enthaltenen Programmobjekte, aufgeteilt in die vier Kategorien Klassen, Variablen/Konstanten, Typen sowie eingebundene Units. Die Sortierung der einzelnen Listen und die Verwendung weiterer Unterkategorien (z.B. *private*, *protected*, *public*) können Sie in den Umgebungsoptionen auf der Seite EXPLORER einstellen. Mit einem Doppelklick auf ein Symbol springen Sie vom CodeExplorer direkt an die Deklaration des Symbols im Quelltext (Tastenkürzel: Shift+Strg+E). Im Kontextmenü finden Sie außerdem einen Menüpunkt, über den Sie den Namen des Symbols ändern können. Im Falle von Methoden ist dies eine nützliche Funktion, da Delphi den Namen automatisch in der Deklaration *und* in der Definition aktualisiert. Sie können im Code-Explorer sogar neue Symbole definieren (Popup-Menüpunkt NEU, ab Delphi 5), müssen sich aber an die korrekte Deklarationssyntax halten, sonst wird Ihre Eingabe einfach gelöscht (das abschließende Semikolon muss allerdings nicht angegeben werden).

- Klassenvervollständigung ist eine grafisch unauffällige Funktion, die Sie mit dem Tastenkürzel Shift+Strg+C aufrufen. Wenn sich die Eingabemarkierung zu diesem Zeitpunkt in der Deklaration einer Klasse befindet, fügt Delphi bei Bedarf leere Methodenrümpfe für alle von Ihnen deklarierten Methoden dieser Klasse in den Implementationsteil der Unit ein. Wenn Sie mehrere Klassen in einer Unit definieren, berücksichtigt Delphi sogar die Zwischenüberschriften wie etwa »{ TClass2 }« für alle Methoden der Klasse *TClass2* und ordnet die Methoden unter der passenden Zwischenüberschrift ein.

  Wenn Sie die Klassenvervollständigung von einem Methodenrumpf aus aufrufen, stellt sie wieder zuerst die Klasse der Methode fest, die sich an der Eingabeposition befindet, und arbeitet dann umgekehrt zum oben beschriebenen Fall: Sie deklariert alle noch nicht deklarierten Methoden dieser Klasse, die Sie schon in den Implementationsteil geschrieben haben. Delphi nimmt übrigens an, dass es sich um private Methoden handelt, und deklariert sie als *private*.

Ob auch unvollständige Property-Deklarationen vervollständigt werden sollen – mitsamt einer privaten Variablen zur Speicherung und einer Property-Schreibmethode zum Beschreiben des Wertes –, können Sie ebenfalls unter EXPLORER einstellen (UNVOLLST. EIGENSCHAFTEN VERVOLLST.).

- Auch zur Navigation zwischen der Deklaration einer Methode in der Klassendeklaration und dem Methodenrumpf gibt es nun Tastenkürzel: Shift+Strg+Pfeil ↓ zum Springen nach unten (zum Methodenrumpf) und Shift+Strg+Pfeil ↑ in umgekehrter Richtung.
- Bei gedrückter Steuerungstaste werden außerdem alle Bezeichner im Editor, über die Sie die Maus halten, zu einem hervorgehobenen Hyperlink. Wie in einem Web-Browser springen Sie durch einen Klick zur Definition des Symbols, sofern Delphi den entsprechenden Quelltext finden kann bzw. der Quelltext überhaupt vorliegt. In der rechten oberen Ecke des Editorfensters finden Sie sogar Schalter zum Hin- und Herblättern zwischen verschiedenen Sprungpositionen inklusive einer aufklappbaren History-Liste. Leider berücksichtigt diese Liste keine Sprünge, die auf eine andere Weise gemacht wurden.
- Sie können das Sprungziel des Hyperlinks auch in Erfahrung bringen, ohne den Link zu verfolgen. Setzen Sie dazu den Mauszeiger über einen Bezeichner, *ohne* Strg zu drücken. Delphi zeigt dann die Deklarationsposition des Symbols, über dem sich der Mauszeiger befindet, in einem Hinweisfenster an, z.B. »var Name – unit1.pas (48)« .

Hinweis: Der CodeExplorer markiert standardmäßig alle in der Klassendeklaration genannten Methoden, zu denen er noch keine Definition finden kann, durch fette Schrift. Dadurch sehen Sie, welche Methoden vervollständigt werden, wenn Sie die Klassenvervollständigungs-Funktion aufrufen. Die Fett-Markierung können Sie in den CodeExplorer-Einstellungen auch unterbinden.

CodeExplorer-Einstellungen

Die Einstellungen für den CodeExplorer finden Sie in den Umgebungsoptionen auf der Seite EXPLORER, Sie können sie aber auch direkt aus dem Kontextmenü des Explorers mit EIGENSCHAFTEN aufrufen (Abbildung 1.9). Einige auf dieser Seite zu findenden Optionen beziehen sich nur auf den CodeExplorer, andere nur auf den in Kapitel 1.6.5 beschriebenen Browser, wieder andere gelten für beide zugleich. Die Option UNVOLLST. EIGENSCHAFTEN VERVOLLST. bezieht sich wie erwähnt auf die Klassenvervollständigungs-Funktion.

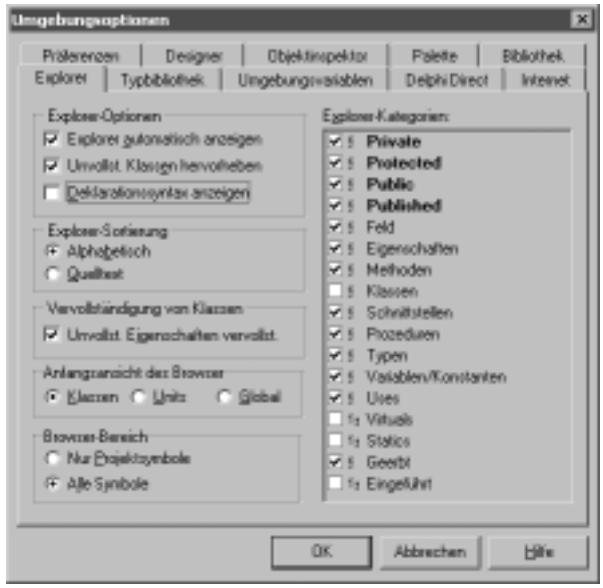


Abbildung 1.9: Die Optionen für CodeExplorer, Browser und Klassenvervollständigung

Interessant sind vor allem die folgenden Einstellungen, die auch für den Browser gelten:

- DEKLARATIONSSYNTAX ANZEIGEN ist per Voreinstellung abgeschaltet und bewirkt, dass Sie eine augenschonende, aber wenig informative Liste der Symbolnamen erhalten – weder erfahren Sie die Typen von Variablen noch die Parameter und Funktionsergebnisse von Methoden. All das ändert sich, wenn Sie diese Option einschalten.
- Die EXPLORER-SORTIERUNG kann entweder alphabetisch sein oder sich an der Reihenfolge der Symbole im Quelltext orientieren.
- In den EXPLORER-KATEGORIEN können Sie angeben, welche Arten von Symbolen der Explorer in einem »Ordner« (Baumknoten) zusammenfassen soll. Sind beispielsweise die Kategorien *private* und *protected* gewählt, *public* und *published* aber nicht, so werden alle als *public* und *published* deklarierten Symbole direkt unter dem Klassen-Knoten aufgelistet (sofern nicht andere gewählte Kategorien dagegensprechen). Um *private*- und *protected*-Symbole zu sehen, müssten Sie in diesem Beispiel erst die entsprechenden durch Ordner-Icons dargestellten und mit *private* und *public* beschrifteten Unterknoten öffnen.

  Wenn Sie eine Kategorie nicht in einem Knoten zusammenfassen, hat das zwar den Vorteil, dass Sie einen bestimmten Symbolnamen schneller finden können, aber Sie erhalten auch keinerlei Informationen mehr, welcher Kategorie das Symbol nun

angehört (in der Symbolanzeige von Delphi 4 kann man die Kategorien eines jeden Symbols noch an vorangestellten Icons ablesen).

Um sich einen Teil des Mausklickens zu ersparen, können Sie auch angeben, welche Kategorien standardmäßig expandiert werden sollen. Klicken Sie dazu in der Kategorie-Liste auf das unscheinbare graue Symbol neben dem Markierungsfeld.

Programmierhilfen

In Kapitel 1.5.4 sind uns schon zwei der »Programmierhilfen« über den Weg gelaufen, die in der amerikanischen Originalversion unter der Bezeichnung *Code Insight* geführt werden, welche treffend darauf hindeutet, dass der Editor eine besondere »Einsicht« in den Programmcode erreicht hat: Bei eingeschalteten Programmierhilfen arbeitet der Compiler im Hintergrund und aktualisiert laufend die internen Symbolinformationen. Dabei arbeiten die Programmierhilfen offenbar mit einer anderen Technik als der App-Browser, denn sie können im Gegensatz zu den AppBrowser-Funktionen nicht während des Debuggens genutzt werden.

»Code Insight« besteht aus den folgenden Funktionen, die Sie in den Umgebungsoptionen auf der Seite *Programmierhilfe* wiederfinden (siehe Abbildung 1.10):

- *Code-Vervollständigung:* zum Aufklappen einer Liste der Unterelemente eines Objekts, dessen Namen Sie gerade in den Editor geschrieben haben und hinter den Sie einen Punkt gesetzt haben. So erweitern Sie z.B. die Eingabe »Paintbox.« per Listenauswahl zu »Paintbox.Canvas«. Wenn Sie die Größe der Liste mit der Maus verändern, merkt sich Delphi diese Einstellung und stellt auch alle später aufgeklappten Listen auf die gleiche Größe ein (dies ist nur eine der vielen Detailverbesserungen, die die Code-Vervollständigung in Delphi 6 erfahren hat; für Weiteres sei auf das Kapitel *Neuerungen - Neue IDE-Funktionen* der Online-Hilfe verwiesen).

  Die Code-Vervollständigungs-Funktion kann auch an Positionen im Code verwendet werden, die nicht hinter einem Qualifizierungs-Punkt liegen, allerdings müssen Sie die Funktion dann manuell per Strg+Leertaste starten. Sie erhalten dann eine oft sehr unübersichtliche Liste aller Symbole, die an der aktuellen Eingabeposition verwendet werden können.

  Sollten Sie einmal vergeblich nach einem bestimmten Eintrag in der Vervollständigungs-Auswahl suchen, bedenken Sie, dass in seltenen Fällen nicht alle theoretisch möglichen Bezeichner aufgelistet werden (so zeigt z.B. Delphi 6 für *TShellListView* die Funktion *ShellFolder* nicht an, wenn die Code-Vervollständigung hinter »*Caption := ShellListView1.*« aufgerufen wird).

- *Code-Parameter:* zur Anzeige, welche Parameter zum Aufruf der Funktion erforderlich sind, deren Namen Sie gerade in den Editor geschrieben haben oder hinter deren Namen Sie die Eingabemarke gesetzt haben (siehe Beispiel in Kapitel 1.5.4).

- *Auswertung durch Kurzhinweis:* zur Anzeige des Wertes einer Variablen beim Debuggen, wenn Sie den Mauszeiger über die Nennung dieser Variablen im Editor halten. Ohne diese Hinweise müssten Sie, um einen Variablenwert einmalig abzufragen, ein Dialogfenster öffnen und dort den Namen der Variablen eingeben.

  Wichtig ist, dass diese Funktion keine *with*-Blöcke berücksichtigt, also nicht immer den Wert des Symbols anzeigt, das mit dem Wort unter dem Mauszeiger wirklich gemeint ist. Wenn Sie z.B. innerhalb einer Methode eines Formulars *Form1* geschrieben haben

  ```
  with Button1 do Width := Width+10;
  ```

  und den Mauszeiger beim Debuggen über *Width* halten, so zeigt Delphi Ihnen den Wert von *Form1.Width* an und nicht etwa den Wert von *Button1.Width*, der an dieser Stelle tatsächlich gemeint ist.

Hinweis: Kleine Experimente mit dem Debugger, wie gerade mit *with Button1 do Width:=Width+10* gezeigt, funktionieren oft nur, wenn Sie die Optimierung des Compilers abschalten, siehe hierzu auch Kapitel 1.7.1.

Abbildung 1.10 zeigt die entsprechenden Einstellungen im Optionsdialog (TOOLS | EDITOR-OPTIONEN). Unter Umständen kann es aber auch bequemer sein, die *Code-Vervollständigung* abzuschalten und sie manuell per Strg+Leertaste aufzurufen, nachdem Sie einen Qualifizierungspunkt eingegeben haben (z.B. »Button1.«).

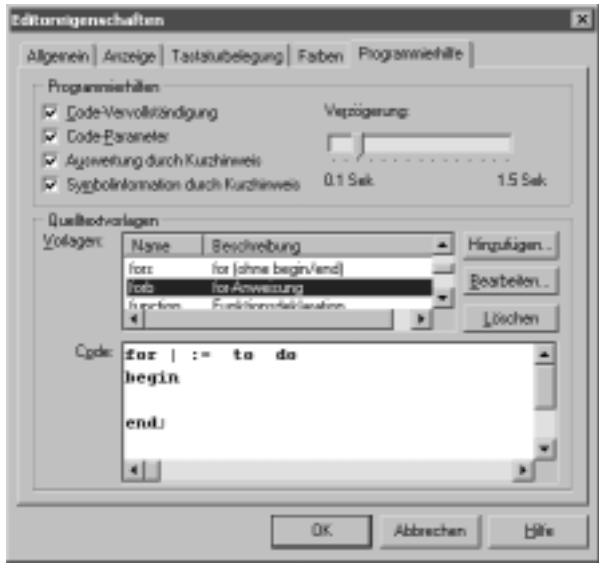


Abbildung 1.10: Die Einstellungen zu den Programmierhilfen des Editors

Der größte Teil der in Abbildung 1.10 gezeigten Dialogbox soll hier nur der Vollständigkeit halber erwähnt werden, denn er hat nichts mit den Symbolinformationen des Compilers zu tun, sondern ist eine völlig unspektakuläre Funktion, die es schon seit Jahren in allen Arten von Editiersoftware gibt: Das Einfügen von häufig benötigten Textbausteinen per Tastendruck. In Delphi läuft das folgendermaßen ab: Sie geben beispielsweise ein »if« ein, drücken das Tastenkürzel für die Quelltextschablonen (Strg+J) und der Editor fügt dann im günstigsten Fall bereits die restlichen zum bisher eingegebenen Konstrukt gehörenden Wörter ein. Im Falle von »if« gibt es in Object Pascal jedoch mehrere Auswahlmöglichkeiten. Von diesen müssen Sie dann aus einer Liste eine wählen.

Jede Quelltextschablone hat auch ein Kürzel, mit dem Sie sich die Auswahl aus einer Liste ersparen (geben Sie z.B. statt *if* das Kürzel *ifeb* ein und drücken dann Strg+J, so erhalten Sie sofort die ausführlichste Version einer *if*-Anweisung in Object Pascal). Schließlich gehört zu jeder Schablone auch noch ein festgelegter Punkt innerhalb der Schablone, auf den der Cursor nach dem Einfügen gesetzt wird, also die erste Lücke, die Sie noch per Hand ergänzen müssen. All diese Optionen stehen Ihnen zur Verfügung, wenn Sie in der Dialogbox aus Abbildung 1.10 eigene neue Schablonen erzeugen. Die genannten Schablonen mit *if* im Namen sind bereits von Borland vordefiniert, wie auch weitere Schablonen für andere Object-Pascal-Konstrukte. Wenn Sie ein Backup Ihrer eigenen Schablonen machen wollen, kopieren Sie die Datei *delphi32.dci* aus `Delphi-Verzeichnis\bin`.

Wichtige Tastenkürzel

Die folgende Tabelle listet die bis hier erwähnten wichtigen Tastenkürzel auf, denn einige davon sind nicht aus dem Kontextmenü des Editors ersichtlich und können leicht in Vergessenheit geraten:

Tastenkürzel	Wirkung
Shift+Strg+C	Vervollständigen der Klasse an der Eingabeposition (kann in Interface und in Implementation aufgerufen werden)
Shift+Strg+Pfeil ↑	Springen zur Deklaration der gerade bearbeiteten Methode
Shift+Strg+Pfeil ↓	Springen zum Rumpf der Methode, deren Deklaration sich an der Eingabeposition befindet
Strg+Leer	Expliziter Aufruf der Programmierhilfen
Strg	Umschalten in den Hyperlink-Modus – Mausklicks auf Symbolnamen im Quelltext bewirken Sprung zur Deklaration des Symbols
Strg+J	Einfügen einer Code-Schablone
Shift+Strg+E	Umschalten zwischen Code-Editor und Code-Explorer.

1.6.2 Aufbau von Projekten

Bei einfachen Projekten mit nur einem Formular brauchen Sie sich noch keine Gedanken um die Verwaltung des Projekts zu machen. Zur Speicherung und zur Benennung von Dateien genügen hier noch die Erklärungen von Kapitel 1.4.4. Sobald weitere Formulare hinzukommen, werden die Eigenschaften der Projektverwaltung interessant. Wir untersuchen zuerst den Aufbau eines Projekts.

Die Dateien eines Projekts

Zur Erstellung einer Anwendung in Delphi sind mehrere Dateien notwendig, die zu einem Projekt zusammengefasst werden. Dieses können Sie dann als Ganzes über das DATEI-Menü laden und speichern. Grundlage für die Aufteilung in Dateien sind die Formulare. Delphi erstellt für jedes Formular, das Sie erzeugen, eine eigene Object-Pascal-Datei. Diese Datei wird auch als *Modul* bezeichnet, welches in Object Pascal den speziellen Namen *Unit* hat. Sie können auch eigene Units erstellen, die nichts mit Formularen zu tun haben, sondern nur dazu dienen, Ihr Programm in überschaubare Einheiten aufzuteilen.

Jedes Delphi-Projekt besitzt, wie herkömmliche Pascal-Projekte, nicht nur eine Vielzahl gleichberechtigter Units, sondern darüber hinaus ein Hauptmodul, das nach dem Programmstart als Erstes die Kontrolle erhält (das Hauptmodul wird nicht als Unit bezeichnet).

In verschiedenen Computersprachen gibt es Projektdateien, die dem Entwickler und dem Entwicklungssystem einen Überblick über alle Module des Projekts geben. In Delphi ist es nicht notwendig, dafür eine eigene Datei anzulegen: Das Hauptmodul *ist* die Projektdatei und trägt daher die Endung `.dpr` (für **D**elphi**PR**oject). Sie wird von Delphi automatisch verwaltet, kann aber auch vom Programmierer verändert werden. Einsicht in die Projektdatei erhalten Sie nach Auswahl des Menüpunktes PROJEKT | QUELLTEXT ANZEIGEN.

Alle Units tragen die Endung `.pas`. Darüber hinaus muss Delphi den Aufbau der Formulare speichern und verwendet dazu binäre Dateien, die seit dem Erscheinen von Kylix mit zwei verschiedenen Endungen auftreten: Die Endung `.dfm` steht für DelphiForM und wurde bis Delphi 5 ausschließlich verwendet. Die Endung `.xfm` wird standardmäßig von Kylix verwendet, wobei das »X« für »Cross Platform« steht. (Sie können auch in Kylix die Endung `.dfm` verwenden, wobei Kylix unabhängig von der Endung immer denselben Inhalt in die Datei schreibt.) In Delphi 6 ist die Dateiendung wichtig, um zwischen einer VCL- und einer CLX-Anwendung zu unterscheiden: `.xfm` steht für eine Datei, die die CLX als Grundlage hat, `.dfm` für ein Formular, das aus VCL-Komponenten besteht (was im Folgenden als Normalfall vorausgesetzt wird).

Für jedes Projekt gibt es also eine einzige .dpr-Datei und für jedes Formular ein Dateipaar, bestehend aus einer .dfm-Datei und einer .pas-Unit. Dies sind die Dateien, die standardmäßig im Fenster der Projektverwaltung (Abbildung 1.11) angezeigt werden. Alle anderen Dateien wie Units und Bitmap-Ressourcen werden durch Verweise in der *uses*-Anweisung der Units oder durch einen *$R*-Compilerbefehl in das Projekt eingebunden.

Vorübersetzte Units

Damit Delphi nicht jedes Mal alle Units neu übersetzen muss, legt es für jede Unit eine Datei mit der Endung .dcu an, die die zur Unit gehörige Übersetzung enthält und sehr schnell mit anderen dcu-Dateien zur EXE-Datei zusammengebunden werden kann. dcu steht für *Delphi Compiled Unit*. Die dcu-Dateien werden im selben Verzeichnis gespeichert wie die EXE-Dateien, und zwar in dem Verzeichnis, das Sie in den Projektoptionen auf der Verzeichnisseite unter AUSGABEVERZEICHNIS angeben.

Verwandt mit den dcu-Dateien sind übrigens die dpu-Dateien (Delphi Package Unit), die der Compiler beim Kompilieren eines Packages erzeugt.

Nicht-visuelle Units in der Projektverwaltung

Damit die von Ihnen verwendeten Nicht-Formular-Units ebenfalls in der Projektverwaltung aufgelistet werden, können Sie sie explizit darin aufnehmen. Ein entsprechender Schalter befindet sich in der Standard-Symbolleiste; aber auch im Menü PROJEKT und im lokalen Menü der Projektverwaltung sind entsprechende Punkte zu finden. Nicht-visuelle Units, die Sie mit DATEI | NEU erzeugen, werden übrigens automatisch in das aktive Projekt eingefügt.

Obwohl Sie auch diese Units im Projektfenster aufnehmen können, ist es nicht der Sinn der Projektverwaltung, den absoluten Überblick über das Projekt zu geben. Für eine vollständige Übersicht über wirklich alle im Programm benutzten Units können Sie den in die IDE integrierten Browser verwenden (siehe Kapitel 1.6.5).

1.6.3 Die Projektverwaltung

Ein Projekt in Delphi ergibt nach der Übersetzung durch den Compiler immer eine einzelne ausführbare Datei, etwa eine EXE-Datei oder eine DLL. Da viele Projekte aber aus mehreren Dateien bestehen, können Sie seit Delphi 4 mehrere Projekte zu einer Projekt*gruppe* zusammenfassen, die dann im Fenster *Projektverwaltung* in einer hierarchischen Anzeige (Abbildung 1.11) dargestellt wird. Auf der Buch-CD finden Sie als Beispiel dafür die fünf Versionen des Beispielprogramms aus Kapitel 1.9 in der Projektgruppe Kapitel1.9.bpg (ein Projekt mit fünf EXE-Dateien als Zieldateien) und einige Projektgruppen bestehend aus EXE/EXE- und EXE/DLL-Paaren in den Ordnern Kapitel6 und Kapitel8.

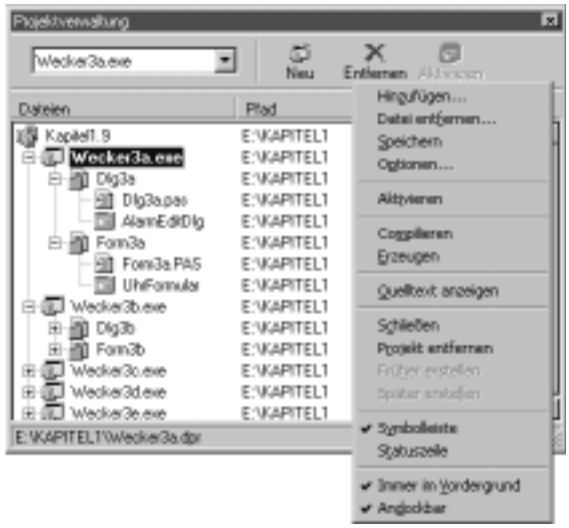


Abbildung 1.11: Die Projektverwaltung

Projektdateien

Um den Projektmanager zum Vorschein zu bringen, verwenden Sie den Menüpunkt ANSICHT | PROJEKTVERWALTUNG. Bei einem einfachen Projekt finden Sie dort unter einem Wurzelknoten, der per Voreinstellung den Namen *ProjektGroup1* trägt, die Zieldatei des Projekts, üblicherweise eine EXE-Datei. Diesem Eintrag untergeordnet sind alle Module, die an diesem Projekt beteiligt sind. Für ein Modul, das ein Formular enthält, finden Sie wiederum zwei Untereinträge, einen für die Formulardatei und einen für den Object-Pascal-Quelltext.

Jeder Knoten für eine Zieldatei entspricht einem einzelnen Projekt. Zu jedem Projekt gibt es genau einen Projektquelltext, der die Endung `.dpr` trägt und den Sie über den Menüpunkt PROJEKT | QUELLTEXT ANZEIGEN einsehen können. Ein typischer Projektquelltext ist der des bisherigen Beispielprogramms:

```
program Wecker1;

uses
  Forms,
  UhrForm1 in 'UhrForm1.PAS' {UhrFormular};

{$R *.RES}

begin
  Application.Initialize; // (fehlt in 16-Bit-Delphi)
  Application.CreateForm(TUhrFormular, UhrFormular);
  Application.Run;
end.
```

Die Optionen, die in der Projektdatei festgelegt werden, sind:

- Mindestens jede Formular-Unit wird per *uses*-Klausel in die DPR-Datei eingebunden und findet sich auch in der Liste des Projektfensters wieder.
- Neben einer solchen Unit befindet sich der Name des Formulars, das in dieser Unit enthalten ist, und zwar in Kommentarklammern. Diese Kommentare unterscheiden sich von normalen Kommentaren dadurch, dass die Projektverwaltung sie *auswertet*. Nur über diese Kommentare kann die Projektverwaltung erkennen, dass zu einer Unit auch ein Formular gehört.
- Für jedes Formular, das automatisch erstellt werden soll, ist im Hauptprogramm ein Aufruf von *Application.CreateForm* zu finden. Im Dialog PROJEKT | OPTIONEN können Sie dies auf der Seite *Formulare* verändern, indem Sie die Formulare zwischen den Listen *Autom. Formularerstellung* und *Verfügbare Formulare* austauschen. Standardmäßig wird jedes neue Formular in die Liste der automatisch zu erzeugenden Formulare aufgenommen. Sie können dieses Standardverhalten ab Delphi 5 jedoch auch umkehren, indem Sie die Option FORMULARE AUTOM. ERZEUGEN in den Umgebungsoptionen, Seite PRÄFERENZEN, deaktivieren.

  Für Formulare, die nicht automatisch erstellt werden, müssen Sie den Konstruktor des Formulars selbst aufrufen, bevor Sie ein Exemplar davon benutzen können. Detaillierte Informationen dazu finden Sie in Kapitel 3.4.4 (*Arbeiten mit mehreren Formularen*), ein häufiges Beispiel dafür sind dynamisch erzeugte MDI-Kindfenster (siehe Kapitel 5.7.4).

Hinweis: Damit manuelle Änderungen im Quelltext von Projektdateien in den Projektmanager übernommen werden, müssen Sie das Projekt neu laden.

Befehle für das Projektmanagement

Die mit der Projektverwaltung zusammenhängenden Befehle sind an verschiedenen Orten anzutreffen: im Hauptmenü (DATEI, ANSICHT und PROJEKT), in der Symbolleiste unter dem Hauptmenü, in der Symbolleiste des Projektfensters und im lokalen Menü desselben.

Der Inhalt des lokalen Projektmanager-Menüs hängt davon ab, welche Art von Knoten Sie anklicken. Bei einer Zieldatei (einem »Einzelprojekt«) finden Sie darin z.B. Befehle zum Hinzufügen oder zum Entfernen von Dateien, zum Speichern aller aufgelisteten Dateien und zum Editieren der Projektoptionen, wie etwa zum Festlegen des Icons für die EXE-Datei.

Projekte erzeugen

Das vollständige Neu-Kompilieren des Projekts inklusive all seiner Units, für das es im Englischen den schönen Fachbegriff *Build* gibt, wird in der deutschsprachigen Version von Delphi als ERZEUGEN bezeichnet, was Sie nicht dazu verleiten sollte, dies mit dem nicht eingedeutschten Menüpunkt für das COMPILIEREN zu verwechseln. Im normalen Entwicklungsalltag wird das Erzeugen eines Projekts meistens dann fällig, wenn Sie etwa eine globale Compiler-Option geändert haben und nun den gesamten Quelltext mit der neuen Option übersetzen wollen. Bei solchen Optionsänderungen führt Delphi nämlich vor dem nächsten Programmstart nicht von sich aus eine automatische Neu-Kompilierung durch.

Es empfiehlt sich, ein Projekt außerdem neu zu erzeugen, bevor Sie die kompilierte Datei weitergeben oder wenn es bei der Programmausführung zu unerklärlichen Fehlern kommen sollte. Beim normalen Kompilieren übersetzt Delphi nämlich nur die Quelltexte, die seit der letzten Übersetzung aktualisiert wurden, sowie die Dateien, die von diesen abhängig sind. Fehler bei solchen Abhängigkeitsprüfungen (die von der aktuellen Delphi-Version zwar noch nicht berichtet wurden, aber nie ganz auszuschließen sind) könnten dazu führen, dass eine veraltete Unit in die neue ausführbare Datei eingebunden wird. Und selbst wenn diese veraltete Unit an sich keine Fehler enthält, können zur Laufzeit die erwähnten rätselhaften Fehler entstehen, wenn die anderen Units mit einer neuen Version der veralteten Unit kompiliert wurden und daher z.B. ein falsches Aufrufformat für Methoden verwenden (dieser Fehler kann dann natürlich auch bei Aufrufen von neuen Units aus der veralteten Unit vorkommen).

Automatisch angezeigte Formulare und das Hauptformular

Zwar werden alle automatisch erzeugten Formulare Ihrer Anwendung beim Programmstart erzeugt, allerdings ist es sinnvoll, dass davon zunächst nur eines sichtbar wird. Sichtbar sind zunächst nur die Formulare, deren *Visible*-Property den Wert *True* aufweist. Formulare, die am Anfang nicht sichtbar sind, können Sie zur Laufzeit des Programms bei geeigneter Gelegenheit von der Unsichtbarkeit befreien, indem Sie *Visible* auf *True* setzen.

Schließlich ist es noch von Bedeutung, welches Formular das *Hauptformular* einer Anwendung ist. Wenn Sie nämlich dieses Formular zur Laufzeit schließen, wird die gesamte Anwendung geschlossen. Andere Formulare können Sie jederzeit schließen, ohne dadurch die Anwendung zu beenden.

Standardmäßig ist das erste Formular des Projekts das Hauptformular. Um das später zu ändern, wählen Sie ein anderes Formular aus der Liste *Hauptformular* in den Projektoptionen (Seite *Formulare*) aus. Hinter dieser Dialog-Kulisse verbirgt sich der folgende Mechanismus, den Sie im Projekt-Quelltext sehen können, sobald Sie die Dialogbox wieder schließen:

Die Methode *Application.CreateForm* erkennt es, wenn zum ersten Mal ein Formular erzeugt wird. Dieses erste Formular sieht sie unbeirrbar als Hauptformular an. Später können Sie dieses über das Property *Application.MainForm* abfragen, aber nicht mehr ändern. Um das Hauptformular zum Hauptformular zu machen, muss dieses also im ersten Aufruf von *CreateForm* der Projektdatei erzeugt werden. Genau das ist es, was Sie auch mit der Auswahl eines neuen Hauptformulars in der Dialogbox *Projektoptionen* erreichen.

Projektgruppen

Wenn Sie wissen, was ein Projekt ist, können Sie sich leicht in die Projektgruppen einarbeiten. In einer Projektgruppe werden mehrere Einzelprojekte zusammengefasst, das Hinzufügen von Projekten geht denkbar einfach über die entsprechenden Menüpunkte im Hauptmenü unter PROJEKT und im lokalen Menü der Projektgruppe im Projektfenster. Die Reihenfolge der Projekte lässt sich über FRÜHER ERSTELLEN und SPÄTER ERSTELLEN nachträglich ändern.

Innerhalb einer Gruppe können Sie jedes Projekt weiter wie ein einzelnes Projekt behandeln, jedes Projekt verfügt also über einen eigenen Satz an Optionen, die Sie im bekannten Optionsdialog einstellen können. Diesen Dialog können Sie aus dem lokalen Menü des Projekts im Projektmanager aufrufen oder aus dem Hauptmenü, falls das Projekt das *aktive* Projekt ist.

Das Konzept des *aktiven Projekts* ist auch für andere Aufgaben wie etwa das Kompilieren und Starten einer Anwendung von Bedeutung: Aus allen Projekten einer Gruppe kann immer nur ein Projekt aktiv sein. Welches das gerade ist, können Sie aus dem Namen in der Delphi-Titelleiste, aus der Hervorhebung im Projektfenster und ab Delphi 5 auch aus der Auswahlliste am oberen Ende des Projektfensters entnehmen. Über diese Auswahlliste, über das lokale Menü des Projektmanagers und über dessen Symbolleiste können Sie ein Projekt manuell aktivieren – eine automatische Aktivierung findet statt, wenn Sie aus dem Projektmanager heraus zu einer Datei des Projekts wechseln.

Die meisten Befehle im Menü PROJEKT beziehen sich auf dieses aktive Projekt, ebenso wie der Menüpunkt START | START. Im Projektmenü können Sie schon durch die Namen der Menüpunkte erkennen, ob sie sich auf die ganze Gruppe oder nur auf das aktive Projekt beziehen. So gibt es sowohl den Punkt *[Name des Projekts]* ERZEUGEN als auch ALLE PROJEKTE ERZEUGEN.

Mehrprozess-Debuggen

Das Starten von Projekten aus einer Projektgruppe führt zur interessanten Frage, was passiert, wenn Sie schon ein Projekt gestartet haben und nun ein anderes aktivieren. Kann dieses dann auch gestartet werden? Die Antwort lautet, dass Sie auf diese Weise

mehrere Projekte gleichzeitig starten und unter Windows NT sogar gleichzeitig debuggen können. Hier kommt das Modulfenster des Debuggers ins Spiel, in dem alle zurzeit aktiven Prozesse aufgelistet werden (zur Laufzeit wird jede ausfürbare Datei zu einem *Prozess*). Eine unvermeidliche technische Beschränkung liegt allerdings darin, dass immer nur ein Prozess vom Debugger angehalten werden kann. Welcher Prozess gerade angehalten ist, erkennen Sie im Modulfenster am grünen Pfeil. Eine weitere Beschränkung liegt darin, dass Delphi keine Programme kompilieren kann, während ein anderes gerade von der IDE aus gestartet wurde. Bevor Sie das erste von mehreren Projekten starten, sollten Sie daher die Aktion PROJEKT | ALLE PROJEKTE ERZEUGEN ausführen.

Einzelprojekte und »virtuelle Gruppen«

Schon die Tatsache, dass Sie im DATEI | NEU...-Dialog keinen Eintrag für »Neues Projekt«, sondern nur noch für »Neue Projektgruppe« finden, weist darauf hin, dass Sie eigentlich gar nicht ohne Projektgruppen arbeiten können. Auch wenn Sie eine einzelne DPR-Datei öffnen, wird dieses Projekt im Projektmanager immer innerhalb einer neuen Gruppe dargestellt. Solange Sie diese Gruppe nicht speichern, bleibt sie aber »virtuell«, sie existiert also nicht als physikalische Datei auf der Festplatte. *Wenn* Sie eine Gruppe speichern (und das wird spätestens dann erforderlich, wenn Sie mehr als ein Projekt darin aufgenommen haben), erzeugt Delphi dafür eine Datei mit der Endung `.bpg` (Borland Project Group) – eine lesbare Datei, die nach einer einfachen Syntax aufgebaut ist, welche mit Object Pascal keinerlei Ähnlichkeiten aufweist und für die eine genauere Beschreibung in diesem Buch sicher uninteressant wäre. Mit QUELLTEXT DER PROJEKTGRUPPE ANZEIGEN aus dem lokalen Menü des Projektmanagers (nur bei Markierung des Wurzelknotens) können Sie diese Datei im Editor einsehen, unabhängig davon, ob sie auf der Festplatte existiert oder nicht.

1.6.4 Objektablage und Komponentenschablonen

In fast jeder neuen Version lernt Delphi neue Wege, das Prinzip der Wiederverwendung aus der objektorientierten Programmierung auch auf die Formulargestaltung zu übertragen. In Delphi 6 stehen Ihnen die folgenden Möglichkeiten zur Wiederverwendung von visuell gestalteten »Bauelementen« zur Verfügung:

- Wiederverwendung eines einmal angelegten Formulars/Projekts, das Sie in der Objektablage untergebracht haben (seit Delphi 1, wo der Ablageort noch als Galerie bezeichnet wurde)
- Erweiterung eines einmal angelegten Formulars (hier *erbt* ein neues Formular den Aufbau und die Ereignisbearbeitung von einem in der Objektablage gespeicherten Formular; seit Delphi 2)

- Wiederverwendung von Komponenten oder Komponentengruppen (seit Delphi 3) als Komponentenschablone (Komponentenvorlage).
- Zusammenfassung von Komponentengruppen in einem *Frame*, der so leicht wiederverwendet werden kann wie eine Komponentenschablone, aber gegenüber dieser erhebliche Vorteile aufweist (seit Delphi 5).

Die folgenden Abschnitte gehen auf den Einsatz der ersten drei Techniken ein. Eine Diskussion der Vor- und Nachteile der einzelnen Techniken sowie Beispiele für die Anwendung von Formularvererbung und Frames finden Sie in Kapitel 3.7.

Zugriff auf die Objektablage

Der übliche Zugriff auf die Objektablage läuft über das Menü DATEI | NEU | WEITERE ab bzw. weniger umständlich über den ersten Schalter unter dem Dateimenü (entspricht bis Delphi 5 dem Menüpunkt DATEI | NEU). Sie gelangen dadurch in die in Abbildung 1.12 gezeigte Dialogbox, wählen eines der auf den Seiten enthaltenen Objekte aus, stellen die Art der Nutzung ein (KOPIEREN, VERERBEN oder VERWENDEN, dazu später mehr), wählen *OK* und erhalten ein neues Objekt, das von nun an zu Ihrem Projekt zählt.

Sie können die IDE jedoch auch so einstellen, dass sie bei Auswahl der Menüpunkte DATEI | NEU | ANWENDUNG und DATEI | NEU | FORMULAR kein leeres Projekt bzw. Formular erstellt, sondern eine Kopie eines Projekts/Formulars aus der Objektablage. Hierzu markieren Sie das Projekt/Formular in den Optionen zur Projektablage (TOOLS | OBJEKTABLAGE...) als NEUES FORMULAR bzw. NEUES PROJEKT.

Eine dritte Option an dieser Stelle ist HAUPTFORMULAR. Ein damit markiertes Formular der Objektablage wird beim Öffnen eines neuen Projekts automatisch als Hauptformular verwendet.

Die drei Arten, ein abgelegtes Objekt zu verwenden

Die Objektablage bietet Ihnen grundsätzlich die drei oben erwähnten Möglichkeiten, die in ihr aufbewahrten Objekte in neuen Projekten wiederzuverwenden, auch wenn nicht für jedes Objekt alle drei Optionen zur Verfügung stehen:

- Wenn Sie KOPIEREN wählen, lädt Delphi alle Dateien, die zum ausgewählten Objekt gehören, und legt sie als unbenannte Dateien im aktuellen Projekt ab. Handelt es sich beispielsweise um ein Formular, so lädt Delphi die Formulardatei und die zugehörige Unit. Beim nächsten Speichern werden Sie dann automatisch nach einem Namen für die noch unbenannten Dateien gefragt. Delphi speichert die Dateien unter dem neuen Namen ab. Diese neuen Dateien haben nichts mehr mit den Objekten der Objektablage zu tun.

Abbildung 1.12: Abbildung 1.12 Die Objektablage befindet sich im Neu-Dialog; das vordere Fenster zeigt die Dialogschablonen in der Symbolansicht.

- Die beim Erscheinen von Delphi 2 zu Recht viel gepriesene »visuelle Formular-Vererbung« erhalten Sie mit der Option VERERBEN. Wie beim Kopieren des Objekts legt Delphi beim Vererben eine oder mehrere neue unbenannte Dateien an, jedoch werden diese nicht aus den Dateien der Objektablage erzeugt.

  Am Beispiel eines Formulars bedeutet dies: Die neu erzeugte Formular-Unit enthält eine neue Formularklasse, die keinerlei Komponentendeklarationen und keine Ereignisbearbeitungsmethoden enthält, so als handele es sich um ein völlig neues Formular. Einziger Unterschied zu einem solchen völlig neuen Formular ist, dass die Formularklasse nicht von *TForm*, sondern von der Klasse des Formulars der Objektablage abgeleitet ist. Die Unit enthält also statt »*TForm1=class(TForm)*« eine Zeile der Art »*TForm1=class(TFormXYZAusAblage)*« .

  Für Delphi ist es mit diesem kleinen Unterschied natürlich nicht getan; es muss im Hintergrund einigen Verwaltungsaufwand betreiben, um schon zur Entwurfszeit ein Formular darzustellen, das sowohl die Komponenten des Ablageobjekts als auch die Komponenten, die Sie nachträglich hinzufügen, anzeigt. (Dieser hohe Verwaltungsaufwand, der durch einen langsameren Bildaufbau zur Entwurfszeit auffallen kann, ist zur Laufzeit nicht erforderlich. Die Vererbung beeinträchtigt also die Geschwindigkeit Ihrer Anwendung nicht.)

  Jede Ihrer Änderungen des per Vererbung neu erzeugten Formulars wirkt sich nur auf dieses Formular aus. Umgekehrt wirken sich jedoch alle Änderungen, die Sie an einem Formular der Objektablage vornehmen – sowohl Änderungen der Kom-

ponenten und Properties als auch Änderungen der Ereignisbearbeitungsmethoden – auf alle Formulare aus, die Sie über die Option *Vererben* von diesem Formular erzeugt haben.

Außer Formularen (dazu gehören auch die Formulare auf der Objektablage-Seite *Dialoge*) können Sie noch Datenmodule vererben, bei Projekten scheidet diese Möglichkeit aus, da ein Projekt nicht als Klasse vorliegt.

- Schließlich können Sie manche Objekte der Ablage auch noch VERWENDEN. Diese Option bewirkt, dass Delphi keine neue unbenannte Datei anlegt, sondern die Originaldateien aus der Objektablage direkt in Ihr Projekt einbindet. Wenn Sie also Änderungen durchführen, verändern Sie das Objekt, das sich in der Objektablage befindet. Eine solche Veränderung ist besonders dann erwünscht, wenn es sich um eine Formularklasse handelt, die an andere Formulare vererbt wird, und wenn alle diese Formulare gleichzeitig verändert werden sollen. Ändern Sie dann das Formular aus der Objektablage, so wirkt sich das auch auf diese Formulare aus. (Allerdings müssen Sie ein Ablageobjekt nicht über VERWENDEN in das Projekt einbinden, um es direkt zu verändern. Sie können es auch mit DATEI | ÖFFNEN direkt aus der Ablage laden – Delphi speichert die Ablageobjekte im Verzeichnis `Delphi-Verzeichnis\ObjRepos`, für die Aufteilung auf die verschiedenen Seiten der Ablage ist die Datei `Delphi-Verzeichnis\delphi32.dro` zuständig.)

Die Optionen VERERBEN und VERWENDEN stehen übrigens nur zur Verfügung, wenn Sie den Menüpunkt DATEI | NEU verwenden, nicht aber bei der automatischen Auswahl eines Ablageobjekts durch DATEI | NEUE ANWENDUNG und DATEI | NEUES FORMULAR.

Inhalt der Objektablage

Um einen Einblick in den Inhalt der Objektablage zu gewinnen, wählen Sie den Menüpunkt DATEI | NEU | WEITERE bzw. das Symbolleisten-Icon *Neu*. Sie erhalten den in Abbildung 1.12 gezeigten Dialog, der mehrere Seiten mit Icons enthält. Die Icons werden von einem *ListView*-Steuerelement dargestellt, das Sie wie die entsprechenden Elemente des Windows-Explorers auch auf andere Darstellungsmodi umstellen können (hier zu finden im lokalen Menü des *ListViews*).

Die Seiten *Neu* und *ActiveX* enthalten Vorlagen für verschiedene neue Objekte wie Units, Formulare, Anwendungen, Threads und natürlich verschiedene ActiveX-Objekte. Die drei oben beschriebenen Nutzungsoptionen entfallen für diese Objekte (je nach Delphi-Ausgabe finden Sie in der Objektablage noch weitere Seiten mit solchen Objekten). Eine Seite der Objektablage ist immer dem gerade geladenen Projekt gewidmet (in Abbildung 1.12 dem TreeDesigner-Projekt) und listet all seine Formulare auf. Mit Hilfe dieser Seite können auch Formulare innerhalb Ihres Projekts voneinander erben, ohne dass Sie das Formular, von dem geerbt werden soll, zuerst in die Objektablage einfügen müssen.

Die weiteren Seiten stellen den eigentlichen Inhalt der Objektablage dar. Sie enthalten Formulare und Projekte, denen Sie weitere hinzufügen können. Auch so genannte Experten und Wizards zum Anlegen von Formularen und Projekten werden auf diesen Seiten aufgeführt (Kapitel 7.2.4 behandelt beispielsweise den Datenbankformular-Experten).

Anpassen der Ablage und Hinzufügen von Objekten

Es stellt sich jetzt noch die Frage, wie Sie eigene Objekte in die Objektablage einfügen. Hier gilt es wieder, nach Projekten und Formularen zu unterscheiden:

- Für Formulare finden Sie in deren lokalem Menü den Punkt DER OBJEKTABLAGE HINZUFÜGEN..., der Sie in eine Dialogbox führt, in der Sie unter anderem ein Formular des aktuellen Projekts, eine Beschreibung, ein Icon und die Seite der Objektablage, auf der das Formular erscheinen soll, auswählen können.
- Mit demselben Dialog können Sie auch das Projekt in die Ablage einfügen, wenn Sie den Menüpunkt PROJEKT | DER OBJEKTABLAGE HINZUFÜGEN auswählen.

Rein organisatorischer Natur sind die Einstellmöglichkeiten, die Ihnen der Optionsdialog zur Objektablage bietet (TOOLS | OBJEKTABLAGE...). Neben den Optionen, die bereits im letzten Abschnitt erwähnt wurden, können Sie hier auch Objekte zwischen den einzelnen Seiten verlagern, neue Seiten erstellen und Objekte aus der Ablage entfernen (ähnlich der Anpassung der Komponentenpalette mit TOOLS | UMGEBUNGSOPTIONEN | PALETTE).

Wenn Sie neue Objekte in die Ablage einfügen, die Sie später in der Ablage modifizieren, ist es natürlich auch eine gute Idee, das gesamte Verzeichnis der Objektablage in etwaige Backup-Operationen Ihres Projekts einzubeziehen.

Komponentenschablonen

Wenn Sie viele verschiedene Formulare entwerfen, in denen manche speziell eingestellte Komponenten und Komponentengruppen immer wieder vorkommen, werden Sie wahrscheinlich die Komponentenschablonen schätzen lernen (z.B. für eine Schablone für das Duo aus *Ok* und *Abbruch*-Schalter, bei denen Sie sonst regelmäßig sechs verschiedene Property-Einstellungen im Objektinspektor vornehmen müssten, siehe hierzu Kapitel 1.9.1). Je nachdem für welchen Anwenderkreis Ihr Formular gedacht ist, wollen Sie den *Abbruch*-Schalter vielleicht auch immer mit einer Sicherheitsabfrage verknüpfen. Auch die hierzu notwendige Methode ließe sich in die Schablone aufnehmen.

Um eine einmal entworfene Komponente oder Komponentengruppe zu einer neuen Komponente in der Komponentenpalette zu machen, markieren Sie alle notwendigen Komponenten und führen den Menüpunkt KOMPONENTE | KOMPONENTENVORLAGE

ERZEUGEN... aus. In der daraufhin erscheinenden Dialogbox geben Sie der Schablone einen Namen, bestätigen das vorgegebene Icon oder wählen ein anderes und geben an, auf welcher Seite der Komponentenpalette die Schablone angezeigt werden soll.

Hinweis: Intern handelt es sich beim neuen Symbol in der Komponentenpalette nicht um eine *echte* Komponente. »Echte« Komponenten haben immer eine zugehörige Klasse, die Sie bei eigenen Komponenten auch selbst entwickeln müssen (zur Erweiterung oder Veränderung bestehender Komponenten siehe auch Kapitel 6.5.1).

Eine so in die Komponentenpalette eingefügte Schablone können Sie wie eine normale Komponente in jedes Formular einfügen. Delphi stellt dabei nicht nur alle Property-Werte der Komponenten wieder her, sondern übernimmt auch die Ereignisbearbeitungsmethoden, die die Komponenten im Original-Formular hatten, in das neue Formular. Nur die Namen der Komponenten (und folglich auch die Namen der Methoden) ändern sich bei dieser Einfügen-Operation (Delphi verwendet hier wieder die automatisch erzeugten Vorgabenamen wie *Button1* usw.).

Speicherung der Komponentenschablonen

Für den Fall, dass Sie auch die Komponentenschablonen in Ihr Backup einbeziehen wollen (und Sie nicht immer die gesamte Festplatte sichern): Die Komponentenschablonen werden nicht im Verzeichnis der Objektablage, sondern in der Datei `delphi.dct` im BIN-Verzeichnis der Delphi-Installation gespeichert.

1.6.5 Der Browser

Nachdem wir ein Projekt im letzten Kapitel aus der Sicht seiner einzelnen Dateien untersucht haben, kommen wir nun zu einer anderen Sichtweise des Projekts, wie sie im Browser dargestellt wird (Abbildung 1.13).

Hinweis: Der Browser weist gegenüber der Symbolanzeige von Delphi 1 bis 4 keine großen funktionellen Unterschiede auf. Zu allem, was im folgenden beschrieben wird, gibt es in der Symbolanzeige früherer Delphi-Versionen eine Entsprechung: Es gibt auch dort drei Arten von Listen (Units, globale Symbole und Klassen), zwei Arten der Sortierung, eine Suchfunktion, Detailinformationen in der rechten Fensterhälfte, aufgeteilt in BEREICH, VERERBUNG und REFERENZEN, eine Filtermöglichkeit und eine History-Funktion, die ein wenig dafür entschädigt, dass die Möglichkeit, mehrere Browser-Fenster zu öffnen, noch fehlt.

Der Browser liefert Ihnen Informationen über die Klassenhierarchie Ihres Projekts und der VCL und erlaubt es Ihnen, gezielt nach Klassen, deren Elementen und beliebigen anderen Programmsymbolen zu suchen. Dabei ist der Browser grundsätzlich zuverlässiger als die Hilfedatei, da er mit den aktuellsten, vom Compiler bereitgestellten Daten arbeitet, während die Hilfedatei schon nach einer kleinen nachträglichen Änderung der VCL durch Borland nicht mehr auf dem neuesten Stand wäre.

Sie können den Browser auf zwei Arten aufrufen:

- Mit ANSICHT | BROWSER erhalten Sie ein Fenster, in dem alle Symbole einer bestimmten Art und eines bestimmten Bereichs dargestellt werden[2].
- SUCHEN | SYMBOL ANZEIGEN... fragt Sie zunächst nach dem Namen eines Symbols. Dieses wird daraufhin in einem Browserfenster angezeigt, falls es gefunden werden konnte. Sie können nur aus der globalen Perspektive suchen. Um nicht-globale Symbole finden zu können, müssen Sie vorher das globale Symbol angeben, in dem das gesuchte Symbol enthalten ist; so finden Sie z.B. mit der Eingabe *TForm.Caption* das *Caption*-Property der Formularklasse.

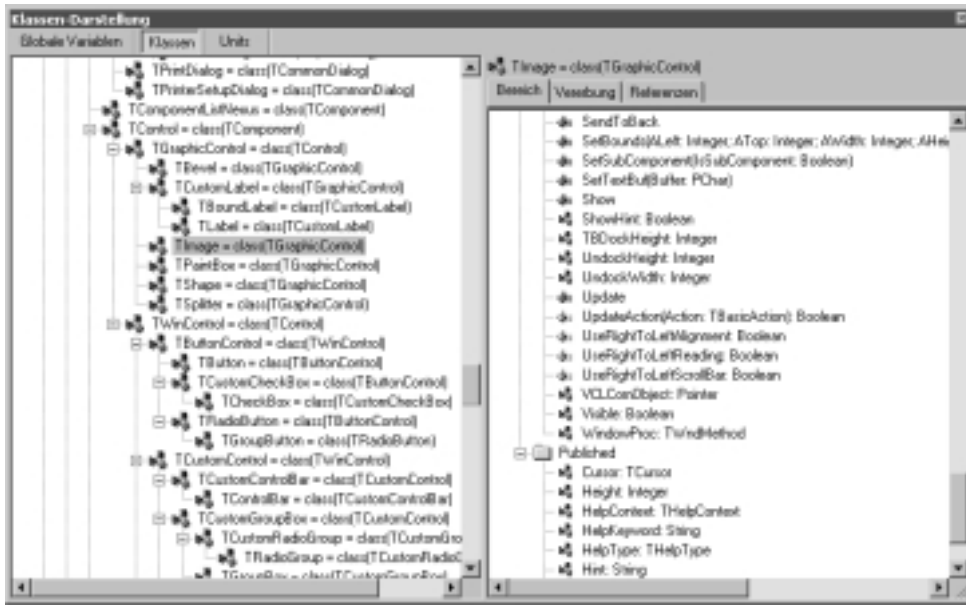


Abbildung 1.13: Der Browser

2 In den Versionen vor Delphi 5 heißt der Menüpunkt ANSICHT | SYMBOLANZEIGE und ist erst verfügbar, wenn der Compiler das aktuelle Projekt einmal übersetzt hat.

Die Eigenschaften des Browsers befinden sich zusammen mit denen des CodeExplorers auf der schon in Kapitel 1.6.1 beschriebenen Dialogseite (Abbildung 1.9). Zwei exklusiv für den Browser geltende Einstellungen wurden jedoch in Kapitel 1.6.1 noch nicht erläutert:

- die Anfangsansicht, die Sie nach dem Start des Browsers durch ANSICHT | BROWSER erhalten und für die KLASSEN, UNITS und GLOBAL zur Auswahl stehen,
- der Bereich der Symbole, der entweder nur das aktuelle Projekt oder das Projekt mit allen Symbolen der Delphi umfassen kann.

Ansichtsarten und Browserbereich

Die Ansichten, die Sie nachträglich auch über die drei Schalter unterhalb der Titelzeile des Browsers ändern können, haben folgende Bedeutung:

- Die KLASSEN-Ansicht zeigt einen Hierarchiebaum der im gewählten Browserbereich definierten Klassen an. Dieser beginnt in jedem Fall bei der allen Klassen zugrunde liegenden Basisklasse *TObject*. Falls der Browserbereich auch Interfaces umfasst, gibt es neben *TObject* noch eine zweite Wurzel namens *IUnknown*.
- UNITS zeigt eine Liste aller im Programm überhaupt vorkommenden Units an, also auch die Units, die Sie nicht direkt eingebunden haben, sondern die von irgendeiner anderen Unit per *uses*-Klausel geladen wurden. Sollte es sich dabei um Units handeln, die nicht in der Online-Hilfe erwähnt sind, dann sind dies Units, die von der VCL nur intern benötigt werden.
- Mit GLOBALE VARIABLEN erhalten Sie eine sehr lange Liste aller Bezeichner, die global, also weder innerhalb einer Klasse noch innerhalb einer Prozedur / Funktion / Methode definiert sind. Hierzu gehören sämtliche Funktionen der Laufzeitbibliothek, alle Konstanten wie z.B. die Farbkonstanten *cl...* usw. Übersichtlicher ist es, wenn Sie zuerst die Liste der Units anzeigen lassen und dann eine Unit auswählen. Sie erhalten dann im rechten Teil des Browsers eine Liste aller in dieser Unit definierten globalen Symbole.

Der Browserbereich ist normalerweise auf alle Symbole der VCL eingestellt, Sie können ihn im Eigenschaftsdialog aber auch auf die Symbole des aktuellen Projekts einschränken. Wenn Sie beispielsweise gerade ein neues Projekt mit nur einer Formular-Unit angelegt haben, so wird Ihnen der Browser bei dieser Einschränkung als einzige Variable die automatisch definierte Variable *Form1* zeigen und in der Klassenansicht wird es nur die Klasse *TForm1* geben sowie die Klassen, von denen *TForm1* abstammt.

Hinweis: Unter bestimmten Umständen – z.B. wenn der Quelltext aufgrund eines Fehlers nicht im Hintergrund übersetzt werden konnte – kennt der Browser nur die Projektsymbole, die beim letzten manuellen Kompilieren des Projekts definiert waren.

Bedienung

Die Bedienung eines typischen Browser-Fensters wie in Abbildung 1.13 ist intuitiv recht leicht erfassbar. Der rechte Fensterteil zeigt jeweils die Details zu dem im linken Bereich gewählten Symbol. Mit einem Doppelklick erhalten Sie zu den in beiden Bereichen aufgeführten Elementen weitere Informationen:

- Für die auf der Seite REFERENZEN angegebenen Quelltextpositionen, an denen ein bestimmtes Symbol verwendet wird, springen Sie direkt an die entsprechende Position im Quelltext-Editor.
- Bei allen anderen Einträgen erhalten Sie ein neues, kleines Browserfenster ohne Detailansicht. Sofern es sich bei dem Eintrag um eine Variable oder Methode handelt, enthält dieses Fenster lediglich die Seite REFERENZEN. Falls Sie auf eine Klasse doppelgeklickt haben, finden Sie alle drei möglichen Seiten BEREICH, VERERBUNG und REFERENZEN.

1.7 Der Debugger

Im Verlauf des bisherigen Beispielprogramms gab es noch keinen Anlass, den Debugger zu Rate zu ziehen, bei größeren Projekten ist er jedoch ein unverzichtbares Werkzeug, um fehlerhafte Programmteile auf frischer Tat zu ertappen. In den Delphi-Versionen 4 und 5 wurden sowohl Funktionsumfang als auch Komfort des Debuggers stark erweitert, so dass sich dieser jetzt nicht mehr vor anderen Debuggern zu verstecken braucht oder auf die Hilfe eines externen Debuggers angewiesen wäre.

Neben diesem integrierten Debugger befasst sich dieses Kapitel auch mit einem automatischen Hilfsmittel der Fehlersuche, den Assertions. Spezielle Einsatzzwecke des Debuggers werden in zwei anderen Kapiteln behandelt:

- Das Debuggen von DLLs wird anhand eines Praxisbeispiels in Kapitel 8.5.3 näher erläutert.
- Das Debuggen mehrerer Prozesse basiert auf dem Projekt-Manager. Siehe hierzu Kapitel 1.6.3.

Hinweis: Für ein bequemes Arbeiten mit den vielen Arten von Debugger-Fenstern empfiehlt es sich, eine Symbolleiste einzurichten, über die die einzelnen Fenster schnell aufgerufen werden können (siehe etwa Abbildung 1.17), und dann die entsprechenden Tastenkürzel zu lernen, die im Hinweisfenster unter der Maus angezeigt werden.

1.7.1 Übersetzungsoptionen für den Debugger

Nach der Installation ist die Delphi-IDE auf die Verwendung des internen Debuggers voreingestellt, so dass Sie alle ab Kapitel 1.7.2 beschriebenen Funktionen sofort ausprobieren können. Dieses Kapitel fasst die für den Debugger wichtigen Optionen zusammen, damit Sie eventuelle Fehleinstellungen schnell beheben können.

Der Debugger benötigt umfangreiche Informationen über die Symbole (Namen von Objekten, Variablen etc.), über den Aufbau der Komponenten und Klassen und über die Position der Quelltextzeilen im übersetzten Maschinencode. Das einzige Tool der Delphi-Umgebung, das sich damit richtig auskennt, ist der Compiler. Um den Debugger verwenden zu können, müssen Sie diesen also anweisen, die Informationen in einem für den Debugger leicht verständlichen Format abzulegen.

Zuerst einmal müssen diese Informationen in den DCU-Dateien abgelegt werden. Um die maximal mögliche Menge von Informationen zu erhalten, wählen Sie unter den Compileroptionen (in den Projektoptionen enthalten auf der Seite COMPILER) die ersten drei Optionen im Bereich *Debugger*, nicht aber das Markierungsfeld NUR DEFINITIONEN (Abbildung 1.14).

Des Öfteren kann es sich auch als nützlich erweisen, wenn Sie zum Debuggen die Optimierung abschalten, denn durch diese können Variablen und Quelltextzeilen wegoptimiert werden, so dass das Debuggen erschwert werden kann. Wenn Sie z.B. eine neue Variable einführen, die nur zum Debuggen dienen soll, der Sie dann einen Wert zuweisen (den Sie im Debugger untersuchen wollen), die Sie aber im Programmcode nicht weiter verwenden, dann optimiert der Compiler diese Variable standardmäßig weg, und Sie bekommen nur ein entschuldigendes Meldungsfenster, wenn Sie die Variable im Debugger untersuchen wollen.

Zum Abschalten der Optimierung haben Sie zwei Möglichkeiten:

- Schalten Sie die Optimierung global in den Projektoptionen ab (Abbildung 1.14),
- oder steuern Sie die Optimierung gezielt über die Angabe von Compilerbefehlen im Quelltext. So können Sie die Optimierung mit *{$Optimization Off}* z.B. vor einer Methode abschalten und mit *{$Optimization Off}* danach wieder anschalten (Sie können auch die Kürzel *{$O-}* und *{$O+}* verwenden).

Abbildung 1.14: Compileroptionen für Debugger- und Browsereinsatz

1.7.2 Allgemeine Debugger-Fenster

Die in diesem Kapitel erläuterten allgemeinen Debugger-Fenster wurden in Delphi 4 eingeführt; in der Personal- bzw. Standard-Ausgabe enthalten sind allerdings nur die beiden ersten davon: das Modul- und das CPU-Fenster.

Das Modul-Fenster

Sobald Sie eine Anwendung gestartet haben, steht Ihnen das Modul-Fenster (Abbildung 1.15) über das Menü ANSICHT | DEBUG-FENSTER | MODULE zur Verfügung. Sinn dieses Fensters ist es, alle Module aufzulisten, die an der Ausführung Ihrer Anwendung beteiligt sind. Dazu gehören die ausführbare EXE-Datei, alle direkt und indirekt benötigten DLLs sowie die Packages, falls Sie sich für die Nutzung der Package-Technologie entschieden haben (siehe Kapitel 6.1.3). Beim Mehrprozess-Debuggen zeigt es auch mehrere laufende Prozesse an und markiert den aktiven Prozess mit einem grünen Pfeil (jede ausführbare Datei wird zur Laufzeit zu einem *Prozess*).

Die beiden Bereiche am rechten und unteren Rand des Fensters zeigen die Details zu dem Modul, das im Hauptbereich gewählt ist:

- Der untere Bereich listet, sofern das Modul mit Debugger-Informationen übersetzt wurde, die für das Modul verwendeten Quelltextdateien auf. Über das lokale Menü können Sie natürlich gleich zur entsprechenden Quelltextdatei in den Editor springen.

- Der rechte Bereich listet – ebenfalls für das in der Modulliste gerade ausgewählte Modul – alle Einheiten des Programmcodes auf, die im Debugger verfügbar sind. Wenn Debugger-Informationen für das Modul vorliegen, sind dies alle Prozeduren, Funktionen, Methoden, die *initialization*- und *finalization*-Sektionen von Units sowie der Initialisierungs-Programmcode des Projekts (Hauptprogramm der Projektdatei, unter dem Namen des Projekts am Ende der Liste zu finden). Wie zu erwarten, können Sie über das lokale Menü direkt in den Editor zur entsprechenden Stelle in der Quelltextdatei springen.

  Aber auch ohne Debugger-Informationen erhalten Sie in diesem Fensterbereich normalerweise eine lange Liste, und zwar die Liste aller Einsprungspunkte, die von der gewählten DLL exportiert werden. Dazu gehören beispielsweise bei den DLLs des Betriebssystems alle API-Funktionen, die Sie aus einer Delphi-Anwendung aufrufen können. Seit Delphi 5 können Sie die Liste auch nach den Daten in einer der beiden Spalten sortieren.

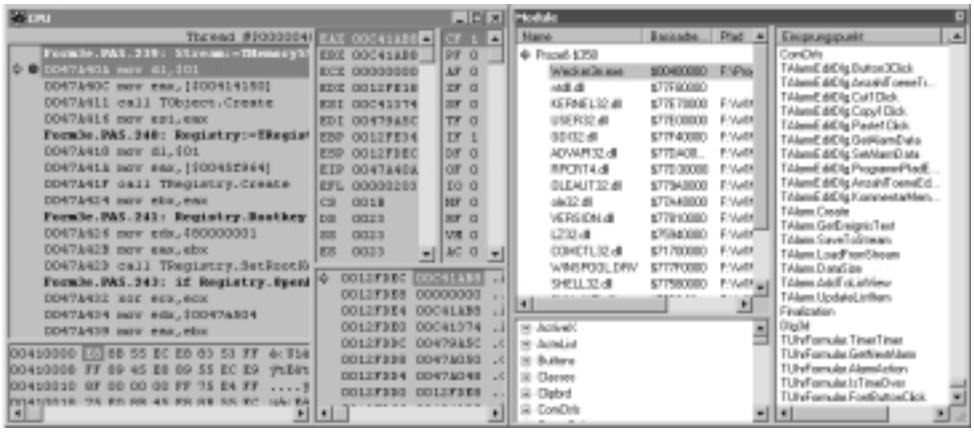


Abbildung 1.15: Zwei systemnahe Fenster des Delphi-Debuggers: CPU- und Modul-Fenster

Eine interessante Funktion des Modulfensters ist, dass Sie sich den Code *aller* Funktionen ansehen können, die in der Liste der Funktionen aufgelistet sind. Wenn es sich um eine Funktion handelt, die nicht im Quelltext vorliegt, wird die Funktion einfach in disassemblierter Form im CPU-Fenster dargestellt.

CPU- und FPU-Fenster

Im CPU-Fenster können Sie die Abarbeitung des Maschinencodes einer Anwendung »live« verfolgen, bei Verwendung von Fließkommazahlen assistiert dabei noch das FPU-Fenster. Während das CPU-Fenster (Abbildung 1.15) den Maschinen- und Assemblercode des Programms zusammen mit den zugehörigen Zeilen des Quelltextes dar-

stellt – sofern dieser vorliegt –, stellt das FPU-Fenster die Register der Fließkomma- bzw. MMX-Einheit des Prozessors dar. Zu den Einsatzgebieten dieser Fenster gehören:

- Schon alleine die Anzeige des Assemblercodes hat einen wichtigen informativen Charakter, denn wenn Sie über gute Assemblerkenntnisse verfügen, können Sie sich selbst ein Bild von der Arbeit des Compilers machen und davon, wie effektiv er Ihr Programm in Maschinencode übersetzt hat.
- Ähnlich wie beim Debuggen von Quelltext können Sie hier (Assembler-)Anweisungen einzeln ausführen, Haltepunkte setzen und die angezeigten Werte (hier also Register- und Speicherinhalte) testweise ändern.
- In die Abarbeitungsreihenfolge des Programms eingreifen: Über den lokalen Menüpunkt NEUER EIP können Sie die aktuelle Programmposition verändern. Stellen Sie sich beispielsweise vor, Sie stehen nach einer langen und komplizierten Debugger-Sitzung kurz vor einer wichtigen Entdeckung, gelangen dann aber an eine an sich unwichtige Anweisung, die Ihr Programm mit Sicherheit in den Absturz führen wird. Statt sich in dieses Schicksal zu fügen, die Absturz-Anweisung aus dem Programm zu entfernen, es neu zu übersetzen und die Debugger-Sitzung von vorne zu beginnen, setzen Sie die Ausführungsposition einfach hinter diese gefährliche Anweisung und lassen das Programm ohne Absturz weiterlaufen.
- Im Datenbereich des CPU-Fensters (links unten) können Sie den Hauptspeicher Ihrer Anwendung in einem maschinennahen Anzeigeformat untersuchen und ändern. Mit dem lokalen Menü können Sie zwischen verschiedenen Formaten wählen und eine Variable angeben, deren Speicherbereich Sie sehen wollen. Auch per Drag&Drop können Sie einen Variablennamen oder Ausdruck aus dem Editorfenster in den Datenbereich des CPU-Fenster ziehen.

Die vom Turbo Debugger bekannte Möglichkeit, während des Programmlaufs neue Assembleranweisungen einzugeben, existiert im CPU-Fenster übrigens noch nicht.

Das Ereignisprotokoll

Im Ereignis-Fenster (Abbildung 1.16) können Sie während des Programmlaufs Meldungen über fünf verschiedene Arten von Ereignissen sammeln, wenn Sie die entsprechenden Debuggeroptionen einstellen (TOOLS | DEBUGGER-OPTIONEN, Seite *Ereignisprotokoll*). Im Gegensatz zu Meldungsfenstern, bei denen der Benutzer OK drücken muss, führen diese Meldungen nicht zu einer Unterbrechung des Programmablaufs. Auf diese Weise können in kurzer Zeit hunderte solcher Meldungen gesammelt werden, wobei es dann an Ihnen liegt, sich aus einer langen Meldungsliste die entscheidenden Meldungen herauszusuchen. Die fünf Arten von Meldungen sind:

- Statusmeldungen der Prozessverwaltung (PROZESSMELDUNG): Hierzu gehört beispielsweise, dass jede von einem Prozess (EXE-Datei) verwendete DLL beim Starten des Prozesses in dessen Adressraum eingeblendet wird. Dieser Vorgang findet auch dann statt, wenn die DLL sich bereits im Speicher befindet und von einem anderen Programm verwendet wird. Dieser Ladevorgang wird im Protokoll vermerkt.
- HALTEPUNKT-MELDUNG: Wenn das Programm vom Debugger angehalten wird, notiert Delphi die genaueren Umstände dieser Unterbrechung ebenfalls im Protokoll. Als Unterbrechungsursache kommen nicht nur Haltepunkte, sondern auch Ausnahmebedingungen (Exceptions) innerhalb Ihres Programms in Frage.
- AUSGABEMELDUNG: Die Windows-API bietet jedem Programm über die Funktion *OutputDebugString* die Möglichkeit, während des Programmlaufs zusätzliche Meldungen zur Fehlersuche an einen Debugger zu senden. Seit Delphi 4 können Sie diese Meldungen auch von Delphis Debugger aus anzeigen lassen, was den Vorteil hat, dass sie mit den anderen Meldungen in einer chronologische Reihenfolge erscheinen, wie in Abbildung 1.16 gezeigt.
- FENSTERMELDUNGEN (oder auch falsch übersetzt als »MELDUNGSFENSTER«) sind Nachrichten, die von den Fenstern Ihrer Anwendung auf der Ebene des Windows-API empfangen werden. Um diese Informationen nutzen zu können, müssen Sie sich näher mit der Nachrichtenverarbeitung auf der Windows-Ebene auskennen. In Kapitel 3.1.4 wird der Zusammenhang zwischen der API-Ebene und den Delphi-Ereignissen erläutert. Hilfe zu den Typangaben der einzelnen API-Nachrichten (z.B. *WM_ActivateApp*, *WM_MouseMove* und *WM_NCPaint*) finden Sie in der Win32-Online-Hilfe.

  Die automatische Aufzeichnung der Fensternachrichten ist eine sehr einfache Alternative zum *WinSight*-Hilfsprogramm, das in allen Delphi-Versionen verfügbar ist und bei dem Sie die Nachrichtenflut ziemlich detailliert filtern können. Eine Beschreibung von *WinSight* würde jedoch den Rahmen dieses Kapitels sprengen.
- Schließlich können Sie über das lokale Menü des Fensters zusätzliche Kommentare in das Protokoll aufnehmen lassen.

Per Voreinstellung (BEIM START LÖSCHEN in den Debugger-Optionen) wird das Ereignisprotokoll bei jedem Neustart des Projekts geleert. Als sehr nützlich kann sich auch die Möglichkeit erweisen, den gesamten Inhalt in einer Datei zu speichern (lokales Menü) oder einen Teil des Protokolls zu markieren und in die Zwischenablage zu kopieren, denn nur so ist es beispielsweise möglich, nach einem bestimmten Text zu suchen.

Abbildung 1.16: Das Ereignisprotokoll hält Meldungen über verschiedene Geschehnisse während des Programmablaufs chronologisch fest.

1.7.3 Breakpoints

Delphi-Anwendungen sind aufgrund ihrer ereignisorientierten Struktur unmöglich schrittweise von Anfang bis Ende ablauffähig, denn zwischen den Ereignissen haben entweder Windows oder die VCL die Kontrolle. Sie werden daher wahrscheinlich häufig damit beginnen, einen Haltepunkt (*Breakpoint*) in eine Methode zu setzen, die offensichtlich nicht funktioniert oder in der Sie einen Fehler vermuten.

Delphi unterscheidet zwischen drei Arten von Haltepunkten:

- Quelltext-Haltepunkte, die sich im Quelltexteditor durch eine rot hervorgehobene Zeile widerspiegeln, an deren linkem Ende sich ein Stopschild befindet. Bevor die rot markierte Zeile vom Prozessor ausgeführt wird, hält Delphi das Programm an.
- Adress-Haltepunkte sind vergleichbar mit den Haltepunkten im Quelltext, befinden sich aber quasi im CPU-Fenster. Sie bewirken, dass das Programm vor der Ausführung einer Maschineninstruktion an einer bestimmten *Adresse* angehalten wird.
- Daten-Haltepunkte (in der Personal-Ausgabe nicht verfügbar) sind unabhängig von einer bestimmten Stelle im Quelltext, denn sie beziehen sich nicht auf den Code, sondern auf die Daten eines Programms und führen zu einer Unterbrechung, bevor ein Schreibzugriff auf die angegebenen Daten stattfindet.

Unabhängig von der Art erlaubt doch jeder Haltepunkt, nachdem er das Programm einmal angehalten hat, dieselben Untersuchungsmöglichkeiten. Solange das Programm unterbrochen ist, können Sie:

- sich Variablen und Objekte des Programms ansehen, um festzustellen, ob diese die erwarteten Werte aufweisen,
- Ausdrücke berechnen lassen (zu beidem siehe Kapitel 1.7.4),

- jede Anweisung einzeln ausführen lassen,
- den Aufruf-Stack untersuchen (zu beidem siehe Kapitel 1.7.5)
- und natürlich weitere Breakpoints setzen, bestehende löschen oder vorübergehend ausschalten.

Temporäre Breakpoints

Falls Sie nur vorübergehend einen Breakpoint benötigen, ist die Funktion ZU CURSORPOSITION GEHEN einfacher zu handhaben: Delphi setzt, ohne die Zeile zu markieren, einen temporären Breakpoint in die Zeile, in der sich momentan der Cursor befindet. Sobald diese Zeile erreicht wird, stoppt Delphi das Programm und löscht den Haltepunkt wieder. Falls das Programm vor dem Erreichen der gewünschten Zeile durch einen anderen Haltepunkt gestoppt wird, geht dieser temporäre Breakpoint jedoch verloren. Um dennoch bei dieser Zeile anzuhalten, müssen Sie den Cursor erneut darauf positionieren und die Funktion ZU CURSORPOSITION GEHEN ausführen.

Programmausführung fortsetzen

Um ein angehaltenes Programm weiterlaufen zu lassen, verwenden Sie denselben Befehl wie zum Starten des Programms, also START | START, F9, oder den entsprechenden Schalter der Symbolleiste.

Dauerhafte Breakpoints

Kommen wir zu den dauerhaften Haltepunkten, die über einige zusätzliche Merkmale verfügen können, und zwar sollen im Folgenden zunächst nur die in allen Delphi-Versionen verfügbaren Quelltext-Haltepunkte gemeint sein. Die einfachste Möglichkeit, einen solchen Haltepunkt zu setzen bzw. wieder abzuschalten, ist über das klassische Tastenkürzel Strg+F8 bzw. über einen Klick in das Editorfenster links neben der Haltezeile. Alle weitergehenden Optionen finden Sie in einem eigenen Fenster, das alle derzeitigen Unterbrechungspunkte auflistet (Menü: ANSICHT | DEBUG-FENSTER | HALTEPUNKTE) – Abbildung 1.17, Fenster *Haltepunkteliste*.

In diesem Fenster können Sie Haltepunkte über das lokale Popup-Menü vorübergehend abschalten, was gegenüber dem Löschen den Vorteil hat, dass Sie sie schnell wieder anschalten können, ohne zurück zur Unterbrechungsposition blättern zu müssen. Alle weiteren Merkmale der Haltepunkte werden in einem Dialogfenster bearbeitet, das Sie schnell mit einem Doppelklick oder mit ↵ aufrufen können, oder ab Delphi 5 auch über das Popup-Menü der Haltepunktmarkierung im Quelltext-Fenster.

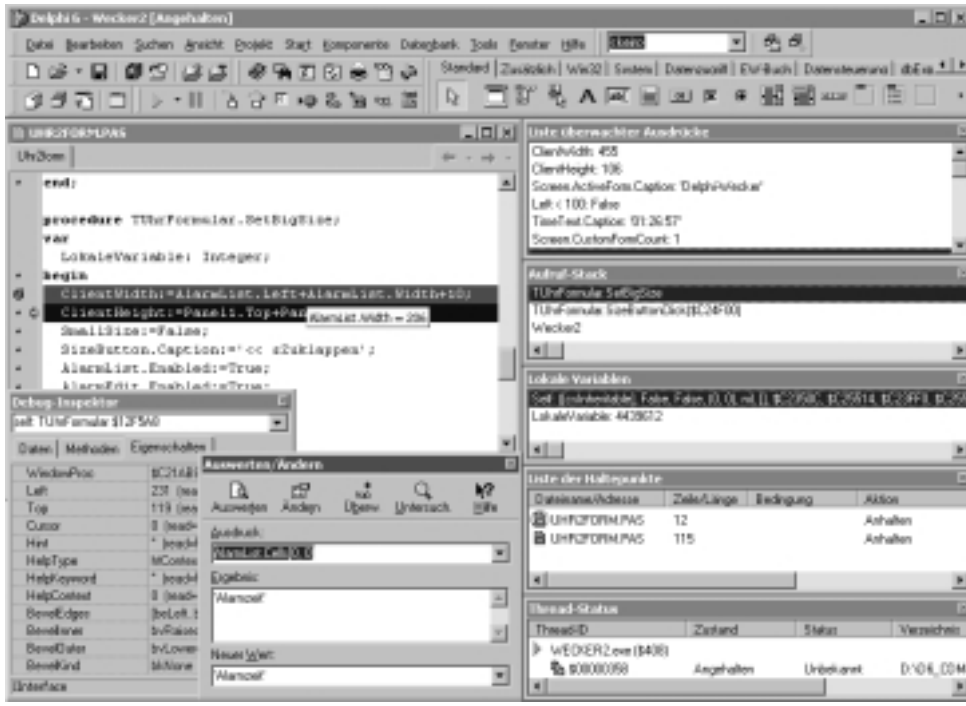


Abbildung 1.17: Übersicht über weitere Fenster des Debuggers inklusive eines temporär erscheinenden Kurzhinweises im Editor

Neben den Eigenschaften DATEINAME und ZEILENNUMMER, die ja schon durch das Setzen des Haltepunktes im Editor festgestellt werden, gibt es je nach Delphi-Version noch einige weitere:

- BEDINGUNG: Wenn Sie hier einen Ausdruck eintragen, der *True* oder *False* ergibt, z.B. »*StringList.Count<10*« , prüft Delphi jedes Mal, wenn der Haltepunkt erreicht wird, diese Bedingung und hält das Programm nur dann an, wenn sie zutrifft, also den Wert *True* ergibt.
- DURCHLAUFZÄHLER: Soll die Programmzeile erst mehrere Male ohne Unterbrechung durchlaufen werden, tragen Sie die Zahl der Durchläufe in diesem Feld ein. Es werden hierbei nur die Durchläufe gezählt, bei denen auch die Bedingung des vorhergehenden Feldes zutrifft. Wenn die Zahl der Durchläufe erreicht ist, hält Delphi beim nächsten Mal an dieser Stelle an und setzt den internen Durchlaufzähler dieses Haltepunkts auf Null zurück.
- Jedem Haltepunkt können Sie einen Gruppennamen zuweisen (ab Delphi 5). Die Gruppen dienen dazu, mehrere Haltepunkte gemeinsam ein- und ausschalten zu können. Das lokale Menü des Haltepunktfensters enthält zu diesem Zweck die

zwei neuen Punkte GRUPPE AKTIVIEREN und GRUPPE DEAKTIVIEREN, unter denen jeweils alle von Ihnen definierten Gruppennamen aufgelistet werden.

- Hinzukommen eine ganze Reihe von Einstellungen für Haltepunkt-Aktionen (im Haltepunkt-Dialog erreichbar über den Schalter WEITERE). Haltepunkt-Aktionen werden nachfolgend in einem eigenen Abschnitt erläutert.

Adress- und Datenhaltepunkte

Die Eigenschaften BEDINGUNG und DURCHLAUFZÄHLER finden Sie auch bei den Adress- und Datenhaltepunkten wieder, hinzu kommt bei den Adresshaltepunkten logischerweise noch eine Adresse, die Sie normalerweise dadurch festlegen, dass Sie einen Haltepunkt auf einer einzelnen Maschinen-Instruktion im CPU-Fenster setzen (per Mausklick an den linken Rand des CPU-Fensters oder über das auch für Quelltext-Haltepunkte verwendete Tastenkürzel).

Datenhaltepunkte werden durch eine Adresse und eine Längenangabe definiert:

- Die Adresse ist zwar hier nichts weiteres als eine achtstellige Hexadezimalzahl, Sie können diese jedoch auch festlegen, indem Sie eine Speicherposition wie in Object Pascal durch einen Variablennamen oder ähnlichen Ausdruck (z.B. *Zahlen[100]*) angeben.
- Die Länge gibt an, wie viele Bytes ab der genannten Adresse überwacht werden sollen. Für *Integer*-Variablen genügen schon die 4 standardmäßig voreingestellten Bytes; wenn Sie 10 Zeichen eines *Char*-Arrays überwachen wollen, geben Sie statt dessen eine 10 ein.

Der Debugger hält daraufhin das Programm an, noch bevor es zu einem Schreibzugriff auf die angegebene Adresse kommt. Falls möglich zeigt er die aktuelle Programmposition im Quelltexteditor an, Sie können daraufhin aber sogar in das CPU-Fenster schalten, um genau die Maschinenanweisung sehen zu können, die im nächsten Taktzyklus der CPU den Speicherzugriff durchgeführt hätte.

Datenhaltepunkte weisen einen grundsätzlichen Unterschied zu den Quelltext-Haltepunkten auf: Sie können erst gesetzt werden, nachdem das Programm gestartet wurde, und müssen nach jedem neuen Programmstart manuell über das Haltepunkte-Fenster wieder aktiviert werden. Dies liegt daran, dass sich die Adressen von Speicherobjekten (etwa Variablen) zwischen den Programmläufen ändern können und der Debugger daher bei jedem neuen Programmlauf erneut feststellen muss, welche Adresse gemeint ist. Dafür ist es erforderlich, dass das Programm angehalten wird, weil der Debugger immer eine bestimmte Programmposition als Kontext braucht, um den Variablennamen bestimmten Adressen zuordnen zu können (man denke hierbei etwa an dynamisch angelegte Objekte, die beim Programmstart noch gar nicht, oder an lokale Variablen, die nur innerhalb einer bestimmten Funktion gültig sind).

Hinweis: Datenhaltepunkte können die Ausführung Ihres Programms stark verlangsamen, auch wenn sie ausgeschaltet sind.

Haltepunkt-Aktionen

Die Haltepunkt-Aktionen erlauben Ihnen, beim Eintreten einer Haltepunkt-Bedingung bestimmte Aktionen vom Debugger automatisch ausführen zu lassen (ab Delphi 5). Sie können sogar darauf verzichten, dass der Debugger das Programm anhält und die Kontrolle an Sie weitergibt, so dass Sie aus einem Haltepunkt quasi einen Aktionspunkt machen. Neben der voreingestellten Aktion ANHALTEN stehen folgende Aktionen zur Wahl:

- Mit SPÄTERE EXCEPTIONS IGNORIEREN und SPÄTERE EXCEPTIONS BEHANDELN, verteilt auf zwei verschiedene Haltepunkte, können Sie einen bestimmten Programmteil von der standardmäßigen Exception-Behandlung ausklammern – etwa weil Ihnen bereits bekannt ist, dass darin zahlreiche Exceptions auftreten werden, die Sie aber nicht jedes Mal in einem Meldungsfenster angezeigt bekommen wollen. Gegenüber der generellen Abschaltung aller Exception-Reaktionen hat dies den Vorteil, dass Sie von eventuell überraschend auftretenden Exceptions in anderen Programmteilen dennoch informiert werden.
- MELDUNG PROTOKOLLIEREN schreibt den Text, den Sie in den Haltepunkt-Eigenschaften eingeben, in das Ereignisfenster.
- EVAL-AUSDRUCK wertet einen Ausdruck aus und protokolliert das Ergebnis im Ereignisfenster. Diese Protokollierung können Sie auch ausschalten, was dann Sinn macht, wenn bei der Auswertung des Ausdrucks Seiteneffekte entstehen, weil etwa eine Methode des Programms ausgeführt wird.
- GRUPPE AKTIVIEREN und GRUPPE DEAKTIVIEREN entspricht den gleichnamigen Punkten des Popup-Menüs und kann auf ähnliche Weise verwendet werden wie die anfangs genannten Optionen zum Ein- und Ausschalten der Exception-Behandlung.

1.7.4 Variablen untersuchen

Das häufigste Ziel beim Anhalten des Programms besteht darin, die Werte von Properties, Variablen und Ausdrücken zu untersuchen. Hierfür gibt es je nach Bedürfnis verschiedene Methoden.

Das Variablenfenster

Das Variablenfenster (ANSICHT | DEBUG-FENSTER | LOKALE VARIABLEN) zeigt ab der Professional-Version immer die gerade gültigen lokalen Variablen mitsamt ihren Wer-

ten an und erspart Ihnen dadurch eine Menge an manuellen Aufrufen von Variablen-Auswertungsfenstern. Wenn Sie die Optimierung nicht wie in Kapitel 1.7.1 angeraten abgeschaltet haben, kommt es jedoch sehr häufig vor, dass Variablen wegoptimiert wurden und so deren Inhalte nicht angezeigt werden.

Auswertung durch Kurzhinweis

Eine zweite sehr bequeme Möglichkeit der Variablenauswertung sind die ToolTip-Fenster (TOOLS | EDITOR-EIGENSCHAFTEN | PROGRAMMIERHILFE | AUSWERTUNG DURCH KURZHINWEIS): Wenn Sie bei angehaltenem Programm den Mauszeiger über einen Variablennamen im Quelltexteditor halten, erhalten Sie ein Hinweisfenster mit dem aktuellen Wert dieser Variablen. Diese Debuggerfunktion gehört zu den einzeln an- und abschaltbaren »Programmierhilfen« und verwechselt Variablen in *with*-Blöcken (hierzu und zu den Programmierhilfen siehe Kapitel 1.6.1).

Ausdrücke dauerhaft überwachen

Wollen Sie einen bestimmten Ausdruck über eine längere Zeit beobachten, können Sie ihn in ein so genanntes Watch-Fenster setzen. In der Übersetzung wurde *Watch* zu *Überwachte Ausdrücke*. Dieses Fenster (Menü ANSICHT | DEBUG-FENSTER | ÜBERWACHTE AUSDRÜCKE, in Abbildung 1.17: Fenster *Liste überwachter Ausdrücke*) enthält eine Liste von Ausdrücken, die Sie nach und nach erweitern, verändern und verkleinern können. Neben einem Ausdruck steht jeweils sein aktueller Wert. Jedes Mal wenn das Programm erneut angehalten wird, erneuert Delphi die Anzeige der Werte, da diese sich inzwischen geändert haben können. Es findet jedoch keine Aktualisierung des Fensters in Echtzeit statt, d.h., wenn das Programm einen Wert ändert, erfährt Delphi das erst, wenn das Programm das nächste Mal angehalten wird.

Um die Variable, die sich im Editor an der Cursorposition befindet, im Überwachungsfenster einzutragen, wählen Sie AUSDRUCK AM CURSOR ANZEIGEN... aus dem lokalen Menü des Editors bzw. das zugehörige Tastenkürzel. Wenn das Überwachungsfenster bereits angezeigt wird, können Sie die Maus auch zum Drag&Drop von markierten Ausdrücken vom Editor- in das Überwachungsfenster verwenden (wenn das Programm gerade nicht im Debugger untersucht wird, müssen Sie dabei noch Alt drücken). Im Überwachungsfenster selbst haben Sie eine Reihe weiterer Möglichkeiten, die Ihnen das dort befindliche Popup-Menü zeigt.

Angabe von Ausdrücken

Bei den Ausdrücken kann es sich um einzelne Variablen, Properties und Funktionsaufrufe oder um eine Verknüpfung von solchen handeln. Der Ausdruck

```
List1.ItemIndex >= LastSelection+1
```

ist zum Beispiel *True*, wenn das *ItemIndex*-Property einer *ListBox List1* größer dem Wert *LastSelection* plus 1 ist (bei diesem Beispiel wurde kein tieferer Sinn beabsichtigt). Mehr zu Ausdrücken in Object Pascal finden Sie in Kapitel 2.4.2.

Für die Ausdrücke in den Debugger-Fenstern gelten kleine Besonderheiten:

- Sie können darin auch Prozessorregister angeben (z.B. *eax*, *ebx*, *esi*).
- Hinter dem Ausdruck können Sie ein Komma und eine Formatanweisung folgen lassen. Mit der Angabe »eax, h« können Sie den Wert des eax-Registers beispielsweise in hexadezimaler Form überwachen. Eine Übersicht über die Formatanweisungen erhalten Sie in der Online-Hilfe unter dem Stichwort *Formatbezeichner*. Im Fenster der überwachten Ausdrücke brauchen Sie diese Formatanweisungen allerdings nicht, da Sie das Format dort durch eine Schaltergruppe festlegen können.
- Funktionsaufrufe liefern oft nicht nur ein Ergebnis, sondern führen im Programm eine Aktion aus und beeinflussen dadurch als so genannter *Nebeneffekt* den nachfolgenden Programmablauf. Daher sind Funktionsaufrufe im Überwachungsfenster standardmäßig nicht erlaubt, was Sie aber seit Delphi 5 im Eigenschaftsdialog der einzelnen Ausdrücke ändern können (Dialogfeld FUNKTIONSAUFRUFE GESTATTEN).

Inspektorfenster

Die erst wieder ab der Professional-Version mitgelieferten Inspektorfenster (siehe Abbildung 1.17) sind besonders dazu geeignet, Objekte und Arrays detailliert darzustellen. Sie können schnell mit Alt+F5 oder über das lokale Menü des Editorfensters aufgerufen werden (FEHLERSUCHE | UNTERSUCHEN...). Wenn Sie die Eingabemarkierung vorher auf einen Bezeichner im Quelltexteditor setzen, erhalten Sie sofort das zugehörige Inspektorfenster, ansonsten erfragt Delphi das anzuzeigende Programmobjekt über einen Dialog. Auch hier steht seit Delphi 5 wieder die Möglichkeit des Drag&Drop offen – vom Quelltext-Editor in ein bestehendes Inspektorfenster.

In einem Inspektorfenster wird jedes Objekt- bzw. Array-Element in einer eigenen Zeile dargestellt, Objekt-Inspektorfenster verteilen Daten, Properties und Methoden auf drei verschiedene Seiten. Da es beim Lesen von Properties über Lesemethoden in diesen Methoden zu unerwünschten Nebeneffekten kommen kann, werden solche Properties nicht automatisch vom Debugger ausgelesen und angezeigt. Um die Werte dieser Properties zu sehen, müssen Sie auf den Mini-Schalter neben dem Property drücken. Für die einzelnen Datenelemente können Sie außerdem weitere Inspektorfenster aufrufen und so nur per Mausklicks große verzweigte Zeigerstrukturen durchwandern.

Eine wichtige Option des Popup-Menüs ist die TYPUMWANDLUNG, mit der Sie dem Debugger den »wahren« Typ eines polymorphen Objekts mitteilen können. Inspizieren Sie beispielsweise eines der häufig als Parameter auftretenden *Sender*-Objekte,

zeigt Ihnen das Inspektorfenster nur die in *TObject* deklarierten Datenelemente an, da *Sender* als *TObject* deklariert ist. Wenn es sich bei *Sender* aber z.B. um eine *TListBox*-Komponente handelt, müssen Sie die Klasse *TListBox* zuerst im Typumwandlungs-Dialog angeben, um auch die exklusiven Datenelemente von *TListBox* angezeigt zu bekommen.

Ausdrücke auswerten, Objekte untersuchen und Werte ändern

Schließlich gibt es in Delphi noch ein AUSWERTEN / ÄNDERN-Fenster (ebenfalls in Abbildung 1.17 gezeigt). Gegenüber den Kurzhinweisfenstern hat er die folgenden Vorteile anzubieten:

- Sie können wie im Überwachungsfenster auch Ausdrücke auswerten und Formatanweisungen angeben;
- Sie können Variablen angeben, die gerade nicht im Editor genannt werden, vor allem die Variable *self* für das gerade aktive Objekt;
- Sie können Variablenwerte ändern.

Die letzten beiden Vorteile können Sie natürlich auch bei Verwendung der Inspektor-Fenster genießen. Das Auswerten / Ändern-Fenster finden Sie im globalen START-Menü, im lokalen Editormenü und je nach gewählter Tastaturbelegung unter Strg+F7 oder Strg+F4.

1.7.5 Code-Ausführung

Wenn Sie einige Ausdrücke in das Überwachungsfenster geschrieben haben oder wenn Sie nur den schrittweisen Ablauf des Programms verfolgen wollen, können Sie sich der Einzelschrittbefehle des Debuggers bedienen. Davon gibt es zwei verschiedene Typen, deren (englische) Kurzbezeichnung *Step* bzw. *Trace* ist. In der Symbolleiste finden Sie beide Befehle in den beiden Schaltern rechts unten. Im START-Menü haben sie die folgenden Namen:

- Mit GESAMTE ROUTINE (Step) führen Sie den Quelltext zeilenweise aus. Die Übersetzung rührt daher, dass Funktionsaufrufe in diesem Einzelschrittmodus als Ganzes ausgeführt werden.
- Im Gegensatz dazu gelangen Sie mit EINZELNE ANWEISUNG in jede Funktion, die in der aktuellen Zeile aufgerufen wird und zu der der Quelltext vorliegt. Allerdings ist auch hier der bescheidenere Name *Trace* besser, denn wenn sich mehrere einfache Anweisungen ohne Routinenaufruf in einer Zeile befinden, werden diese mit EINZELNE ANWEISUNG eben nicht einzeln, sondern als gesamte Zeile ausgeführt.

Schon eine der bisher gezeigten kleinen Methoden zeigt die Unterschiede:

```
        procedure TUhrFormular.TimerTimer(Sender: TObject);
{ Z.1 }begin
{ Z.2 }  TimeText.Caption := TimeToStr(Time);
{ Z.3 }  if IsTimeOver(AlarmEdit.Text)
{ Z.4 }    then AlarmMessage('Alarm zur Zeit '+AlarmEdit.Text);
{ Z.5 }end;
```

Mit *einzelner Anweisung* besuchen Sie die Zeilen 1 und 2 und überspringen dabei den Aufruf von *TimeToStr*, da dieser nicht im Quelltext vorliegt. In Zeile 3 bringt Sie *einzelne Anweisung* dann auf das *begin* der selbst definierten Methode *IsTimeOver*. Nachdem Sie diese schrittweise ausgeführt haben und bei deren *end* angelangt sind, können Sie einen weiteren Einzelschritt machen, um zur obigen Methode zurückzugelangen, und zwar in Zeile 4, falls die *if*-Abfrage *True* ergibt, ansonsten in Zeile 5. Auch die Anweisung in Zeile 4 ist selbst definiert, weshalb Sie sie einzeln abarbeiten können.

Nachdem dies geschehen ist, erreicht Ihr Programm als Letztes die abschließende Zeile 5, von der es nicht mehr leicht vorhersehbar weitergeht, da die Methode *TimerTimer* von der VCL aufgerufen wurde. Wenn Sie auf dem *end* wieder einen der Einzelschrittbefehle ausführen, hält Delphi das Programm bei der nächsten Ereignisbearbeitungsmethode, die aufgerufen wird, an.

Würden Sie in diesem Beispiel statt *einzelner Anweisung* den *ganzen Routinen*-Modus wählen, würden Sie auch die Zeilen 3 und 4 in einem Schritt durchlaufen.

Beim schrittweisen Ausführen des Programms mit dem Befehl GANZE ROUTINE setzt der Debugger einen temporären Breakpoint auf die nächste Zeile. Dieser verhält sich so wie der oben beschriebene temporäre Haltepunkt beim Springen zur Cursorposition.

Größere Schritte

Ein weiterer nützlicher Eintrag im START-Menü ist AUSFÜHRUNG BIS RÜCKGABE. Er lässt die aktuelle Methode vollständig ablaufen und hält das Programm dann bei ihrem abschließenden *end;* wieder an.

Interessant ist auch ein spezieller Einzelschritt-Befehl, der sich lohnt, wenn Sie sich im CPU-Fenster und in einem Bereich befinden, zu dem es keinen Quelltext gibt: START | NÄCHSTE QUELLTEXTZEILE lässt das Programm dann bis zur nächsten Quelltextzeile weiterlaufen und hält es dort an.

In diesem Zusammenhang sei auch noch einmal die in Kapitel 1.7.3 beschriebene Funktion ZU CURSORPOSITION GEHEN erwähnt, die Sie nicht nur als temporären Breakpoint, sondern auch zum Überspringen mehrerer Anweisungen einsetzen können.

Der Aufruf-Stack

Abbildung 1.17 enthält (außer dem Fenster *Thread-Status*, das wir hier nicht weiter beachten) mit *Aufruf-Stack* noch ein letztes, bisher nicht erwähntes Fenster. Es zeigt die Funktionsaufrufe, die zurzeit aktiv sind, soweit der Debugger den Aufruf-Stack zurückverfolgen kann. So zeigt das Fenster in der Abbildung beispielsweise, dass die Methode *SetBigSize* von der Methode *SizeButtonClick* aufgerufen wurde. Um die Zeile zu finden, in welcher der Aufruf einer Methode stattfand, doppelklicken Sie im Aufruf-Stack-Fenster auf diese Methode, und Sie gelangen zu der Zeile, die dem Aufruf folgt.

Wenn Sie den Quellcode der VCL besitzen und ihn mit Debug-Informationen in das Programm eingebunden haben, können Sie im Stack-Fenster mitunter sogar viel über die VCL lernen (Abbildung 1.18).

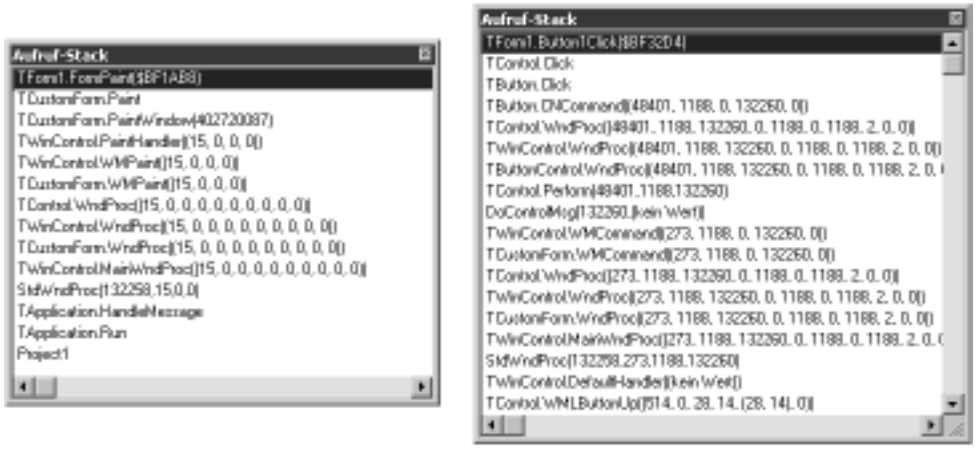


Abbildung 1.18: Diese Stack-Fenster zeigen die Hintergründe zum Aufruf des OnPaint-Ereignisses eines Formulars und des OnClick-Ereignisses eines Schalters.

Debuggen mit dem VCL-Quelltext R141

Auch, wenn es meistens sehr vorteilhaft ist, dass die VCL unzählige Details vor dem Entwickler versteckt, kann es sehr interessant sein, ihre internen Abläufe per schrittweiser Programmausführung genauestens zu untersuchen. Dies kann großen Nutzen bei der Fehlersuche bringen und vor allem förderlich für das Verständnis der VCL sein (da Sie z.B. Informationen wie in Abbildung 1.18 erhalten können). Damit der integrierte Debugger in Besitz der Debugging-Informationen der VCL kommt, genügt es seit Delphi 5, in den Projektoptionen auf der Seite COMPILER die Option MIT DEBUG-DCUS zu markieren und das Projekt neu zu erzeugen (PROJEKT | PROJEKTNAME ERZEUGEN).

Eine allgemeine Vorgehensweise, die zum selben Ergebnis führt, aber zusätzlich noch die Vorteile hat, dass eventuelle Änderungen im VCL-Quelltext mitkompiliert werden und dass sie in allen Delphi-Versionen funktioniert, ist folgende:

- Zunächst müssen die Optionen aktiviert sein, die sowieso schon zum Debuggen notwendig sind.
- Fügen Sie im Feld *Pfad für Bibliothek* in den Umgebungsoptionen auf der Seite *Bibliothek* den Pfad des VCL-Quelltextes ein (für Delphi 6 z.B. D:\BORLAND\DELPHI6\SOURCE\VCL). Das VCL-Verzeichnis muss dabei vor dem LIB-Verzeichnis stehen und von diesem durch ein Semikolon getrennt werden.
- Übersetzen Sie dann Ihr aktuelles Projekt komplett neu.

Egal, welche Vorgehensweise Sie gewählt haben, ein schneller Test, ob es funktioniert hat, ist folgender: Starten Sie das Projekt schließlich mit START | EINZELNE ANWEISUNG. Wenn die Installation der VCL-Debug-Informationen erfolgreich war, müsste der Debugger Ihr Programm statt in einer Ihrer Dateien im *initialization*-Bereich einer der VCL-Units anhalten.

Hinweis: Auch wenn Sie die VCL-Verzeichnisse in den Projektoptionen unter VERZEICHNISSE/BEDINGUNGEN in den SUCHPFAD eintragen und das Projekt neu erzeugen, kompiliert Delphi die Quelltextdateien der VCL neu und mit Debug-Informationen, falls die entsprechende Option aktiviert ist.

1.7.6 Assertions

Mit den in den bisherigen Abschnitten erläuterten Debugger-Funktionen können Sie sich während des Programmablaufs selbst davon überzeugen, ob alles wie erwartet abläuft, ob z.B. eine *while*-Schleifen-Variable richtig bis zum Wert 0 »heruntergezählt« wird – kurz: Sie können interaktiv überprüfen, ob bestimmte *Bedingungen* zutreffen.

Delphi unterstützt seit der Version 3 auch das maschinelle Überprüfen solcher Bedingungen in einer besonderen Weise:

- Mit der neuen Standardanweisung *Assert* können Sie eine Bedingung überprüfen, die beim korrekten Ablauf des Programms gelten muss. *Assert* zeigt ein Meldungsfenster an, wenn die Bedingung einmal nicht erfüllt ist. So gesehen ähnelt die Anweisung *Assert(Bedingung, Meldung)* erst einmal nur der Anweisung *if not Bedingung then Meldunganzeigen(Meldung)*.
- Ihren besonderen Status erhält die *Assert*-Anweisung jedoch dadurch, dass Sie in den Compileroptionen einstellen können, ob die *Assert*-Anweisungen im Quelltext beachtet werden sollen oder nicht (siehe auch Abbildung 1.14, Schalter *Assertion*).

Während der Programmentwicklung können Sie also durch großzügigen Gebrauch dieser Bedingungstests möglicherweise Fehler schneller finden. Wenn das Programm dann fertig ist, schalten Sie alle *Assert*-Anweisungen auf einmal aus. Angenommen, dass in Ihrem fertigen Programm alle gewünschten Bedingungen sowieso immer zutreffen, dann hat das Abschalten der *Assert*-Anweisungen noch den Vorteil, dass die Ablaufgeschwindigkeit des Programms erhöht und die Größe der kompilierten Datei reduziert wird. Ein ausführlicheres Beispiel:

```
i:=1;
repeat
  Assert(i<100, 'i<100 gilt nicht');
  i:=i+2;
until i=100;
```

In diesem Programmbeispiel soll sichergestellt werden, dass die Schleife wirklich nur so lange ausgeführt wird, wie *i* kleiner als 100 ist. Da *i* jedoch nur ungerade Werte annimmt und am Ende der Schleife nicht *i>=100* abgefragt wird, erreicht *i* irgendwann den Wert 101, ohne dass die Schleife abgebrochen wird. Sofort schlägt die *Assert*-Anweisung zu und weist auf diesen Fehler hin.

Hinweis: Bei den voreingestellten Umgebungsoptionen für den Debugger unterbricht der integrierte Debugger das Programm an der Stelle der fehlgeschlagenen *Assert*-Anweisung. Die Werte der Variablen sind zu diesem Zeitpunkt jedoch schon nicht mehr definiert. Wenn Sie den Wert von *i* anzeigen lassen, erhalten Sie also in diesem Beispiel mit höchster Wahrscheinlichkeit nicht mehr den Wert 101, sondern beispielsweise 4349104.

Gegenüber den *Assert*-Makros aus C++ hat die Object-Pascal-Anweisung zwar den Nachteil, dass der genaue Wortlaut der Bedingung nicht automatisch im Meldungsfenster angezeigt wird, sondern dass Sie sie in den Meldungsstring kopieren müssen, trotzdem erspart Ihnen eine einzige *Assert*-Anweisung den folgenden Code, der – hundertfach angewendet – Ihre Quelltextdatei ganz schön unübersichtlich machen würde:

```
{$ifdef AktiviereAssertions}
  if not (i<100) then
    // Meldung ausgeben (bzw. Exception erzeugen, siehe Hinweis unten)
{$endif}
```

Hinweis: Genau genommen ist es nicht die Anweisung *Assert*, die das Meldungsfenster öffnet, sondern die automatische Exception-Behandlung der VCL. *Assert* erzeugt lediglich eine *EAssertionFailed*-Exception, durch die die Ausführung des Codes, in der die Bedingung nicht zutrifft, abgebrochen wird. Daraufhin übernimmt die VCL die Kontrolle und gibt das Meldungsfenster aus. Weitere Details zu Exceptions lesen Sie in Kapitel 2.6.

1.8 Noch mehr Praxis: Verbesserung des Beispielprogramms

In diesem Praxisteil soll das Beispielprogramm um einige Funktionen erweitert werden, anhand derer Sie einige weitere Techniken der Delphi-Programmierung kennen lernen können, wie etwa den Einsatz von (Markierungs-)Schaltern, das Aktivieren und Deaktivieren von Eingabeelementen, die Behandlung von Exceptions, die Veränderung der Fenstergröße durch das Programm sowie die Verwendung der etwas komplexeren Komponente *StringGrid*.

Die zweite Version des Beispielprogramms soll mehrere Alarmzeiten berücksichtigen und dem Benutzer gestatten, die Alarmfunktion aus- und wieder einzuschalten. Damit der Benutzer die verschiedenen Alarmzeiten leichter unterscheiden kann, soll außerdem jede Alarmzeit mit einem Hinweistext versehen werden können, der bei Eintreten des Alarms vom Programm angezeigt wird.

Es sei gleich angemerkt, dass wir diese Erweiterungen ziemlich direkt in das Programm einbauen werden, ohne große vorherige Planung. Da sich noch kein einziger Leser der Vorgängerbücher über diese Vorgehensweise beschwert hat, werden wir sie weiter beibehalten und die eventuell aus unserer fehlenden Vorausschau resultierenden Komplikationen erst in Kapitel 1.9 ausräumen.

Die »Ad-hoc-Erweiterung« des bisherigen Programms sieht so aus, dass das Eingabefeld für einen Alarm einfach ausgetauscht wird gegen eine Komponente, in die der Benutzer mehrere Alarme mitsamt zugehörigen Hinweistexten tabellarisch eingeben kann (siehe Abbildung 1.19).

1.8.1 Erweiterung des Beispielformulars

Die zweite Version des Beispielformulars (auf der CD-ROM im Projekt *Wecker2* enthalten), weist die folgenden Ergänzungen gegenüber der ersten Version auf:

- ein Markierungsfeld, durch das die Alarmfunktion ein- und ausgeschaltet werden kann (hierzu benötigen wir eine *CheckBox*-Komponente)

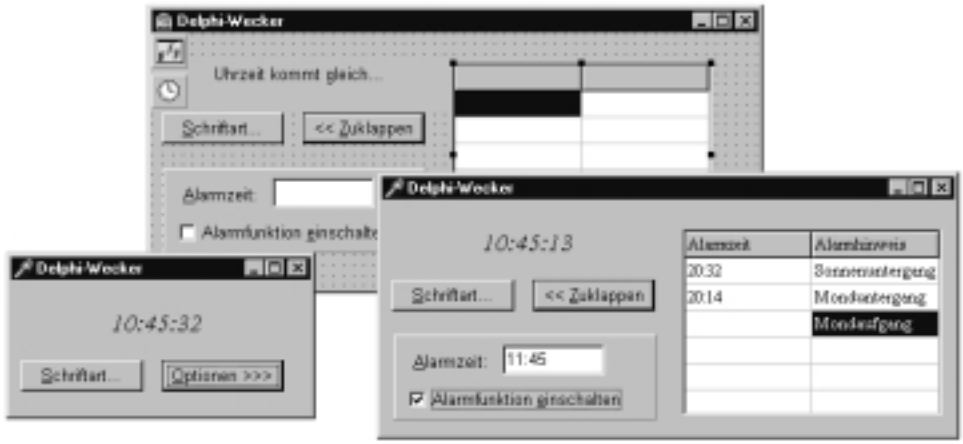


Abbildung 1.19: Die neue Version des Beispielprogramms zur Entwurfszeit (im Hintergrund, mit provisorischer Größeneinstellung) und zur Laufzeit (im Startzustand und im aufgeklappten Zustand)

- eine Stringtabelle zum Eintragen weiterer Alarmzeiten (die Komponente *StringGrid* befindet sich in der Komponentenpalette auf der Seite ZUSÄTZLICH)
- einen Schalter (*Button*), mit dem das Fenster »auf- und zugeklappt« werden kann, so dass die Alarmzeiten entweder angezeigt werden oder nicht.

Da die Stringtabelle (*StringGrid*) eine sehr flexible Komponente ist und mit sehr allgemeinen Einstellungen vorbelegt ist, müssen wir hier sehr viele Properties ändern. Die Tabelle listet alle geänderten Properties der drei neuen Komponenten auf sowie zwei Properties des Formulars, die auch angepasst werden müssen:

Komponente	**Property**	**Wert**
StringGrid	Name	AlarmList
	ColCount	2
	DefaultColWidth	100
	DefaultRowHeight	20
	FixedCols	0
	Font.Name	Times New Roman
	Options	alle voreingestellte Flags + *goEditing*
	ParentFont	False
	RowCount	7
	TabOrder	4
Button	Name	SizeButton
	Caption	&weitere >>>

Komponente	Property	Wert
	Default	True
	TabOrder	2
CheckBox	Name	AlarmActive
	Caption	Alarmfunktion &einschalten
	TabOrder	3
UhrFormular	VertScrollBar.Visible	False
	HorzScrollBar.Visible	False

Die *TabOrder*-Properties der schon in der ersten Formularversion vorhandenen Elemente sollten die Werte 0 und 1 aufweisen, damit sie zu den neuen *TabOrder*-Properties passen.

Die Properties, die im bisherigen Verlauf des Beispielprogramms noch nicht besprochen wurden, sind:

- *ColCount* und *RowCount* geben die Anzahl der Spalten bzw. Zeilen der Stringtabelle an. Der obere Teil der Zeilen und die links befindlichen Spalten können *befestigt*, also nicht scrollbar sein. Wie viele Spalten bzw. Zeilen das sind, geben Sie in *FixedCols* bzw. *FixedRows* an. Die Stringtabelle hat eine feste Zeile, die zur Beschriftung der beiden Spalten dient. Diese feste Zeile wird grau dargestellt und kann nicht gescrollt werden. Letzteres wirkt sich jedoch erst dann aus, wenn die Zahl der Zeilen so groß ist, dass nicht alle Zeilen dargestellt werden können (was in Beispielprogramm *Wecker2* nicht der Fall ist). Wenn Sie dann die unteren Einträge über die Schiebeleiste ins Bild rollen, bleibt die befestigte Zeile sichtbar, direkt unter ihr erscheint dann aber eventuell die vierte, fünfte oder sechste Zeile. Abbildung 1.20 zeigt eine Demonstration der StringGrid-Komponente, bei der die elfte Spalte der Zeile 9 in die linke obere Ecke geschoben wurde.
- In *DefaultColWidth* und *DefaultRowHeight* geben Sie die voreingestellte Größe der Spalten bzw. Zeilen an. Zur Laufzeit kann der Benutzer die Breite mit der Maus verändern, wenn Sie im Property *Options* die Flags *goColSizing* bzw. *goRowSizing* einschalten. Im Beispielprogramm ist dies nicht erforderlich.
- Das wichtigste Flag im Property *Options* ist *goEditing* – es erlaubt zur Laufzeit, den Inhalt der Tabelle zu editieren. Wenn es ausgeschaltet ist, können Sie statt dessen einen rechteckigen Bereich von Zellen mit der Maus markieren. Falls Sie dies in Ihrem Programm ebenfalls tun möchten, so benötigen Sie eine Möglichkeit, den Editiermodus der Stringtabelle zur Laufzeit umzuschalten (beispielsweise einen einfachen Markierungsschalter, in dessen *OnClick*-Methode Sie das Flag *goEditing* löschen oder setzen).

- *Default* ist ein Property der Komponente *Button* und macht diese zum voreingestellten Schalter. Das bedeutet, dass Sie ihn aus der ganzen Dialogbox heraus mit der Eingabetaste aufrufen können, es sei denn, die Eingabetaste wird vom Element, das den Tastaturfokus besitzt, bereits »verschluckt« (beispielsweise von einem mehrzeiligen Editierfeld).
- VertScrollBar.Visible und HorzScrollBar.Visible werden auf False gesetzt, damit das Formular im verkleinerten Zustand keine überflüssigen Bildlaufleisten anzeigt, mit denen Sie sonst die unsichtbaren Teile des Formulars in den Sichtbereich schieben könnten.

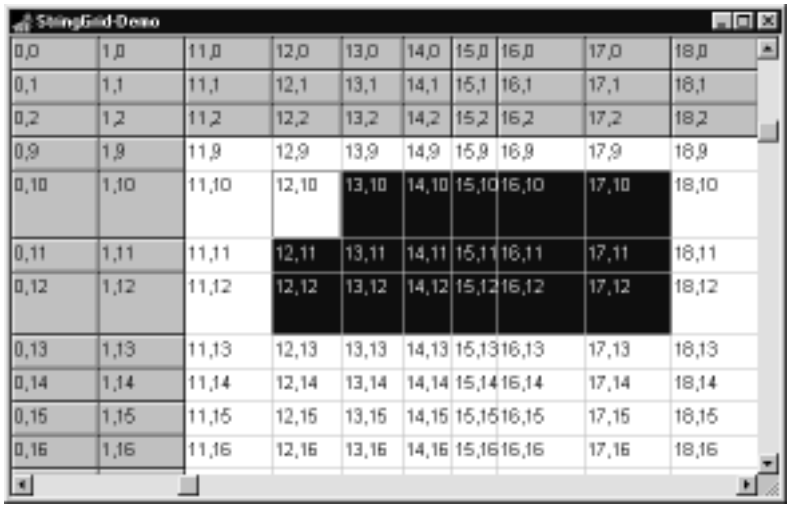


Abbildung 1.20: Die StringGrid-Komponente im Einsatz

Die Beschriftung der Stringliste ist nicht zur Entwurfszeit möglich, sondern muss in der *OnCreate*-Methode nachgeholt werden, die wir im nächsten Kapitel besprechen. Auch in Kapitel 1.8.4 werden wir noch einmal auf die interessante *StringGrid*-Komponente zurückkommen.

1.8.2 Anpassen der Fenstergröße zur Laufzeit

Wir implementieren als Erstes die Möglichkeit, das Formular aufzuklappen, damit die Alarmliste sichtbar wird. Um diese zu verdecken bzw. zu zeigen, müssen wir Breite und Höhe des Formulars zur Laufzeit anpassen. Das aufgeklappte Fenster soll so breit sein, dass der rechte Rand der *StringGrid*-Komponente sichtbar ist und dass daneben noch ein kleiner Abstand zum Fenster bleibt. Allerdings soll nicht die Breite des gesamten Fensters berechnet werden, die sich im Property *Width* befindet und zu der

auch der Fensterrahmen gehört, sondern nur die Breite seines Arbeitsbereichs (ansonsten müssten wir auch noch die Breite des Fensterrahmens, die je nach Systemkonfiguration variieren kann, in Erfahrung bringen). Die Breite des Arbeitsbereichs eines Fensters können Sie durch das Property *ClientWidth* ebenso leicht ändern wie die Breite des gesamten Fensters (Property *Width*).

Die erforderliche Breite des Arbeitsbereichs lässt sich für das aufgeklappte Fenster aus dem linken Rand der Alarmliste (*AlarmList.Left*), aus ihrer Breite (*AlarmList.Width*) und aus dem zusätzlichen Rand berechnen. Die Summe dieser drei Teile weisen wir also dem Property *ClientWidth* des Formulars zu:

```
ClientWidth := AlarmList.Left+AlarmList.Width+10;
```

Für die zugeklappte Version orientiert sich das Programm erneut am linken Rand der Alarmliste, schneidet das Fenster aber an dieser Stelle ab. Wir schreiben für das Auf- und Zuklappen zwei neue Methoden, die wieder von Hand in der Formulardeklaration erwähnt werden müssen, vorläufig lauten diese:

```
procedure TUhrFormular.SetBigSize;
begin
  ClientWidth := AlarmList.Left+AlarmList.Width+10;
  ClientHeight := Panel1.Top+Panel1.Height+10;
  SmallSize := False;
  { Schaltertext zeigt die Zuklapp-Möglichkeit: }
  SizeButton.Caption := '<<<<';
end;

procedure TUhrFormular.SetSmallSize;
begin
  ClientHeight := Panel1.Top-1;
  ClientWidth := AlarmList.Left-1;
  SmallSize := True;
  { Schaltertext zeigt, dass das Fenster
    aufgeklappt werden kann: }
  SizeButton.Caption := 'weitere >>>';
end;
```

Zusätzlich zu den schon erwähnten Größenberechnungen wechseln die beiden Methoden die Beschriftung (*Caption*) des Schalters. Je nachdem, ob das Formular ganz oder nur teilweise sichtbar ist, soll er anzeigen, dass seine nächste Betätigung zur Vergrößerung bzw. zur Verkleinerung führt.

Die Methode für den Schalter

Jetzt ist es Zeit, das Ereignis *OnClick* für den Schalter, der das Fenster auf- und zuklappt, zu implementieren. Diese muss lediglich die beiden gerade geschriebenen Hilfsmethoden aufrufen und sich in einer neuen Formularvariablen merken, welche Größe zurzeit eingestellt ist:

```
type
  TUhrFormular = class(TForm)
   ...
  private
  { neue selbst erzeugte private Variablen
      und Methoden des Formulars: }
    SmallSize: Boolean; { << neue Variable }
    procedure SetBigSize;
    procedure SetSmallSize;
   ...
  end;
...
{ neue von Delphi erzeugte Methode }
procedure TForm1.SizeButtonClick(Sender: TObject);
begin
  if SmallSize then SetBigSize
    else SetSmallSize;
end;
```

Initialisierung

Schließlich soll das Fenster beim Programmstart in der kleinen Version angezeigt werden. Dazu verknüpfen wir das *OnCreate*-Ereignis mit einer Initialisierungsmethode, in der wir die Methode *SetSmallSize* aufrufen (dadurch wird dann auch die Formularvariable *SmallSize* initialisiert). Eine zweite noch ausstehende Aufgabe für die *OnCreate*-Methode ist die Beschriftung der Stringtabelle, denn diese können Sie nicht schon im Objektinspektor eingeben (zum Property *Cells* siehe Kapitel 1.8.4):

```
procedure TUhrFormular.FormCreate(Sender: TObject);
begin
  AlarmList.Cells[0,0] := 'Alarmzeit';
  AlarmList.Cells[1,0] := 'Alarmhinweis';
  SetSmallSize;
end;
```

Dynamisches Aktivieren und Deaktivieren

Ein kleines Problem gibt es noch mit den bisherigen Auf- und Zuklappmethoden: Steuerelemente, die nicht sichtbar sind, werden von der VCL nicht automatisch deaktiviert. Wenn der Benutzer der Uhr also im verkleinerten Formularzustand mit Tab durch die Steuerelemente wechseln will, gelangt er nach dem Aufklappenschalter zum Eingabefeld, das gar nicht sichtbar ist. Die Methode *SetSmallSize* sollte also zusätzlich die drei unsichtbar werdenden Komponenten deaktivieren, damit diese mit der Tastatur nicht mehr ansteuerbar sind. Hierzu verwendet *SetBigSize* das *Enabled*-Property, das Sie bei jeder visuellen Steuerelement-Komponente finden:

```
AlarmList.Enabled := True;
AlarmEdit.Enabled := True;
AlarmActive.Enabled := True;
```

Die Methode *SetSmallSize* setzt entsprechend jeweils den Wert *False*, ein Listing erübrigt sich also.

Hinweis: In der Voreinstellung schiebt ein Formular die mit Tab angesteuerten Elemente automatisch in den Sichtbereich (Auslöser dafür ist weniger die Taste Tab als die Tatsache, dass die Steuerelemente den Tastaturfokus erhalten). Für diese Scroll-Automatik verantwortlich sind die Properties des Formulars *VertScrollBar.Visible* und *HorzScrollBar.Visible* (per Voreinstellung *True*) sowie *AutoScroll* (per Voreinstellung ebenfalls *True*); sie sind im Beispielprogramm ausgeschaltet.

1.8.3 Optimierung der Weckfunktion

Wie schon in Kapitel 1.5.5 festgestellt, kann der Timer, besonders wenn er auf grobe Sekundenschritte eingestellt ist, schon mal eine Sekunde übersehen, was in der bisherigen Programmversion ein Übersehen der Alarmzeit zur Folge gehabt hätte. Versuchen Sie beispielsweise in der alten Version, kurz bevor die Alarmzeit erreicht wird, den Rahmen des Fensters mit der Maus zu bewegen und lassen Sie ihn so lange nicht los, bis die Alarmsekunde vorüber ist. Auf diese Weise bringen Sie die bisherige Version der Uhr schnell dazu, die Weckzeit zu verschlafen.

Die neue Funktion muss also vergleichen, ob die aktuelle Zeit später ist als die Weckzeit, ob also ein »Größer-gleich« -Verhältnis besteht. Eine Möglichkeit dazu wäre, das Gleichheitszeichen in der bisherigen Methode durch ein »>=« zu ersetzen. (Dadurch würde zwar nur die alphabetische Reihenfolge der beiden Zeiten verglichen, was aber auch zum Erfolg führen würde.)

Die neue Methode soll auch eine Möglichkeit haben die Eingabe zu überprüfen und kehrt daher vom Vergleichen der Zeichenketten ab. Statt dessen verwendet sie die Funktion *StrToTime*, die die Zeitangabe des Benutzers in eine *TDateTime*-Angabe umwandelt. Intern handelt es sich bei *TDateTime* wie schon gesagt um einen Zahlenwert, daher können Sie diesen mit Hilfe des »>« -Operators mit der aktuellen Zeit (zurückgeliefert von der Funktion *Time*) vergleichen. Die Methode *IsTimeOver* wird also wie folgt geändert:

```
function TUhrFormular.IsTimeOver(Alarm: string): Boolean;
var
  AlarmTime: TDateTime;
begin
  Result := False;
  { Benutzereingabe nur überprüfen, wenn sie
    nicht leer ist: }
```

```
    if Alarm <> '' then begin
      AlarmTime := StrToTime(Alarm);
      Result := Time > AlarmTime;
    end;
  end;
```

Wenn der String fehlerhaft ist, löst *StrToTime* eine Ausnahmebedingung aus, die wir aber erst in der Methode *TimerTimer* abfangen werden (siehe Kapitel 1.8.5).

1.8.4 Programmierung der Stringtabelle

Um dem Benutzer mehrere Alarmzeiten zum Editieren zur Verfügung zu stellen, gibt es viele Möglichkeiten. So könnten die Weckzeiten in einer Liste dargestellt werden und einzelne Listenelemente könnten über Schalter gelöscht und hinzugefügt werden (so wird beispielsweise Delphis Property-Editor für das Property *TOpenDialog.Filter* bedient). Das Programm könnte die Liste dann nach jeder Veränderung sortieren, und der Benutzer könnte sich jederzeit an einer ordentlichen Darstellung erfreuen.

Die TStringGrid-Komponente

Da das Beispielprogramm in diesem ersten Kapitel jedoch sehr einfach sein soll, verwenden wir als Alternative den direkten Editiermodus der Komponente *StringGrid*. Das *StringGrid* basiert im Gegensatz zu den anderen bisher verwendeten Komponenten nicht auf einem Standardelement von Windows, sondern wurde größtenteils durch Object-Pascal-Code innerhalb der VCL realisiert. An ihr lässt sich bereits gut der Umgang mit anderen Komponenten üben, deren Leistungsmerkmale ebenfalls allein in den Händen ihrer Entwickler liegen, wie z.B. viele andere mit Delphi mitgelieferte Komponenten oder Komponenten von anderen Anbietern. Falls die Komponente der Philosophie von Delphi folgt, und das ist bei *StringGrid* der Fall, sind ihre wesentlichen Merkmale über Properties ansprechbar. Diese müssen jedoch nicht immer im Objektinspektor zu finden sein, sondern können auch für die Laufzeit des Programms reserviert sein.

Das Wichtigste an *StringGrid* ist sicherlich ihr Inhalt, der in Form einer (nicht-visuellen) Stringliste vorliegt – und zwar im Property *Cells*. Dieses ist bereits ein Beispiel für ein Property, das nur zur Laufzeit verwendbar sind. Das heißt, dass Sie zur Entwurfszeit noch keinen Text in die Tabelle eintragen können, wie das z.B. bei Editierfeldern oder den normalen Windows-Listen (*ListBox*-Komponenten) der Fall ist.

Das Property Cells

Die gezeigte Methode *FormCreate* verwendet bereits das Property *Cells* und verleiht den beiden Spalten der Tabelle Überschriften. Die erste Spalte soll zur Aufnahme der Weckzeit dienen, in der zweiten Spalte kann der Benutzer einen Text eingeben, der beim »Wecken« angezeigt wird.

Bei *Cells* handelt es sich um ein nach Spalten und Zeilen indiziertes Property, wobei die Zählung in beiden Dimensionen bei 0 beginnt. Um also im Programm alle Einträge der ersten Spalte abfragen zu können, benötigen wir eine Zählvariable *i* und den Ausdruck

```
AlarmList.Cells[0, i]
```

um auf die einzelnen Zeilen der nullten Spalte (die Spalte ganz links) zugreifen zu können. Die folgende Schleife überprüft alle Alarmzeiten der linken Spalte mit der Funktion *IsTimeOver* (das vollständige Listing folgt später):

```
{ Cells[0, 0] enthält die Beschriftung
  -> Cells [0, 1] enthält den ersten Alarm: }
for i := 1 to AlarmList.RowCount-1 do
  if IsTimeOver(AlarmList.Cells[0, i]) then begin
    ...
```

Auf diese Art können Sie die Zellen auch aus dem Programm heraus neu setzen. Das Beispielprogramm macht von dieser Möglichkeit Gebrauch, um Felder, deren Alarm vorüber ist, zu löschen (mit der im letzten Abschnitt eingeführten *IsTimeOver*-Methode würde der Alarm sonst jede Sekunde wiederholt werden):

```
AlarmList.Cells[0, i] := '';
```

Das Beispielprogramm muss sich noch etwas mehr um die Liste kümmern. Delphis VCL aktualisiert das *Cells*-Property auch während des Editierens ständig, also nicht erst nachdem der Benutzer mit ↵ bestätigt hat. Da die Listeneinträge jede Sekunde neu abgefragt werden, der Benutzer die Weckzeit jedoch zwischen zwei Sekundenschlägen kaum vollständig eingeben kann, würde so eine unfertige Weckzeit abgefragt werden, was schließlich zu einem Fehler führen würde.

StringGrid-Ereignisse

Es gilt also, die Komponente auf Einflussmöglichkeiten zu untersuchen, mit denen sich dieser Fehler verhindern lässt. Hierzu gibt es mindestens zwei Möglichkeiten. Zunächst die Möglichkeit, die im Beispielprogramm realisiert ist: Sie besteht einfach darin, dass die Alarmfunktion jedes Mal, wenn der Benutzer eine Taste zum Editieren der Tabelle drückt, also bei jedem *OnKeyPress*-Ereignis der *StringGrid*-Komponente, deaktiviert wird:

```
procedure TUhrFormular.AlarmListKeyPress
          (Sender: TObject; var Key: Char);
```

```
begin
  AlarmActive.Checked := False;
end;
```

Sobald der Benutzer also eine Weckzeit einzugeben beginnt, hört das Programm auf, die Weckzeit zu überprüfen (das in der obigen Methode auf *False* gesetzte Property *AlarmActive.Checked* bewirkt zunächst nur, dass der Markierungsschalter ausgeschaltet wird; damit die Alarmfunktion auch abgeschaltet wird, muss *Checked* in der noch folgenden Methode *TimerTimer* abgefragt werden). Der Benutzer muss die Alarmfunktion manuell wieder einschalten, sobald er mit dem Editieren fertig ist.

Ausblick auf eine kompliziertere Möglichkeit

Eine zweite und bessere Möglichkeit wäre es, nur das Feld von der Alarmprüfung auszuschließen, das gerade editiert wird. Dazu könnten Sie das Ereignis *OnSelectCell* der *StringGrid*-Komponente abfragen, das jedes Mal aufgerufen wird, wenn der Benutzer in eine andere Zelle gelangt, oder Sie könnten direkt die Properties *Col* und *Row* abfragen, um die aktuelle Zelle herauszufinden. Auf diese Weise ließe sich die aktuelle Zelle leicht vom Zeitvergleich ausschließen.

Zusätzlich müssten Sie dann jedoch feststellen, wann der Benutzer die Tabelle verlässt, denn sonst wäre immer irgendeine Zelle vom Zeitvergleich ausgeschlossen, auch wenn der Benutzer längst in einem völlig anderen Fenster arbeitet. Die benötigten Ereignisse wären in diesem Fall *OnExit* für die Stringtabelle (tritt auf, wenn der Benutzer innerhalb des Dialogs zu einem anderen Steuerelement wechselt) und *OnDeactivate* für das Formular (tritt auf, wenn ein anderes Fenster als das Formular aktiviert wird).

Diese Lösung wäre zwar komplizierter, aber sicher benutzerfreundlicher, weil der Benutzer die Alarmfunktion nicht jedes Mal manuell anschalten müsste. Da aber unser Beispielprogramm die Weckfunktion sowieso ab und zu ausschaltet, wie sich in Kapitel 1.8.5 noch herausstellen wird, bleibt die erstgenannte Lösung umso bequemer.

Weiteres zur Stringtabelle

Die Stringtabelle bietet viele weitere interessante Möglichkeiten. So können Sie z.B. durch Einschalten der Flags im *Options*-Property *goRowMoving* und *goColMoving* bewirken, dass der Benutzer die Zeilen bzw. Spalten der Tabelle mit der Maus verschieben kann, indem er eine der festen grauen Zellen verschiebt. Mit dieser Funktion macht die *StringGrid*-Komponente eine Vorlage für die *DBGrid*-Komponente, die eine Datenbanktabelle darstellt, und bei einer solchen kann es besonders sinnvoll sein, einzelne Spalten umzusortieren.

Die Uhr-Anwendung würde jedoch verwirrt werden, würde der Benutzer die Spalten zur Laufzeit vertauschen, denn sie setzt voraus, dass sich die Alarmzeit in der ersten Spalte befindet. Um Spalten oder Zeilen in Ihren Programmen umsortieren zu lassen,

müssen Sie wahrscheinlich auch die Ereignisse *OnColumnMoved* und *OnRowMoved* der *StringGrid*-Komponente bearbeiten, damit Ihr Programm von den Änderungen des Benutzers erfährt.

Für weitere Eigenschaften von *TStringGrid* muss an dieser Stelle auf die Online-Hilfe verwiesen werden. Mit *TStringGrid* verwandt ist die Komponente *TDrawGrid*; sie wird im Grafikbeispiel von Kapitel 4.4.3 verwendet.

1.8.5 Ausnahmebehandlung in der Timer-Methode

Die folgende, zur Abwechslung einmal etwas längere Methode leitet den Abschluss des *Wecker2*-Projektes ein:

```
procedure TUhrFormular.TimerTimer(Sender: TObject);
var
  i: Byte;
  FoundAlarm: string;
begin
  TimeText.Caption := TimeToStr(Time);
  try
    if AlarmActive.Checked then begin
      if IsTimeOver(AlarmEdit.Text) then begin
        FoundAlarm := AlarmEdit.Text;
        AlarmEdit.Text := '';
        AlarmMessage('Primärer Alarm: '+FoundAlarm);
      end;
      for i := 1 to AlarmList.RowCount-1 do
        if IsTimeOver(AlarmList.Cells[0, i]) then begin
          FoundAlarm := AlarmList.Cells[0, i];
          AlarmList.Cells[0, i] := '';
          FoundAlarm := 'Alarmzeit: '+FoundAlarm+
                        ' ('+AlarmList.Cells[1, i]+')';
          AlarmMessage(FoundAlarm);
        end;
    end;
  except
    on E: EConvertError do begin
      AlarmActive.Checked := False;
      MessageDlg('Ungültige Eingabe, bitte korrigieren:'#13#10
                 + E.Message, mtError, [mbOK], 0);
    end;
  end;
end;
```

Die erste Zeile ist Ihnen bereits aus der alten Programmversion bekannt. Da der Rest der Methode stark geschachtelt ist, müssen wir sie quasi von außen beginnend schälen.

Die äußerste Ebene bildet die Exception-Behandlung in einem *try...except...end*-Block. Wie die Exceptions allgemein funktionieren, lesen Sie in Kapitel 2.6. In diesem Fall geht es darum, die *EConvertError*-Exception zu behandeln, die dann auftritt, wenn *StrToTime* mit einer ungültigen Zeitangabe aufgerufen wird. Falls das der Fall ist, schaltet die Methode die Weckfunktion aus, indem sie einfach die Markierung im entsprechenden Markierungsschalter löscht (*AlarmActive.Checked := False*). Daraufhin fordert sie den Benutzer auf, die ungültige Eingabe zu korrigieren. Um statt des normalen Informationsfensters ein Meldungsfenster mit einem Fehlersymbol zu erhalten, verwendet sie nicht die schon bekannte Funktion *ShowMessage*, sondern die Funktion *MessageDlg* und gibt im zweiten Parameter den Namen des Fehlersymbols an (*mtError*)[3]. Die Zeichenfolge »#13#10« steht für einen Zeilenumbruch und bewirkt, dass die Meldung in zwei Zeilen ausgegeben wird.

Während diese Fehlermeldung sich nun auf dem Bildschirm befindet, treten wahrscheinlich schon weitere *OnTimer*-Events auf. Daher ist es wichtig, dass die Weckfunktion schon *vor* dem Aufruf von *MessageDlg* abgeschaltet wird, um so neue Meldungsfenster zu verhindern.

Die zweite Ebene von außen ist eine *if*-Abfrage, die testet, ob der Markierungsschalter gewählt ist. Ist er das nicht, so soll die Weckfunktion ausgeschaltet sein, und die Anweisungen der *if*-Abfrage werden übersprungen.

Im Innern der Funktion befinden sich zwei Blöcke, von denen der erste das einzelne Editierfeld und der zweite die Einträge der Alarmliste abfragt:

```
for i := 1 to AlarmList.RowCount-1 do
  if IsTimeOver(AlarmList.Cells[0, i]) then begin
    FoundAlarm := AlarmList.Cells[0, i];
    AlarmList.Cells[0, i] := '';
    FoundAlarm := 'Alarmzeit: '+FoundAlarm+' ('+
                  AlarmList.Cells[1, i]+')';
    AlarmMessage(FoundAlarm);
  end;
```

Wie oben besprochen, iteriert diese Schleife über alle Einträge der Alarmliste und ruft für jeden die bekannte Methode *IsTimeOver* auf. Falls diese mit dem Ergebnis *True* anzeigt, dass die Zeit überschritten ist, wird das Alarmfeld gelöscht, damit nicht eine Sekunde später erneut Alarm gegeben wird. Vorher muss die Alarmzeit in *FoundAlarm* zwischengespeichert werden, damit sie nachher von *AlarmMessage* in der Dialogbox angezeigt werden kann.

3 In den eckigen Klammern des dritten Parameters wird eine Liste der Schalter angegeben, über die das Meldungsfenster verfügen soll. Zu den zur Verfügung stehenden Optionen für den zweiten und dritten Parameter siehe die Online-Hilfe.

1.8.6 Behandlung mehrerer Ereignisse mit einer Methode

Inzwischen kann der Benutzer relativ entspannt seine Weckzeiten in der Liste eintragen, muss aber im einzelnen Editierfeld nach wie vor selbst darauf achten, dass der Alarm während des Editierens ausgeschaltet ist, weil auch dieses sonst jede Sekunde von der Methode *TimerTimer* überprüft wird. Zur Lösung dieses Problems könnte das Editierfeld eigentlich aus dem Uhr-Formular gelöscht werden, da ja eine Alarmliste schon genügt. Wir lassen es jedoch hier im Formular, denn es eignet sich gut zur Demonstration.

Um Fehlalarme wie bei der Alarmliste zu vermeiden, müssten wir auch bei jeder Eingabe in das Editierfeld den Alarm ausschalten, genauso wie bei jeder Eingabe in die Stringtabelle. Hierzu würde wieder die Zeile

```
AlarmActive.Checked := False;
```

genügen, sofern sie mit dem *OnKeyPress*-Ereignis des Editierfeldes verknüpft wäre. Da wir mit *AlarmListKeyPress* schon eine Methode haben, die genau diese Zeile ausführt, brauchen wir sie nicht noch einmal zu schreiben, sondern wir verknüpfen einfach *AlarmListKeyPress* mit dem *OnKeyPress*-Ereignis des Editierfeldes.

Wählen Sie dazu das *OnKeyPress*-Ereignis des Editierfeldes im Objektinspektor und drücken Sie neben dem freien Feld, in das Sie bisher immer doppelgeklickt haben, auf den Schalter zum Aufklappen einer Liste. Die Liste zeigt alle zu diesem Event passenden Methoden des Formulars an, die Sie schon geschrieben haben. Wählen Sie daraus *AlarmListKeyPress* aus, und fertig ist die Verknüpfung. In den meisten anderen Fällen sparen Sie auf diese Weise natürlich mehr als eine Zeile.

Vorschau auf die Unterscheidung mehrerer Komponenten in einer Methode

Damit Sie in einer Methode, die mit mehreren Ereignissen verknüpft ist, zwischen den unterschiedlichen Komponenten unterscheiden können, hat jedes von der VCL definierte Ereignis (außer einigen Ereignissen der Klasse *TApplication*) den Parameter *Sender: TObject*. Dieser bezeichnet die Komponente, von der das Ereignis ausgelöst wurde. *Sender* ist auch schon das erste Beispiel für ein polymorphes Objekt (siehe Kapitel 2), denn obwohl sein Typ als *TObject* deklariert ist, kann es sich bei *Sender* um eine Komponente der verschiedensten Typen handeln.

Eine sehr häufige Verwendung von *Sender* ist die Abfrage von *Tag*-Properties, wie sie in Kapitel 3.5.2 demonstriert wird. Die einfachste Möglichkeit, *Sender* zu verwenden, besteht jedoch in einem direkten Vergleich mit den Komponenten des Formulars:

```
if Sender = Button1 then ...
else if Sender = Button2 then ...
...
```

1.8.7 Nachwort/Zusammenfassung des Beispielprogramms

Das Beispielprogramm legt sicher einige untypische Verhaltensweisen an den Tag, wobei hier vor allem die automatische Deaktivierung der Weckfunktion, sobald der Benutzer ein Zeichen eingibt, auffällt. Die sicherste Lösung wäre hier eine Dialogbox, in der neue Weckzeiten eingegeben und überprüft werden, bevor sie zur Liste hinzugefügt werden.

So setzt das Beispielprogramm also weniger neue Maßstäbe für Benutzerfreundlichkeit (obwohl es sich an verschiedenen Stellen sehr darum bemüht), verhält sich aber in den folgenden Bereichen so, dass von einer Nachahmung nicht abgeraten zu werden braucht: Benutzung der Timer, Einbinden einer Dialogbox, Abfangen von Tastaturereignissen ohne dadurch die Eingabe des Benutzers zu stören, Behandeln von Exceptions. Der Umgang mit dem Markierungsfeld verdient allerdings noch eine Bemerkung:

Benutzung von Steuerelementen als Datenspeicher

Normalerweise verfügt ein Programm mit abschaltbarer Weckfunktion über eine Variable, die speichert, ob die Weckfunktion gerade an- oder abgeschaltet ist. Dieses Beispielprogramm missbraucht gewissermaßen den Markierungsschalter, genauer: dessen Property *Checked* als Datenspeicher. Dies ist in Object Pascal kein fauler Trick, sondern eine sichere Vorgehensweise, da das Wesen der Properties dafür sorgt, dass zwischen dem Wert dieses Properties und der Markierung auf dem Bildschirm immer Übereinstimmung herrscht. Voraussetzung dafür ist natürlich, dass die entsprechende Komponente das Property richtig verwaltet.

Außerdem lässt sich diese Methode nur dann anwenden, wenn das Property die gesamte Laufzeit des Programms gültig ist. Wenn beispielsweise eine Dialogbox des Programms nur für die Dauer ihrer Sichtbarkeit existiert (siehe dynamische Konstruktion von Formularen in Kapitel 3.4.4), können ihre Properties auch nur dann gelesen werden, denn nach der Freigabe des Dialogbox-Objekts werden ihre Werte verworfen.

1.9 Das aktuelle Einmaleins der Komponenten

In den folgenden Abschnitten geht es nicht mehr um grundlegende Abläufe in der Delphi-IDE, sondern um konkrete Komponenten: Eingebunden in eine Übersicht über die Komponentenpalette finden Sie hier eine Reihe weiterer einfacher Beispiele zur Verwendung und Programmierung von Komponenten in Delphi. Die Übersicht orientiert sich an der Funktion der Komponenten und nicht an der eher historisch bedingten

Aufteilung von Delphis Komponentenpalette. So finden Sie beispielsweise in Kapitel 1.9.4 eine Übersicht über alle Eingabekomponenten von Delphi, egal ob sie sich nun auf der Seite *Standard*, *Zusätzlich*, *Win3.1* oder *Win32* befinden.

Wenn Sie das »Einmaleins« der Komponenten beherrschen, können Sie in Kapitel 3.1 bis 3.4 eine fundierte Einführung in die Grundlagen und Gemeinsamkeiten aller Komponenten mitsamt der dahinter liegenden Klassenhierarchie der VCL lesen.

Die neue Beispielprogramm-Serie

Die Beispielprogramme in den folgenden Abschnitten bauen auf dem bisherigen Beispiel auf und sind auf der CD als *Wecker3a* bis *Wecker3e* zu finden. Diese Versionen weisen alle einen wesentlichen Vorteil gegenüber *Wecker2* auf: Sie verwenden zur Eingabe der Alarmzeit ein eigenes (zweites) Formular. Für den Benutzer bedeutet das zwar, dass er zuerst eine Dialogbox aufrufen muss, anstatt die Alarmzeit direkt einzugeben. Wenn er den Dialog aber über ein Tastenkürzel auf der Tastatur aufrufen kann, so wird er mit Dialog immer noch viel schneller an sein Ziel kommen als mit der bisherigen Programmversion, in der er zuerst eine neue Zelle in der Tabelle ansteuern und die Alarmfunktion am Schluss wieder manuell anschalten musste.

Für die Architektur des Programms ist diese Umstellung in jedem Fall nützlich, denn durch die eigene Dialogbox kann das Programm die Eingabe der Zeit so gut kontrollieren, dass die Umstände mit dem Ein- und Ausschalten der Alarmfunktion und der Exception-Behandlung aus Kapitel 1.8 schlicht und einfach wegfallen[4].

1.9.1 Verwendung von Formularen als modale Dialoge

Da aus den oben erläuterten Gründen alle folgenden Versionen des Beispielprogramms mit einem Dialog zur Eingabe der Alarmzeiten arbeiten, nutzen wir zunächst diese Gelegenheit, um uns etwas weiter mit der wichtigsten Komponente eines Delphi-Programms zu beschäftigen – mit dem Formular (das zwar nicht in der Komponentenpalette auftaucht, aber in der Vererbungshierarchie der VCL auch als Komponente angesehen wird). Ab Kapitel 1.9.2 geht es dann mit den »normalen« Komponenten weiter.

Wir fangen erst einmal wieder klein an und streichen die Liste der Alarme aus dem Formular von *Wecker2*. Es bleiben noch das Textelement (*TLabel*) für die Uhrzeit, der Schalter für den Schriftwahldialog, die Dialogkomponente selbst und natürlich der Timer. Für den Aufruf des Dialogs kommt ein neuer Schalter hinzu. Abbildung 1.21 zeigt die beiden Formulare dieser Programmversion, auf der CD zu finden unter dem Namen *Wecker3a*.

4 Da Exceptions in Timer-Methoden immer noch zu einer Flut an Meldungsfenstern führen würden, wurde eine vereinfachte Exception-Behandlung zur Sicherheit noch im Programm belassen.

Das Dialogformular *R32*

Die Erzeugung des neuen Formulars kann kurz zusammengefasst werden, da hierbei im Wesentlichen nur Grundlagen zusammengefasst werden, die schon in vorangegangenen Kapiteln erläutert wurden:

Ein neues Formular erzeugen Sie über das Datei-Menü der Delphi-IDE. Wenn Ihnen ein leeres Formular ausreicht, wählen Sie einfach den Punkt DATEI | NEUES FORMULAR. Zur Erzeugung eines Formulars auf Basis einer Vorlage oder mit Hilfe eines Experten verwenden Sie den Punkt DATEI | NEU... (siehe Kapitel 1.6.4). In beiden Fällen erhält Ihr Projekt zwei neue Dateien: `form?.dfm` und `form?.pas` (zum Aufbau von Projekten siehe Kapitel 1.6.3). Letztere enthält den Object-Pascal-Code zum Formular und hat nach ihrer Erzeugung den in Kapitel 1.5.3 beschriebenen Standard-Aufbau.

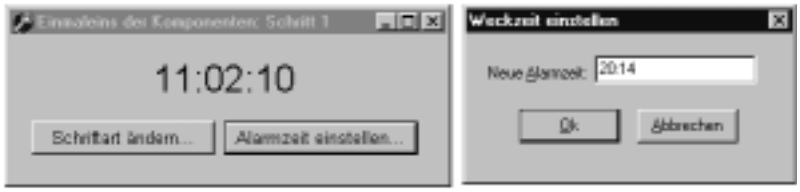


Abbildung 1.21: Nach dieser Aufteilung in zwei Formulare wird sich das Programm erheblich einfacher erweitern lassen.

Wie die Abbildung zeigt, erhält das neue Formular zunächst ein Editierfeld mit vorangehender Beschriftung sowie zwei Aktionsschalter. Um das neue Formular zu einem funktionsfähigen Dialog zu machen, werden die folgenden Properties gesetzt:

Property-Einstellung	**Wirkung**
Formular: BorderStyle = bsDialog	Zur Laufzeit erhält das Fenster einen nichtverstellbaren Dialograhmen.
Formular: Position = poScreenCenter	Das Fenster wird zur Laufzeit in der Mitte des Bildschirms angezeigt.[5]
Formular: Name = AlarmEditDlg	Bessere Lesbarkeit des Quelltextes
OK-Schalter: ModalResult = mrOK	Das Drücken dieses Schalters bewirkt das Schließen des Fensters bei Rückgabe des Wertes *mrOK* (wird im Programm abgefragt).
Abbruch-Schalter: ModalResult = mrCancel	Analog zum OK-Schalter ist das Ergebnis hier *mrCancel*.

5 Eine noch bessere Einstellung ist möglicherweise *poMainFormCenter*.

Property-Einstellung	**Wirkung**
OK-Schalter: Default = True	Vereinfacht die Tastaturbedienung: Der Schalter ist als Vorgabeschalter per Eingabetaste aktivierbar, und zwar unabhängig davon, welches Dialogelement gerade aktiv ist (mehrzeilige Eingabefelder fangen die Eingabetaste allerdings per Voreinstellung ab).
Abbruch-Schalter: Cancel = True	Ebenso eine Vereinfachung der Tastaturbedienung: Macht den Schalter zum Schalter, der bei Betätigung von ESC automatisch ausgelöst wird.

Komponentenschablonen *R9*

Die oben gezeigte Kombination aus OK- und Cancel-Schalter wäre ein ideales Beispiel für eine Komponentenschablone. Markieren Sie die beiden Schalter im Formular-Designer und rufen Sie aus dem lokalen Popup-Menü den Punkt DER OBJEKTABLAGE HINZUFÜGEN... auf, und Sie sparen sich in Zukunft mit zwei Mausklicks die Einstellung von sechs verschiedenen Properties (obige vier plus die beiden *Caption*-Properties mit Unterstreichung der ersten beiden Buchstaben!).

Modale Anzeige des Formulars

Die folgende Methode ist zur Bearbeitung des *OnClick*-Ereignisses des Schalters ALARMZEIT EINSTELLEN... des Hauptformulars gedacht:

```
procedure TAlarmForm.Button2Click(Sender: TObject);
begin
  if AlarmEditDlg.ShowModal = mrOk then begin
    AlarmAktiviert := true;
    Alarmzeit := AlarmEditDlg.Edit1.Text;
  end;
end;
```

AlarmEditDlg ist der in der obigen Tabelle festgelegte Name für das von Delphi automatisch angelegte Formular-Objekt (deklariert in der Unit des Dialog-Formulars). Um in der Hauptformular-Unit darauf zugreifen zu können, muss die Unit des zweiten Formulars in die Unit des Hauptformulars eingebunden, also in der *uses*-Zeile aufgeführt werden. Sie brauchen das jedoch nicht manuell zu tun, denn Delphi fragt Sie beim Versuch, das Programm einfach so zu starten, ob es diese Änderung der *uses*-Klausel für Sie durchführen soll.

Die Methode *ShowModal* bewirkt, dass unser Zweitformular (welches wie alle Nicht-Hauptformulare standardmäßig unsichtbar ist, da *Visible=false*) modal angezeigt wird. Modal bedeutet, dass der Benutzer in dieser Anwendung nur mit diesem Fenster arbeiten kann. Alle anderen Fenster in derselben Anwendung werden erst dann wie-

der zugänglich, wenn er das modale Fenster schließt[6]. Und da wir oben die *ModalResult*-Properties der beiden Schalter entsprechend gesetzt haben, brauchen wir uns bei der Programmierung um dieses Schließen nicht zu kümmern (wir müssen also nicht *AlarmEditDlg.Close* aufrufen).

ShowModal gibt den *ModalResult*-Wert zurück, der zum Schließen des Fensters geführt hat (bzw. *mrNone*, falls wir das Fenster durch Aufruf der *Close*-Methode geschlossen hätten). Da die gerade eingestellte *Alarmzeit* nur bei Verwendung des OK-Schalters geändert werden soll, muss das Ergebnis von *ShowModal* wie oben gezeigt abgefragt werden. Falls *ModalResult mrOk* zurückliefert, wird außerdem die Alarmfunktion eingeschaltet. *AlarmAktiviert* und *Alarmzeit* sind zwei Variablen des Hauptformulars, die beim *OnCreate*-Ereignis wie folgt initialisiert werden:

```
procedure TAlarmForm.FormCreate(Sender: TObject);
begin
  AlarmAktiviert := true;
  Alarmzeit := '';
  TimerTimer(nil); (* Hier können wir auch gleich dafür sorgen,
    dass die Zeit SOFORT angezeigt wird und nicht erst nach dem
    ersten OnTimer-Ereignis. *)
end;
```

Überprüfen der Dialogeingaben *R26*

Ein wesentlicher Vorteil des Dialogfensters im Beispielprogramm sollte ja sein, dass das Programm die Alarmzeiten leichter kontrollieren kann. Und zwar genügt es anstelle der komplizierten Mechanismen aus Kapitel 1.8, wenn jede Zeitangabe nur einmal genau dann überprüft wird, wenn der Benutzer den Dialog mit OK schließen will. Das Programm soll nun, falls die Zeitangabe nicht regelgemäß ist, eine Meldung anzeigen und dem Benutzer die Gelegenheit geben, die Eingabe zu korrigieren oder ABBRUCH zu drücken.

Ein Fenster zu schließen ist einfach, hierfür gibt es die Methode *Close*. Um aber einen Weg zu finden, wie wir die automatische Schließung des Fensters verhindern können, müssen wir uns noch einmal mit dem *ModalResult*-Property befassen. Nicht nur die Aktionsschalter verfügen über ein solches Property, sondern auch das Formular selbst. Wird das *ModalResult*-Property *des Formulars* auf einen anderen Wert als *mrNone* gesetzt, führt das dazu, dass das Formular geschlossen wird und *ShowModal* das entsprechende Ergebnis zurückliefert.

6 Es ist jedoch möglich, ein zweites modales Fenster zu öffnen. Dann tritt auch das erste modale Fenster in den Wartezustand.

Wenn ein Aktionsschalter betätigt wird, setzt die VCL den *ModalResult*-Wert des Formulars auf den des Schalters. Dies ist der tiefere Grund dafür, warum der Dialog automatisch geschlossen wird, wenn wir die *ModalResult*-Properties von OK und ABBRUCH setzen.

Um eine Eingabeüberprüfung zu implementieren, haben wir nun zwei Möglichkeiten:

- Wir setzen das *ModalResult*-Property des Formulars manuell zur Laufzeit. Die Einstellung des *ModalResult*-Properties der beiden Buttons zur Entwurfszeit wäre dann unnötig.
- Wir stellen das Property im Objektinspektor auf *mrOk* ein, verhindern aber, wenn die Eingabe des Benutzers nicht korrekt ist, dass auch das *ModalResult*-Property des Formulars gesetzt wird.

Für das Beispielprogramm wurde die zweite Möglichkeit gewählt. Drückt der Benutzer nun OK, so setzt die VCL das *ModalResult*-Property des Formulars auf *mrOk*. Die Schließung des Fensters findet aber erst statt, nachdem die *OnClick*-Methode aufgerufen wurde. Dies gibt uns Gelegenheit, innerhalb dieser Methode das *ModalResult*-Property im Fehlerfall wieder zurückzusetzen, so dass der Dialog doch nicht geschlossen wird:

```
procedure TAlarmEditDlg.Button1Click(Sender: TObject);
var
  TestAlarmTime : TDateTime;
begin
  try
    if Edit1.Text <> '' then // nur prüfen, wenn keine Leereingabe
      TestAlarmTime := StrToTime(Edit1.Text);
  except
    on E : EConvertError do begin
      ShowMessage('Ungültige Eingabe:' + E.Message);
      ModalResult := mrNone;
    end;
  end;
end;
```

Weitere Aufgaben für den Dialog

Dies war erst die erste von vielen Methoden, die im weiteren Verlauf des Kapitels 1.9 zu *TAlarmEditDlg* hinzugefügt werden. Die Formularklasse eines Dialogs eignet sich nämlich hervorragend dazu, das Hauptformular zu entlasten:

- indem es eine Eingabeüberprüfung selbst durchführt, wie oben gezeigt.
- indem es dem Hauptformular eine aus mehreren Feldern bestehende Dialogeingabe in Form einer Object-Pascal-Datenstruktur zurückliefert.

- indem es dem Hauptformular erlaubt, auch mehrere Felder in einem Schritt durch Angabe einer solchen Datenstruktur zu initialisieren.

In der Programmversion *Wecker3d* in Kapitel 1.9.4 wird das Beispielprogramm erstmals mit einer solchen Datenstruktur arbeiten.

1.9.2 Komponenten zur Programmsteuerung

So wie die Funktionsvielfalt moderner Software ständig ansteigt, erweitert sich auch das Arsenal der Bedienungselemente, durch die diese Funktionen gestartet werden können. Zwar basieren diese Elemente auf nur zwei Grundbausteinen, den Schaltern und Menüs, aber zum einen gibt es davon verschiedene Variationen, zum anderen wächst mit zunehmender Zahl der Schalter auch der Bedarf an Komponenten, durch die sich die Schalterflut besser organisieren lässt.

In Delphis Komponentenpalette befinden sich für Menüs, Schalter und Schalter-Organisation sowie für weitere Steuerungsaufgaben die folgenden Komponenten:

TMainMenu stellt eine Menüleiste dar, die Sie zum Hauptmenü eines Formulars machen können, indem Sie sie in dessen *Menu*-Property angeben. In Kapitel 1.4.7 wurde bereits beschrieben, wie Sie eine solche Menüleiste in Delphis Menüeditor entwerfen können. Die nächste Version des Beispielprogramms demonstriert auch den praktischen Einsatz einer solchen Menüleiste.

Neu in Delphi 6 ist die *TActionMainMenuBar* (Seite ZUSÄTZLICH), die Sie anstelle einer *MainMenu*-Komponente verwenden können. Sie funktioniert nur mit Hilfe einer *ActionManager*-Komponente, in der Sie die *möglichen* Menübefehle definieren. Die Zusammenstellung des Menüs aus den Befehlen des Action-Managers ist daraufhin sehr einfach, Sie können es dem Benutzer sogar zur Laufzeit freistellen, die Zusammenstellung selbst zu ändern (siehe Kapitel 4.6.5).

Ein *TPopupMenu* ist ein Popup-Menü, wie Sie es auch in der Delphi-IDE überall durch Klicken der rechten Maustaste aufrufen können. Es wird ebenfalls im Menüeditor entworfen. Damit es zur Laufzeit vom Benutzer aufgerufen werden kann, müssen Sie lediglich festlegen, für welche Komponente des Formulars es gelten soll, wo der Benutzer also mit der rechten Maustaste klicken muss, um dieses Menü zu erhalten. Die Verbindung zwischen visueller Komponente und Popup-Menü geschieht über das *PopupMenu*-Property der visuellen Komponente. Mehr zum praktischen Einsatz von Popup-Menüs erfahren Sie in Kapitel 5.2.6.

TMenuItem ist die Klasse für einen einzelnen Menüpunkt. Sowohl *MainMenu*- als auch *PopupMenu*-Komponenten sind aus solchen *MenuItem*-Komponenten zusammengestellt. Die Properties von *TMenuItem* wurden bereits in Kapitel 1.4.7 beschrieben.

Die Klasse *TButton* stellt *den* Standard-Befehlsschalter zur Verfügung. Das bisherige Beispielprogramm ist nicht um das wichtigste Property (*Caption*) und das wichtigste Ereignis (*OnClick*) dieser Komponente herumgekommen. Und in Kapitel 1.9.1 sind uns auch die Properties *Default*, *ModalResult* und *Cancel* begegnet, die eng mit der Logik von modalen Dialogen verknüpft sind. Damit ist zu dieser Komponente bereits alles Wichtige gesagt.

Die Klasse *TBitBtn* auf der Seite ZUSÄTZLICH der Komponentenpalette ist von *TButton* abgeleitet und erweitert letztere durch die Möglichkeit, Bitmaps auf der Schaltfläche darzustellen. Hierzu verfügt sie über das Property *Glyph*, in das Sie eine Bitmap laden können, in der bis zu drei Bilder aneinander gereiht sind. Der Schalter stellt in jedem seiner drei möglichen Zustände eine dieser Bitmaps dar. Die drei Zustände sind der Normalzustand, der deaktivierte Zustand (*Enabled=False*) und der gedrückte Zustand. Für die weitere Gestaltung benötigen Sie auch die Properties *Layout*, *Margin*, *Spacing* und *NumGlyphs*, zu denen hier auf die Hilfedatei verwiesen sei. Wir benötigen *TBitBtn* nämlich in diesem Buch nur zur Anzeige der Standardbitmaps, aus denen Sie über das *Kind*-Property wählen können. Für selbst definierte Bitmaps verwenden die Beispielprogramme des Buchs Schalter der Klassen *TToolButton* und *TSpeedButton*.

TSpeedButton (Seite ZUSÄTZLICH) stellt die gleichen Properties für Bitmaps zur Verfügung, wie sie eben für *TBitBtn* genannt wurden. *TSpeedButton* kennt noch einen vierten Zustand, den Zustand, in dem der Schalter dauerhaft gedrückt ist (*Down=True*). Dieser gedrückte Zustand hängt mit der Fähigkeit dieser Schalter zusammen, in Gruppen aufzutreten, mit der sich Kapitel 5.2.1 genauer befassen wird.

TToolBar (Seite WIN32) stellt eine Werkzeug- bzw. Symbolleiste dar, die, im Gegensatz zu *TPanel*, ihre Kindelemente automatisch anordnet und die Icons effektiv in einer Bilderliste (*TImageList*) verwaltet.

TToolButton ist keine unabhängige Komponentenklasse, daher finden Sie sie auch nicht in der Komponentenpalette, sondern nur im lokalen Menü von *TToolBar*-Komponenten (NEUER SCHALTER). *TToolButton* bietet mehr Variationsmöglichkeiten als *TSpeedButton* und kann beispielsweise für Drop-Down-Schalter oder Textschalter verwendet werden.

TControlBar (ab Delphi Professional auf der Seite ZUSÄTZLICH; in Delphi Standard über Umwege verwendbar – siehe Hinweis in Kapitel 5.2.2) wurde in Delphi 4 eingeführt und stellt eine Möglichkeit zur Verwaltung mehrerer Symbolleisten der *TToolBar*-Klasse dar. Der Benutzer kann diese Leisten zur Laufzeit des Programms verschieben, umordnen und sogar abtrennen und wieder andocken. Allgemein kann eine *ControlBar* auch beliebige andere Typen von Steuerelementen aufnehmen, allerdings ist sie für kleinere Elemente kaum geeignet, da sie für jedes Element ein eigenes, relativ aufwändig gestaltetes »Band« anlegt.

TCoolBar (Seite WIN32) hat die gleichen grundsätzlichen Fähigkeiten wie *TControlBar*, erinnert aber doch im Vergleich mit *TControlBar* eher an eine Spielerei: Zwar können Sie einzelne Bänder auf eine feste Größe einstellen, so dass sich die Bänder nicht mehr versehentlich überlappen können, aber die feste Größe bewirkt gleich, dass das Band überhaupt nicht mehr verschoben werden kann. Zwar war Microsoft so gnädig, dieses im Internet Explorer verwendete Steuerelement über die Bibliothek *ComCtl32.Dll* der Öffentlichkeit zur Verfügung zu stellen, aber wie Sie an den Anwendungen des Microsoft Office erkennen können, zieht man es auch bei Microsoft normalerweise vor, feste, nicht versehentlich überlappbare, aber verschiebbare Toolbars zu verwenden. So macht es auch die Delphi-IDE unter Verwendung der eigenen Komponente *TControlBar* vor, die auch in diesem Buch im TreeDesigner zum Einsatz kommt.

TActionToolBar ist eine in Delphi 6 neu eingeführte Toolbar (ab der Professional-Ausgabe; Seite ZUSÄTZLICH), die Sie mit Befehlen eines Action Managers füllen können; sie wird meist in Verbindung mit einer *TActionMainMenuBar* verwendet; Details dazu lesen Sie in Kapitel 4.6.5.

Auch die unscheinbare *TLabel*-Komponente besitzt eine wichtige Steuerungsfunktion: Wenn Sie durch Voranstellung eines »&« ein Zeichen in der Beschriftung des Labels unterstreichen, dient dieses Zeichen (eventuell in Verbindung mit Alt) als Tastenkürzel zum Anspringen eines Eingabeelementes. Damit dies funktioniert, müssen Sie ein *TLabel* über das *FocusControl*-Property mit einer Eingabekomponente verknüpfen. Im Beispielprogramm wurde dies bereits demonstriert (Kapitel 1.4.5, *Rücksichtnahme auf die Tastaturbedienung*).

Vom Schalter zum Menü

Nachdem das Entwerfen von Menüs mit Hilfe des Menüeditors bereits in Kapitel 1.4.7 zur Sprache gekommen ist, sei hier nur kurz demonstriert, wie das Beispielprogramm sich der Menüs bedient. In der Programmversion *Wecker3b* (Abbildung 1.22) wurden

zunächst die beiden *TButtons* im Dialogformular durch *TBitBtn*-Komponenten mit verändertem *Kind*-Property ersetzt, was aber nur äußerlich eine Änderung bedeutet. Dann wurden die beiden bisher als Schalter zugänglichen Aktionen des Hauptformulars ins Hauptmenü verlegt. Wir haben es hier also nicht mehr mit *TButton*-, sondern mit *TMenuItem*-Komponenten zu tun, die im Formular nicht durch separate Bausteine repräsentiert werden. Um ihre Properties im Objektinspektor editieren zu können, müssen Sie den entsprechenden Menüpunkt im Menüeditor auswählen.

Wir benötigen hier zunächst das *OnClick*-Ereignis, für das wir *keinen* neuen Code zu schreiben brauchen, denn wir können die Methoden wiederverwenden, die bisher beim Betätigen der Schalter Anwendung gefunden haben. Beide Methoden können Sie im Objektinspektor beim *OnClick*-Ereignis aus der aufklappbaren Liste auswählen. Die Methode *Button1Click* wird so auf den Schriftwahl-Menüpunkt umgeleitet, *Button2Click* auf den Menüpunkt zur Einstellung der Alarmzeit, alles durch ein paar Mausklicks. (Allerdings würde sich jetzt eine Umbenennung der Methoden anbieten, was jedoch mit der Maus alleine nur auf eine sehr umständliche Weise möglich wäre.)

Abbildung 1.22: Wecker3b wurde mit einem Hauptmenü versehen.

Um das so gestaltete und mit Aktionen verknüpfte Menü auch anzuzeigen, muss außerdem das *Menu*-Property des Formulars auf die *TMainMenu*-Komponente gesetzt werden (dies geht durch bequemes Anklicken des Namens der *TMainMenu*-Komponente im Objektinspektor beim *Menu*-Property).

Hinweis: Ein weiterführendes Konzept zur Ausführung von Befehlen stellt das der Aktionslisten dar, das auch ohne die den Profi-Versionen von Delphi vorbehaltenen ActionManager-Komponenten schon in der Personal-Edition vertreten ist. Wenn Sie ein und dieselbe Aktion sowohl von einem Schalter als auch von einem Menüpunkt (oder irgendeiner anderen Steuerkomponente) aus starten wollen, bieten diese Aktionslisten eine erhebliche Vereinfachung (siehe Kapitel 4.6.4).

Markierte Menüpunkte

Das Beispielprogramm besitzt mit OPTIONEN | ALARMFUNKTION AKTIVIERT noch einen dritten Menüpunkt, der dem Markierungsschalter aus der früheren Programmversion entspricht, dessen *Checked*-Property schon in Kapitel 1.8.4 gesetzt wurde. Im Gegensatz

zu einem Markierungsschalter (*TCheckBox*) wird das *Checked*-Property eines Menüpunktes aber nicht automatisch umgeschaltet, wenn der Menüpunkt angeklickt wird. Daher muss dies manuell in der zum Menüpunkt gehörigen *OnClick*-Methode erledigt werden:

```
procedure TAlarmForm.Alarmaktiviert1Click(Sender: TObject);
begin
  Alarmaktiviert1.Checked := not Alarmaktiviert1.Checked;
end;
```

Da es in dieser Version des Beispielprogramms keine Variable *AlarmAktiviert* gibt, sondern bei Bedarf einfach das oben gesetzte *Checked*-Property abgefragt wird, sind keine weiteren Aktionen erforderlich.

1.9.3 Listenelemente und die Sicherung ihres Inhalts

Auch zur Darstellung von mehreren Datenelementen bietet die VCL eine Reihe von Alternativen:

TListBox ist quasi der Urahn aller Listboxen, eine Standard-Liste, die nicht nur für den Benutzer einfach handzuhaben ist, sondern auch für den Softwareentwickler – solange es nur darum geht, Texteinträge zu präsentieren und nicht etwa Grafiken.

TComboBox ist eine Kombination aus einer Listbox und einem einzeiligen Eingabefeld, es gibt sie in drei Variationen, die Sie im Property *Style* auswählen können: Im Stil *csSimple* wird die Liste fest unter dem Editierfeld verankert, im Stil *csDropDown* ist sie aufklappbar (die Größe im aufgeklappten Zustand wird dann durch *DropDownCount* festgelegt). Mit dem Stil *csDropDownList* erhalten Sie eine Liste wie mit *csDropDown*, jedoch ist das Editierfeld nicht editierbar, sondern kann nur Werte enthalten, die auch in der Liste vorkommen (daher die Betonung des Namens »csDropDown*List*«). In seinen weiteren Eigenschaften (auch Ereignissen) ist *TComboBox* eine Kombination aus *TEdit* und *TListBox*.

TComboBoxEx ist neu in Delphi 6 (Seite ZUSÄTZLICH), kann mit einer Bilderliste verknüpft werden und erlaubt über das Property *ItemsEx*, jedem Eintrag neben seinem Text verschiedene Bilderindizes sowie eine Einrückungstiefe zuzuordnen. Gegenüber *TComboBox* hinzugekommene Optionen können Sie in *StyleEx* einstellen.

TCheckListBox befindet sich ebenfalls auf der Seite ZUSÄTZLICH. Bei ihr ist jeder Listeneintrag mit einem Markierungsfeld versehen, wodurch dem Benutzer die Auswahl mehrerer Einträge besonders einfach gemacht wird. Auch der Programmierer ist wieder gut bedient, denn er kann den Markie-

rungsstatus aller Einträge einfach über das Array-Property *Checked* abfragen bzw. über das *State*-Array, wenn auch der dritte Zustand *cbGrayed* erlaubt ist (dieser wird in Kapitel 1.9.4 im Zusammenhang mit der *TCheckBox*-Komponente erläutert).

TColorBox (neu in Delphi 6, Seite ZUSÄTZLICH) ist eine ComboBox, aus der Sie eine Farbe (Property *Selected*) auswählen können wie bei den Farb-Properties im Objektinspektor. Die Auswahl der Farben können Sie grob über das Property *Style* einstellen (Standardfarben, erweiterte Palette und Systemfarben), wo auch eine alternative Beschriftungsweise *PrettyNames* zur Verfügung steht (dadurch wird dann z.B. aus »clCaptionText« der Name »Caption Text«).

TListView (Seite WIN32) stellt die mit Windows 95 eingeführte, moderne Variante einer Listbox dar. Wenn Sie eine Idee haben, wie Sie verschiedene Typen von Texteinträgen einer Listbox durch kleine grafische Symbole unterscheiden können, und wenn es vielleicht zu jedem Eintrag Detailinformationen gibt, die sich in einer Tabelle darstellen lassen, eignet sich diese Komponente hervorragend zur Realisierung. Ein erstes Anwendungsbeispiel finden Sie in Kapitel 1.9.5. In der Delphi-IDE finden Sie ein *ListView* beispielsweise im Dialog zur Konfiguration der Komponentenpalette.

TTreeView (Seite WIN32) ist sowohl von der internen Organisation als auch von seinem Äußeren her verwandt mit *TListView*, ist aber mit seinen Baumstrukturen eigentlich schon zu komplex, um noch als »Listenelement« gezählt werden zu können. Die Beispielprogramme dieses Buchs machen des Öfteren Gebrauch von dieser Komponente.

Für den Bereich der Datenbanken gibt es noch spezielle Variationen von Combo- und Listboxen: *TDBListBox*, *TDBComboBox*, *TDBLookupListBox* und *TDBLookupComboBox*, die an entsprechender Stelle in Kapitel 7 erläutert werden.

Diese vier neuen Delphi-6-Komponenten wurden von Borland auf der Seite BEISPIELE etwas versteckt – möglicherweise, weil sie (noch) nicht der üblichen Qualität von VCL-Komponenten entsprechen (so gibt es im Quelltext mehrere »Hacks« und To-Do-Einträge und es funktioniert auch noch nicht alles wie etwa die Abfrage von Detailinformationen). Ein Beispiel zur Verwendung dieser Komponenten finden Sie in Kapitel 8.4.1.

Die ersten drei Komponenten, *TShellTreeView*, *TShellListView* und *TShellComboBox*, arbeiten hervorragend in einem Formular zusammen, wenn Sie sie wie in Abbildung 1.23 gezeigt über Properties miteinander verknüpfen. (Sie müssen nur für die eine Richtung der Property-Verknüpfung sorgen: Wenn Sie etwa das Property *ShellTreeView* des ListViews auf das TreeView setzen, wird in umgekehrter Richtung das *ShellListView*-Property des TreeViews automatisch gesetzt. Dies bewirkt, dass der Inhalt des *ShellListView*s automatisch neu aufgebaut wird, wenn im *ShellTreeView* ein anderes Verzeichnis gewählt wird.)

Auf diese Weise erhalten Sie einen automatisch funktionierenden Dateisystem-Browser wie den Windows-Explorer. Die Einträge von ListView- und TreeView-Komponente verfügen sogar über das vom Windows-Explorer bekannte Popup-Menü, so dass Sie beispielsweise Dateien umbenennen oder löschen können.

Die vierte Komponente, *TShellChangeNotifier*, dient dazu, Sie per *OnChange*-Ereignis über verschiedene Änderungen eines Verzeichnisses (Property *Root*) und optional alle darin enthaltenen Verzeichnisse (Property *SubTree*) zu informieren. In *NotifyFilters* können Sie beispielsweise angeben, ob die Änderung von Dateinamen und Dateigrößen gemeldet werden soll.

Schließlich finden Sie auf der Seite *Win 3.1* der Komponentenpalette eine Reihe von speziellen Listenkomponenten, die den Bestandteilen der Dateiauswahldialoge aus der Zeit von Windows 3.1 entsprechen. Wenn Sie diese Komponenten wie in Abbildung 1.23 gezeigt über ihre Properties miteinander verknüpfen, bauen die Listen ihren Inhalt vollkommen automatisch auf (Sie müssen also keine *OnChange*-Ereignisse bearbeiten). So verbinden Sie beispielsweise durch Setzen des *TDriveComboBox*-Properties *DirList* auf eine *TDirectoryListBox*-Komponente diese beiden Komponenten, wodurch der Inhalt der Verzeichnisliste automatisch gewechselt wird, wenn der Benutzer ein anderes Laufwerk aus der *DriveComboBox* anwählt. Ähnlich wird bei einem Verzeichniswechsel der Inhalt der *TFileListBox* neu aufgebaut. Die *FileListBox* wird auch aktualisiert, wenn der Dateifilter einer *TFilterComboBox* geändert wird. Optional können aktuell gewähltes Verzeichnis und aktuell gewählte Datei in einem *Label*- bzw. *Edit*-Element angezeigt werden.

Kleine Anwendungsbeispiele für diese Komponenten finden Sie in den Beispielprogrammen *DynamicListView*, *OwnerDraw1/2* und *DBBrowse*.

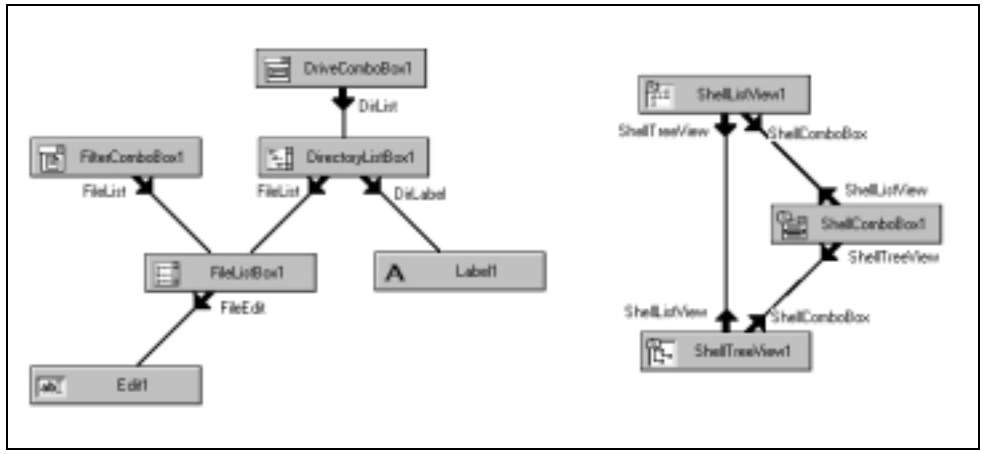


Abbildung 1.23: Die möglichen Verknüpfungen zwischen den Dateiauswahlkomponenten im alten schlichten Stil (Win 3.1) und im grafisch aufwändigen Stil des Windows Explorers (Darstellungen aus dem Diagrammeditor der Delphi-IDE)

Bedienfunktionen für eine TListbox

Der Rest dieses Kapitels beschäftigt sich in der Praxis nur mit der Komponente *TListBox*, erwähnt jedoch auch, wie Sie die hier vorgestellten Techniken einfach auf die anderen Listenkomponenten übertragen können.

Das »Problem« der *TListBox*-Komponente ist, dass sie außer für die Darstellung der Listeneinträge nur für die Auswahl eines oder mehrerer Einträge automatisch sorgt, viele weitere Funktionen, die der Benutzer oft von einer Liste erwartet, müssen Sie von Hand hinzufügen. So möchte der Benutzer wahrscheinlich:

- einen oder mehrere Einträge mit der Entf-Taste löschen
- Einträge mit der Maus innerhalb der Liste verschieben können
- nicht mehrmals dieselben Eingaben machen: Wenn er den Inhalt der Liste selbst eingeben muss, so erwartet er sicher, dass das Programm diese Eingaben bis zum nächsten Programmstart nicht wieder vergisst.

Insbesondere diese drei Funktionen werden im Beispielprogramm *Wecker3c* (Abbildung 1.24) demonstriert, das im Gegensatz zu *Wecker3b* mehrere Alarmzeiten verwalten kann. Anders als das Programm aus Kapitel 1.8 werden die Zeiten jedoch nicht in einem StringGrid, sondern in einer Listbox dargestellt. Der schon aus *Wecker3a/b* bekannte Dialog zur Eingabe der Zeit wird unverändert in das neue Projekt übernommen.

Abbildung 1.24: Die Listbox von Wecker3c bietet eine Reihe von Komfort-Funktionen.

TListBox-Inhalt

Der Inhalt einer Listbox wird unter Object Pascal als Stringliste dargestellt, die Ihnen über das Property *Items* zugänglich ist (intern verwendet *TListBox* hierfür den Typ *TStringList*, nach außen hin erscheint dieses Property aber im Gewand des abstrakten Typs *TStrings*). Sie können *Items* schon zur Entwurfszeit festlegen, wenn Sie im Objektinspektor den Stringlisten-Editor dieses Properties aufrufen; meist wird der Inhalt einer Listbox jedoch zur Laufzeit dynamisch aufgebaut. Im Beispielprogramm muss beispielsweise jedes Mal, wenn der Dialog zum Eingeben einer neuen Zeit mit OK bestätigt wurde, ein neues Element zur Liste hinzugefügt werden:

```
procedure TAlarmForm.AddButtonClick(Sender: TObject);
begin
  with AlarmEditDlg do
    if ShowModal = mrOk then
      ListBox1.Items.Append(AlarmEditDlg.Edit1.Text);
end;
```

Die Methode *Append* ist eine von vielen Methoden und Eigenschaften, die Sie auf entsprechende Weise für das *Items*-Property ansprechen können und die durch die schon erwähnte Klasse *TStrings* definiert werden. Die Zahl der Einträge erhalten Sie beispielsweise über *Items.Count*. Und um den Inhalt einer Listbox programmgesteuert abzufragen, können Sie das *Items*-Property wie ein Array-Property verwenden, unabhängig vom Beispielprogramm etwa so:

```
// Suchen des Eintrages '1984' (erster Eintrag an Position [0]!)
for i := 0 to ListBox.Items.Count - 1 do
  if ListBox.Items[i] = '1984' then Gefunden;
```

Eine allgemeine Besprechung der *TStrings*-Eigenschaften und -Methoden finden Sie in Kapitel 4.1.1.

TListBox-Auswahl und Fokus *R53*

Die wohl zweitwichtigste Funktion einer Listbox ist, dass der Benutzer einen Eintrag auswählen kann. Wenn Sie das *MultiSelect*-Property der Listbox einschalten, ist sogar eine Mehrfachauswahl erlaubt. Im Beispielprogramm kann der Benutzer zur Laufzeit über einen Schalter zwischen Einfach- und Mehrfachauswahl umschalten. Die aktuelle Auswahl in der Liste spielt in *Wecker3c* für die Schalter EDITIEREN und MARKIERTE LÖSCHEN eine Rolle.

Über das Property *ExtendedSelect* können Sie zwischen zwei Arten der Mehrfachauswahl wählen. Bei Einstellung von *False* können Sie ohne zusätzliche Tasten mit der Maus aufeinander folgende Listeneinträge markieren (indem Sie die Maus bei gedrückter Taste bewegen), bei *True* müssen Sie Shift (für zusammenhängende Bereiche) und Strg (für mehrere einzelne Einträge) zu Hilfe nehmen, um mehrere Einträge zu markieren.

Für den Schalter EDITIEREN ist es egal, ob eine Mehrfachauswahl vorliegt, denn es soll immer nur ein Eintrag editiert werden können, und zwar der *fokussierte* Eintrag. Dies ist immer der Eintrag, der zuletzt ausgewählt wurde, er ist an einer gepunkteten Umrandung zu erkennen. Im Falle der Einfachauswahl ist der fokussierte gleichzeitig auch der gewählte Eintrag. In *TListBox* können Sie den fokussierten Eintrag über das Property *ItemIndex* abfragen und ändern. Drückt der Benutzer in *Wecker3c* auf den EDITIEREN-Schalter, wird die folgende Methode aufgerufen:

```
procedure TAlarmForm.EditButtonClick(Sender: TObject);
var
  Index: Integer;
begin
  // wenn kein Eintrag gewählt/fokussiert ist, ist
  // ItemIndex -1:
  if ListBox1.ItemIndex <> -1 then begin
    with AlarmEditDlg do begin
      Index := ListBox1.ItemIndex;
      Edit1.Text := ListBox1.Items[Index];
      if ShowModal = mrOk then
        ListBox1.Items[Index] := AlarmEditDlg.Edit1.Text;
    end;
  end else
    ShowMessage('Kein Eintrag ausgewählt.');
end;
```

Wird der Dialog mit OK beendet, überschreibt das Programm einfach den Text des gewählten Eintrags mit dem Inhalt des Editierfelds im Dialog.

Im Falle des Schalters MARKIERTE LÖSCHEN muss auch eine eventuelle Mehrfachauswahl berücksichtigt werden. Hier kommt ein neues *TListBox*-Property ins Spiel: Das Array-Property *Selected*, das wie das *Items*-Array indiziert wird und zu jedem Eintrag

angibt, ob er gewählt ist. Bei eingeschalteter Mehrfachauswahl (*MultiSelect = True*) löscht die Ereignismethode für MARKIERTE LÖSCHEN alle Einträge, bei denen sie *Selected = True* vorfindet, im Falle der Einfachauswahl wird lediglich der durch *ItemIndex* angegebene aktuelle Eintrag gelöscht:

```
procedure TAlarmForm.DelButtonClick(Sender: TObject);
var
  i: Integer;
begin
(*  with ListBox1 do begin //  Der Klarheit zuliebe hier
                               keine with-Anweisung *)
  if ListBox1.MultiSelect then begin
    for i := ListBox1.Items.Count-1 downto 0 do
      if ListBox1.Selected[i] then ListBox1.Items.Delete(i);
  end else begin
    if ListBox1.ItemIndex <> -1 then
      ListBox1.Items.Delete(ListBox1.ItemIndex)
    else ShowMessage('Kein Eintrag gewählt.');
  end;
end;
```

Wichtig ist auch, dass die Löschmethode die Liste vom letzten Eintrag beginnend durchläuft. Eine einfache, nach oben zählende *for*-Schleife würde zu Fehlern führen, da sie nicht beachtet, dass sich die obere Grenze *(Items.Count*) während der Ausführung verringert und hintere Einträge automatisch in vordere Positionen aufrücken, wenn ein Eintrag gelöscht wird.

Eingaben in die Listbox *R73*

Die beiden bisher gezeigten Methoden wurden über separate Schalter gestartet und insbesondere der LÖSCHEN-Schalter überlässt es ganz der Listbox-Komponente, dass diese die Eingaben des Benutzers zur Auswahl von einem oder mehreren Einträgen richtig bearbeitet. Sie können jedoch auch selbst bei der Verarbeitung der Eingaben in die Listbox mitwirken:

- Eine besonders einfache Möglichkeit stellt die Bearbeitung des *OnClick*-Ereignisses dar, das nicht nur bei einem Klick mit der Maus ausgelöst wird, sondern auch bei jeder Änderung der Auswahl über die Tastatur. In einer Methode für dieses Ereignis könnten Sie beispielsweise Details zum gewählten Eintrag in einem Detailfenster anzeigen.
- Das Beispielprogramm bearbeitet das Ereignis *OnDblClick*, um den Editieren-Dialog aufzurufen, wenn der Benutzer auf einen Eintrag doppelklickt. Standardmäßig wird durch einen Doppelklick nämlich nur ein neuer Eintrag gewählt (wobei dies schon beim ersten der beiden Klicks geschieht). Um zusätzlich diesen Eintrag noch

editieren zu können, braucht das Programm keine neue Methode, denn *OnDblClick* kann einfach mit der oben schon besprochenen *EditButtonClick*-Methode verknüpft werden.

- Auch für den Schalter MARKIERTE LÖSCHEN bietet es sich an, eine Alternative zur Verfügung zu stellen. Denn wozu gibt es auf der Tastatur die Taste Entf? Um Tastatureingaben in die Listbox zu bearbeiten (also Eingaben, die der Benutzer macht, während die Listbox oder einer ihrer Einträge fokussiert ist), schreiben Sie eine Methode für *OnKeyDown* oder *OnKeyPress*. Im Beispiel muss nur abgefragt werden, ob es sich um die Löschtaste handelt, den Rest der Arbeit kann die bereits gezeigte Methode des Löschen-Schalters übernehmen:

```
procedure TAlarmForm.ListBox1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = VK_DELETE then
    DelButtonClick(nil);
end;
```

Eine Auflistung aller Tastencodes wie *VK_DELETE* finden Sie in der mit Delphi mitgelieferten Win32-Hilfedatei unter dem Stichwort »Virtual-Key Codes«. Zu weiteren Details bezüglich *OnKeyDown* und *OnKeyPress* sei hier auf Kapitel 3.3.2 verwiesen.

Speichern des ListBox-Inhaltes *R115*

Damit der Inhalt der Liste zwischen den Programmläufen nicht verloren geht, ruft *Wecker3c* beim Start (*OnCreate*-Ereignis des Formulars) eine Methode namens *RestoreList* und am Ende (*OnClose*-Ereignis des Formulars) eine *SaveList*-Methode auf, die im Folgenden kurz besprochen werden sollen, denn nach dem darin demonstrierten Prinzip können Sie auch beliebige andere Einstellungen und Eingaben des Benutzers in Ihren Anwendungen speichern.

Zur Speicherung des gesamten ListBox-Inhalts erweist sich das Property *Items.CommaText* als sehr vorteilhaft, denn es beinhaltet alle Einträge der Liste in einem einzigen String. Statt also in einer Schleife jeden Listeneintrag einzeln speichern zu müssen, können wir die Liste in einem einzigen Schritt speichern!

Das Beispielprogramm verwendet für diese Speicherung die Windows-Registrierungsdatenbank (Registry), die in einer Verzeichnisstruktur ähnlich einem Dateisystem organisiert ist und die in Kapitel 4.2.2 ausführlich besprochen wird. Für den Pfad innerhalb der Registry und den Namen des Schlüssels, unter dem die Information (*CommaText*) abgelegt wird, definiert es zwei Konstanten:

```
const
  REG_PATH = 'Software\Delphi4BuchEW\Kapitel1Wecker';
  REG_KEY = 'ListContentsFormat1';
```

Mit Hilfe der (ebenfalls in Kapitel 4.2.2 erläuterten) Klasse *TRegistry* sind nur wenige Zeilen erforderlich, um das Ziel zu erreichen:

```
procedure TAlarmForm.SaveList;
var
  Registry: TRegistry;
  i: integer;
begin
  Registry := TRegistry.Create; // Registry-Objekt initialisieren
  Registry.Rootkey := HKEY_Current_User; // Registry-Bereich auswählen
  Registry.OpenKey(REG_PATH, True); // Pfad innerhalb des Bereiches
                            // suchen und bei Bedarf neu erzeugen
  // Ablegen der Information unter dem Schlüssel REG_KEY:
  Registry.WriteString(REG_KEY, ListBox1.Items.CommaText);
  Registry.Free; // Registry-Objekt freigeben
end;
```

Für das Wiederherstellen der Liste wird zunächst der ehemalige *CommaText*-String mit *TRegistry.ReadString* aus der Registry gelesen. Er kann dann sofort wieder in das *Items.CommaText* geschrieben werden. Durch das besondere Format dieses Strings (Anfang und Ende jedes Eintrags sind durch doppelte Anführungszeichen eindeutig erkennbar) kann die Listbox bzw. das in ihr enthaltene *Items*-Objekt die ursprünglichen Einzeleinträge exakt wiederherstellen.

```
procedure TAlarmForm.RestoreList;
var
  Registry: TRegistry;
  s: String;
begin
  Registry := TRegistry.Create;
  Registry.Rootkey := HKEY_Current_User;
  if Registry.OpenKey(REG_PATH, False) then begin
    if Registry.ValueExists(REG_KEY) then begin
      s:=Registry.ReadString(REG_KEY);
      ListBox1.Items.CommaText := s;
    end;
  end;
  Registry.Free;
  if ListBox1.Items.Count > 0 then
    ListBox1.ItemIndex := 0; // den ersten Eintrag der Liste vorauswählen
end;
```

Weitere Informationen zu *TStrings*, zum Property *TStrings.CommaText* und dem ähnlichen Property *TStrings.Text* finden Sie übrigens in Kapitel 4.1.1.

Wie bereits erwähnt, wird *SaveList* beim *OnClose*-Ereignis, *RestoreList* im *OnCreate*-Ereignis des Formulars aufgerufen:

```
procedure TAlarmForm.FormCreate(Sender: TObject);
begin
```

```
  (* Alarmfunktion per Voreinstellung eingeschaltet: *)
  Alarmaktiviert1.Checked := True;
  (* Keinen Alarm mehr für gespeicherte Zeiten
     geben, die schon beim Start des Programms abgelaufen sind: *)
  LastAlarm := Time; // zu LastAlarm siehe IsTimeOver in der
                     // Einleitung zu Kapitel 1.9.
  TimerTimer(nil); (* Zeitanzeige initialisieren *)
  RestoreList; (* Vom letzten Programmlauf gespeicherte Alarme laden *)
end;

procedure TAlarmForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  SaveList;
end;
```

Datenobjekte in einer ListBox speichern

Zur Laufzeit des Programms werden also alle wichtigen Alarmdaten in einer *TListBox*-Komponente aufbewahrt. Da es sich bei jedem Alarm um einen einzelnen String handelt, bereitet dies auch noch keine Probleme. Angenommen aber, zu jedem Alarmereignis gäbe es neben der Alarmzeit noch eine Reihe weiterer Daten wie etwa einen Kommentar oder zusätzliche Angaben dazu, was bei Eintreten der Alarmzeit getan werden soll, dann wird es etwas schwieriger, auch diese Daten im Speicher zu halten.

TListBox ist aber schon auf diese Situation vorbereitet, denn dank der Klasse *TStrings* können Sie zu jedem Eintrag zusätzlich auch ein Objekt speichern – so befindet sich das Objekt zum i-ten Eintrag in *ListBox.Items.Objects[i]*. In Kapitel 1.9.4 wird das Programm auch diese Listbox-Eigenschaft nutzen.

Drag&Drop in einer ListBox *R47*

Eingebaut in die Listbox von *Wecker3c* ist auch eine Drag&Drop-Funktion, die den Drag&Drop-Möglichkeiten einiger Listboxen der Delphi-IDE entspricht (z.B. der Liste der Palettenseiten im Dialog zur Konfiguration der Komponentenpalette und in einigen Property-Editoren wie etwa dem Felder-Editor oder dem Befehlslisten-Editor).

Da die Grundlagen des Drag&Drop ausführlich in Kapitel 5.8.3 beschrieben werden, wird sich dieses Kapitel auf die für *TListBox* relevanten Details beschränken.

- Zunächst ist es empfehlenswert, die Listbox in den *MultiSelect*-Modus zu schalten, denn nur dann kann während des Drag&Drop unter der Maus ein dezentes Fokus-Rechteck gezeichnet werden, ohne dass dadurch die Markierung des aktuellen Eintrags geändert wird.
- Dann ist auch der Modus *ExtendedSelect* erforderlich, damit der Drag&Drop-Vorgang nicht zu einer Mehrfachauswahl führt.

Das Beispielprogramm trifft alle notwendigen Vorbereitungen, sobald Sie zur Laufzeit den Drag&Drop-Markierungsschalter aktivieren. Es schaltet eine eventuelle Sortierung der Liste aus und die genannten Optionen an, deaktiviert die Markierungsschalter (*Enabled:=False*), damit der Benutzer gleich sieht, dass er diese Optionen bei Verwendung von Drag&Drop nicht verändern kann, und schaltet den Drag&Drop-Modus der VCL an (*DragMode:=dmAutomatic*, dies wird wieder in Kapitel 5.8.3 ausführlich besprochen):

```
procedure TAlarmForm.CBDragDropClick(Sender: TObject);
begin
  if CBDragDrop.Checked then begin
    // Drag&Drop-Checkbox wurde eingeschaltet
    SortCheck.Checked := False; SortCheck.Enabled := False;
    CBMultiSel.Checked := True; CBMultiSel.Enabled := False;
    CBExtendedSel.Checked := True; CBExtendedSel.Enabled := False;
    Listbox1.DragMode:=dmAutomatic;
  end else begin
    // Das Gegenteil, wenn DragDrop-Schalter abgeschaltet wird:
    SortCheck.Enabled := True;
    CBMultiSel.Enabled := True;
    CBExtendedSel.Enabled := CBMultiSel.Checked;
    Listbox1.dragMode := dmManual;
  end;
end;
```

Der eigentliche Drag&Drop-Vorgang basiert auf den Ereignissen *OnDragOver* und *OnDragDrop* (Kapitel 5.8.3!). Mit Hilfe der Methode *ItemAtPos* kann das Programm herausfinden, über welchem Eintrag sich die Maus gerade befindet (die Koordinaten *X* und *Y* werden von den *OnDrag*-Ereignissen übermittelt):

```
DragOverIndex := Listbox1.ItemAtPos(Point(X, Y), true);
```

Mit Hilfe der so gewonnenen Eintrags-Indizes ist es ein Leichtes, den gezogenen Eintrag an eine andere Stelle zu versetzen (*Items.Delete* für die alte Position, *Items.Insert* an der neuen Position, siehe CD).

Optimal ist diese Implementierung noch nicht, denn ein automatisches Scrollen findet nicht statt, falls Sie versuchen, einen Eintrag der Liste nach oben oder nach unten über die Listbox hinauszuziehen; aber wie Sie an der Delphi-IDE sehen können, ist diese Funktion nicht so wichtig. Wenn Sie einen Listeneintrag an eine nicht in der Liste sichtbare Stelle verschieben wollen, können Sie in der Delphi-IDE jedoch Schalter verwenden, die den Eintrag schrittweise nach oben oder nach unten befördern. Um solche Schalter könnte auch *Wecker3c* sehr leicht nachgerüstet werden.

ListBox-Alternativen

Die folgende Tabelle dient gleichzeitig als Zusammenfassung der vorherigen Abschnitte und als Hinweis, wo Sie die verschiedenen gezeigten *TListBox*-Merkmale bei der modernen Variante der ListBox, dem *ListView*, finden. Zuerst noch zwei Hinweise auf die näheren Verwandten der ListBox:

- *TFileListBox* und *TDirectoryListBox* sind, was die Elemente der Tabelle betrifft, mit *TListBox* identisch, denn beide sind von *TCustomListBox* abgeleitet, und alle in der Tabelle genannten Elemente sind bereits in *TCustomListBox* deklariert.
- *TComboBox* besitzt bis auf *MultiSelect*, *SelCount* und *ItemAtIndex* ebenfalls alle in der Tabelle für *TListBox* aufgelisteten Elemente.

Information	**TListBox**	**TListView**
Einträge	Items	Items
Zahl der Einträge	ItemCount	Items.Count
i-ter Eintrag	Items[i]	Items[i]
Eintrag hinzufügen	Items.Append/Add oder: AddItem (ab Delphi 6)	Items.Add oder: AddItem (ab Delphi 6)
Eintrag&Objekt hinzufügen	Items.AddObject	Objekt muss extra zugewiesen werden.
Eintrag löschen	Items.Delete	Items.Delete
Fokussierter Index	ItemIndex	Items[ItemFocused].Index oder ItemIndex (ab Delphi 6)
Fokussierter Eintrag	Items[ItemIndex]	ItemFocused
Mehrfachauswahl	MultiSelect	MultiSelect
Zahl gewählter Einträge	SelCount	SelCount
Eintrag i gewählt?	Selected[i]	Items[i].Selected
Eintrag an (Maus-)Position	ItemAtIndex(x, y)	GetItemAt(x, y)
Ereignis bei Auswahländerung	OnChange	OnSelectItem
Ereignis bei Doppelklick	OnDblClick	OnDblClick
Ereignisse bei Tastatureingabe	OnKeyDown, OnKeyPress,...	OnKeyDown, OnKeyPress,...
Verknüpftes Objekt	Items.Objects[i]	Items[i].Data

Hinweis für Kenner der Vererbungsmechanismen in Object Pascal: Borland hat in Delphi 6 neue gemeinsame Basisklassen für *TListBox* und *TListView* eingefügt: Beide haben nun *TCustomMultiSelectListControl* und *TCustomListControl* unter ihren Vorfahren, wodurch die Schnittstellen von *TListBox* und *TListView* teilweise aneinander angeglichen werden. Die genannten *TCustom...*-Klassen sind unter anderem für die Definition der in der Tabelle angegebenen Properties *ItemIndex*, *MultiSelect* und *SelCount* sowie der Methode *AddItem* verantwortlich. Weitere gemeinsame Methoden wie *DeleteSelected*, *MoveSelection* und *SelectAll* finden Sie in der Online-Hilfe oder direkt bei der Code-Eingabe durch die »Programmierhilfen« des Editors. Übrigens ist auch *TComboBox* in diese neuen Verwandtschaftsbeziehungen eingebunden, allerdings nur über *TCustomListControl* und nicht über *TCustomMultiSelectListControl*, da die ComboBox keine Möglichkeit der Mehrfachauswahl besitzt.

1.9.4 Eingabekomponenten

In diesem Kapitel geht es um die zahlreichen Steuerelemente, in die der Benutzer Tastatur- oder Mauseingaben machen kann, die Sie im Programmcode bequem über Properties abfragen können; das Beispielprogramm zeigt auch die mögliche Verarbeitung dieser Eingaben in einer Delphi-Anwendung.

Zwischen Ein- und Ausgabekomponenten lässt sich keine perfekte Trennungslinie ziehen: Wenn Sie einen Mausklick als Eingabe ansehen, können Sie beispielsweise jede Komponente zu einer Eingabekomponente machen, indem Sie ihr *OnMouseDown*-Ereignis bearbeiten. Daher soll die folgende Aufstellung diejenigen Komponenten nennen, die *üblicherweise* zur Eingabe benutzt werden. Manchmal ist es auch wünschenswert, eine Komponente gleichzeitig zur Eingabe und zur Ausgabe zu verwenden. Dies geschieht im TreeDesigner-Projekt z.B. in der Zeichenfläche (die hierfür zugrunde liegende Komponente *TPaintBox* wird in Kapitel 1.9.5 unter Ausgabekomponenten aufgezählt).

Soweit nicht anders angegeben, finden Sie die Komponenten auf der *Standard*-Seite der Komponentenpalette.

TCheckBox-Markierungsschalter haben zwei mögliche Zustände: nicht markiert und markiert (Property *Checked*). Optional können Sie noch einen dritten Zustand annehmen, den Sie nur über das Property *State* abfragen können. In diesem Zustand erscheint das Markierungsfeld des Schalters grau (nicht aber der Schalter selber, der erst bei *Enabled=False* grau dargestellt wird). In diesem dritten Zustand hat *State* den Wert *cbGrayed* und *Checked* ist *False*. Dieser Zustand ist nur erreichbar, wenn Sie das *AllowGrayed*-Property des Schalters auf *True* setzen.

TRadioButton-Komponenten können ebenfalls zwei Zustände annehmen und darüber hinaus von benachbarten Radioschaltern abhängig sein: In einer Gruppe von mehreren Radioschaltern kann immer nur einer eingeschaltet bzw. markiert sein. Schaltet der Benutzer einen anderen Schalter der Gruppe ein, wird der bisher markierte Schalter automatisch ausgeschaltet. Eine *Gruppe* bilden dabei alle Schalter, die sich innerhalb einer übergeordneten Komponente wie etwa einer *GroupBox*, einem *Panel* oder auch dem Formular selbst befinden. Besonders einfach lassen sich Gruppen von Radioschaltern mit der Komponente *TRadioGroup* erstellen.

Bei *TRadioGroup* brauchen Sie keine einzelnen Radioschalter in das Formular einzuzeichnen, denn diese Komponente erzeugt die Radioschalter automatisch. Sie müssen lediglich das Property *Items* mit den Texten der einzelnen Schalter füllen. Klicken Sie dazu im Objektinspektor auf den Schalter rechts neben *Items*. Es erscheint ein Stringlisten-Editor, in dem Sie die Texte für die einzelnen Radioschalter eingeben können.

TRadioGroup erzeugt für jeden String der Liste einen Radioschalter und verteilt sämtliche Schalter gleichmäßig in vertikaler Reihenfolge (bei Verwendung des Properties *Columns* auch auf mehrere Spalten). Dies ist eine besondere Erleichterung, wenn Sie nachträglich einen Schalter hinzufügen oder entfernen. Auch die Abfrage der eingeschalteten Option gestaltet sich mit *TRadioGroup* einfacher als mit einzelnen Radioschaltern, müssen Sie bei *TRadioGroup* doch nur das *ItemIndex*-Property auslesen (für den ersten Schalter nimmt es den Wert 0 an, ist kein Schalter gewählt, gibt es den Wert -1 an).

TEdit ist eine Komponente für ein einzeiliges Editierfeld, bei der Sie normalerweise nicht mehr als das *Text*-Property zur Abfrage des Textes und das *OnChange*-Ereignis zur eventuellen Überwachung von Änderungen abfragen müssen. Über die Vererbungshierarchie ist *TEdit* eng mit *TMemo* verwandt und stimmt auch in einigen weiteren Fähigkeiten mit diesem überein (z.B. bezüglich Zwischenablageoperationen und markiertem Text).

TMemo ist ein mehrzeiliges Editierfeld, das einen automatischen Zeilenumbruch durchführen (Property *WordWrap*) oder / und den Text mit Scrollbars scrollen kann (Property *Scrollbars*). Über das *Lines*-Property haben Sie Zugriff auf jede einzelne Zeile des Textes. *TMemo* hat eine ausführlichere Erläuterung in Kapitel 3.6.1verdient.

TUpDown (Seite *Win32* der Komponentenpalette) besteht aus zwei Schaltern, mit denen Sie einen Zahlenwert in einem Editierfeld per Maus vergrößern oder verkleinern können, und zwar in einer vom *Increment*-Property angegebenen Schrittweite. Das nächste Beispielprogramm wird die weitere Kooperation zwischen *TEdit* und *TUpDown* demonstrieren.

Die Komponente *TRichEdit* (Seite *Win32*) kapselt mit dem RTF-Editierfeld einen mit vielen Möglichkeiten der Textformatierung ausgestatteten Texteditor, dem es lediglich noch an den Kontrollelementen fehlt, über welche die vielen Funktionen auch aktiviert werden können (siehe Kapitel 3.6.1).

THotKey (*Win32*) stellt ein Editierfeld dar, das sich völlig anders als ein normales Editierfeld verhält. Es zeigt die Bezeichnungen der gedrückten Tasten in Textform an und gibt sie als Tastenkürzelwert zurück. Mehr zu dieser Komponente finden Sie im Zusammenhang mit dem Tastenkürzeleditor in Kapitel 4.6.3.

TMaskEdit (Seite ZUSÄTZLICH) ist ein weiterer Typ von Editierfeldern, der sich gegenüber *TEdit* durch das Property *EditMask* auszeichnet, in dem Sie festlegen, welchem Muster die Eingabe des Benutzers entsprechen muss. So bewirkt beispielsweise die Maske »90:00«, dass das Feld sich dem Benutzer mit der Angabe von »__:__« präsentiert. Der Benutzer kann nur Ziffern eingeben, darf aber die erste Ziffer weglassen. Das *EditMask*-Property verfügt sogar über einen eigenen Property-Editor, in dem Sie weitere Beispielmasken finden, Hilfe abrufen und sogar Ihre Maske testen können.

TLabeledEdit ist ein spezielles Editierfeld (Seite ZUSÄTZLICH, ab Delphi 6), das mit einer Beschriftung versehen ist. Die Position der Beschriftung stellen Sie mit *LabelPosition* und *LabelSpacing* ein; die Beschriftung selbst befindet sich im Unterproperty *EditLabel.Caption*.

Ein *TrackBar* (Seite WIN32) ist ein Regler zum Einstellen eines Zahlenwertes, der im Property *Position* zur Verfügung steht. Sehr viele seiner exklusiven Properties befassen sich mit reinen Äußerlichkeiten (*Frequency*, *LineSize*, *Orientation*, *ThumbLength*, *TickMarks*, *TickStyle*), einige betreffen aber auch die Funktionsweise (*Min*, *Max*, *PageSize*, *SelStart*, *SelEnd* und *SliderVisible*). Allen Properties ist gemein, dass sie sich schnell durch ihren Namen oder die Online-Hilfe erklären lassen.

TDateTimePicker (Seite WIN32) ist ein Eingabeelement speziell zur Eingabe eines Datums oder einer Uhrzeit (je nach Einstellung des Properties *Kind*); beides können Sie komfortabel aus den Properties *Date* und *Time* auslesen, die in Delphis Zeitformat *TDateTime* gehalten sind (siehe Kapitel 2.8.3). Der Benutzer kann sich bei der Eingabe der einzelnen Datum- und Zeitbestandteile Pfeilen wie bei einer *UpDown*-Komponente bedienen und das Datum sogar aus einem Kalender auswählen (falls *DateMode=dmComboBox*). Bei der Datumseingabe ist auch eine ausführliche Schreibweise möglich (z.B. *Montag, 19. Mai 1997* statt *19.05.1997*, Property *DateFormat*). Schließlich können Sie auf der linken Seite des Eingabefeldes noch ein Markierungsfeld erscheinen lassen (Property *ShowCheckbox*).

TMonthCalendar (WIN32, ab Delphi 4) stellt den schon von *TDateTimePicker* bekannten Monatskalender als separate Komponente zur Verfügung, in der Sie als zusätzliche Optionen beispielsweise Wochennummern wählen oder den ersten Tag der Woche definieren können.

TStringGrid (Seite *Zusätzlich*) ist eine ganz in Object Pascal geschriebene (also nicht als Windows-Steuerelement existierende) editierbare Stringtabelle mit frei einstellbaren Zeilen- und Spaltengrößen, die bereits in Kapitel 1.8 vorgestellt und eingesetzt wurde. Ähnlich wie *TStringGrid* kann übrigens auch die später bei den Ausgabekomponenten aufgelistete *TDrawGrid*-Komponente zur Eingabe verwendet werden.

TValueListEditor ist ein spezialisiertes StringGrid, das erst ab Delphi 6 in der Komponentenpalette enthalten ist (Seite ZUSÄTZLICH). Schon in früheren Delphi-Versionen können Sie ein Verwendungsbeispiel dieser Komponente im Projektoptions-Dialog auf der Seite *Versionsinfo* oder im Property-Editor von *TDatabase.Params* sehen: Editiert wird eine Liste von Schlüssel/Wert-Paaren, die alle zusammen in einem einzigen *TStrings*-Objekt gespeichert werden (zur Interpretation einer Stringliste als Schlüssel/Wert-Paare siehe Kapitel 4.1.1, Properties *Names* und *Values*). Je nachdem, welche *KeyOptions* Sie für *TValueListEditor* einstellen, können nicht nur die Werte, sondern auch die Schlüssel editiert werden, es können Schlüssel hinzugefügt oder gelöscht werden, und es kann sichergestellt werden, dass jeder Schlüssel nur einmal vorkommt.

Neue Dialogelemente für das Beispielprogramm

Abbildung 1.25 zeigt die neue Version des Beispielprogramms, die sich von der letzten Version zur Entwurfszeit nur im Alarmeingabe-Dialog unterscheidet, in dem die Komponentenklassen *TEdit*, *TUpDown*, *TTrackBar*, *TMemo*, *TDateTimePicker*, *TRadioButton* und *TCheckBox* zum Einsatz kommen. Die Änderungen an der Listbox fallen nur zur Laufzeit auf: Sie zeigt nun einige der zusätzlichen Attribute an, die im Dialog eingegeben wurden.

Der Umgang mit den Eingabekomponenten im Programmcode braucht nicht groß erläutert zu werden – jede Komponentenklasse enthält ein Property, aus dem die Eingabedaten schnell abgefragt werden können, so wie in der folgenden Methode (die besagten Properties sind in Fettschrift wiedergegeben, Bezeichner des Programms, die später erläutert werden, in Kursivschrift):

```
procedure TAlarmEditDlg.GetAlarmData(Ziel: TAlarm);
begin
  with Ziel do begin
```

```
    Alarmzeit := Frac(DateTimePicker1.Time); (* Frac gibt den Nachkomma-
              Anteil der Time-Angabe zurück, die der Uhrzeit entspricht,
              zum Format von Time siehe den Typ TDateTime *)
    Kommentar := KommentarMemo.Text;
    if RadioButton1.Checked then HinweisArt := haMeldung;
    if RadioButton2.Checked then HinweisArt := haSignal;
    if RadioButton3.Checked then HinweisArt := haBeides;
    AnzahlToene := AnzahlToeneUpDown.Position;
    Programm := ProgrammCheckBox.Checked;
    ProgrammPfad := ProgrammPfadEdit.Text;
  end;
end;
```

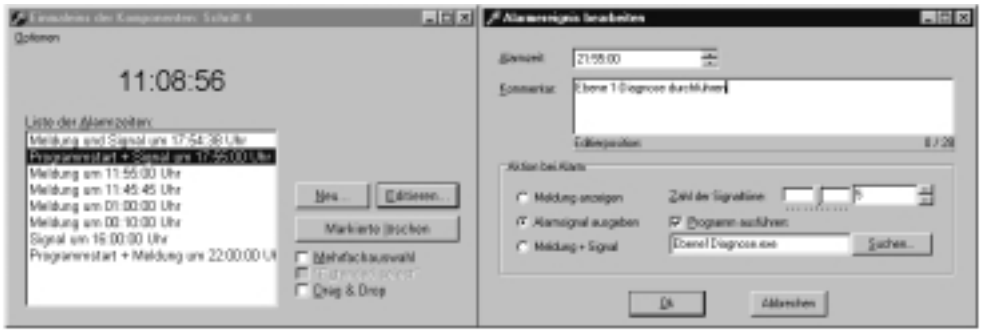


Abbildung 1.25: Wecker3d – die vierte Version des Beispielprogramms

Für jede der verwendeten Klassen finden Sie außerdem ein Ereignis, das bei Änderung des Inhalts durch den Benutzer eintritt. Dieses heißt entweder *OnClick* bei Schaltern oder *OnChange* bei den anderen Komponentenklassen. Im Beispielprogramm wird dieses Ereignis für den *TrackBar* verwendet, um bei jeder Änderung des Reglers den neuen Wert in das nebenstehende Editierfeld zu schreiben:

```
procedure TAlarmEditDlg.AnzahlToeneTrackBarChange(Sender: TObject);
begin
  AnzahlToeneEdit.Text := IntToStr(AnzahlToeneTrackBar.Position);
end;
```

Hinweis: Zur Aktualisierung der Positionsangabe unter dem Memo-Feld bearbeitet *Wecker3d* das *OnKeyUp*-Ereignis. Die aktuelle Eingabeposition finden Sie im *TMemo*-Property *CaretPos*. Weitere Details zu *TMemo* und *TEdit* finden Sie in Kapitel 3.6.1.

Eine Klasse zur Speicherung der Dialogeingaben

Der Sinn des Beispielprogramms liegt jedoch weniger in der Vorführung dieser einfachen Eingabekomponenten als darin,

- die Definition einer neuen Klasse für die Daten
- den Austausch der Daten zwischen dem Dialog und dem Hauptformular
- und die Speicherung der Daten in der ListBox und in der Registry

zu demonstrieren.

Eine aufwändigere Lösung ist hier kaum zu vermeiden, da Delphi für die genannten einfachen Zwecke keine Automatismen zur Verfügung stellt. Wir können die Daten ja nicht einfach in den Komponenten des Dialogs stehen lassen, denn dort ist nur Platz für die Angaben zu einem einzigen Alarm, das Programm muss aber in der Lage sein, mehrere Alarme speichern zu können. Und in der bisher zur Aufbewahrung der Daten verwendeten Listbox gibt es für jedes Element nur einen einfachen String, der den vielen Alarmparametern kaum gerecht werden kann.

Theoretisch wäre dieses Problem einfach dadurch zu lösen, die Alarmeinträge in einer Datenbank abzulegen und somit die Verwaltung der Daten an das Datenbanksystem zu delegieren. Wenn man sich aber überlegt, dass dann ein an sich winziges Utility wie dieses Beispielprogramm von umfangreichen Bibliotheken und Programmen abhängig wäre, wird schnell klar, dass auch ein Delphi-Programm kleine Datentabellen »von Hand« verwalten darf.

Hinweis: Diese Version des Beispielprogramms macht bereits von der Möglichkeit Gebrauch, eigene Properties zu definieren. Dieses Object-Pascal-Sprachmerkmal wird in Kapitel 2.2.5 beschrieben, wo Sie auch nachlesen sollten, falls hierzu an dieser Stelle noch Fragen offen bleiben sollten.

Wecker3d, genauer die Unit des Dialogformulars, deklariert also eine neue Klasse, die alle zu einem Alarm gehörigen Daten umfasst und zusätzlich einige nützliche Hilfsmethoden bereitstellt. Eigentlich handelt es sich um eine Mischung zwischen Record und Klasse, denn *TAlarm* deklariert die einzelnen Datenelemente als öffentlich, so dass man sie wie die Elemente eines Records ansprechen kann. Normalerweise sollte eine Klasse keinen direkten Schreibzugriff auf ihre Daten von außen erlauben, doch die Erfüllung dieser Forderung würde das Programm noch aufwändiger machen.

```
TAlarm = class
  function GetEreignisText: String; // intern (d.h. von den unten
  function GetListText: String;   // deklarierten Methoden) verwendete
  function DataSize: Integer;     // Hilfsfunktionen
public
  (* Datenelemente - Zugriff wie bei einem Record *)
  Alarmzeit: TDateTime;
  Kommentar: String[255];
  HinweisArt: THinweisArt;
  AnzahlToene: Integer;
```

```
    Programm: Boolean;
    ProgrammPfad: String[255];
    (* Methoden und Properties: *)
    constructor Create; // Initialisierung eines neuen TAlarm-Objekts
    // Alle Daten in einem String angeben (für die Listbox im Hauptfenster):
    property ListText: String read GetListText;
    // zur Speicherung der Daten bei Programmende:
    procedure SaveToStream(S: TStream);
    procedure LoadFromStream(S: TStream);
    procedure UpdateListItem(item: TListItem); // in Kapitel 1.9.5 benötigt
    procedure AddToListView(lv: TListView); // in Kapitel 1.9.5 benötigt
  end;
```

Initialisierung und Rückgabe der Dialogeingaben *R25*

Auf Grundlage der Klasse *TAlarm* kann die Dialogformular-Unit dem Hauptformular nun eine einfache Schnittstelle zur Verfügung stellen, die wir uns zunächst aus der Sicht des Hauptformulars ansehen. Wird der Schalter HINZUFÜGEN betätigt, so erzeugt dieses ein neues Objekt der Klasse *TAlarm*, indem es den *Create*-Konstruktor dieser Klasse aufruft. Dieser sorgt auch dafür, dass die einzelnen Datenelemente mit Vorgabewerten belegt werden:

```
procedure TAlarmForm.AddButtonClick(Sender: TObject);
var
  NewData: TAlarm;
begin
  NewData := TAlarm.Create;
```

Um den Dialog mit diesen Vorgabewerten zu initialisieren, genügt die Zuweisung des *TAlarm*-Objekts an ein selbst definiertes Property des Dialogformulars:

```
  AlarmEditDlg.AlarmData := NewData;
```

Der darauf folgende Aufruf des Dialogs findet weiterhin mit *ShowModal* statt. Im Erfolgsfall werden die geänderten Daten mit der Funktion *GetAlarmData* abgefragt:

```
  if AlarmEditDlg.ShowModal = mrOk then begin
    AlarmEditDlg.GetAlarmData(NewData);
```

GetAlarmData erwartet als Parameter ein bereits initialisiertes *TAlarm*-Objekt, in das es die Daten hineinschreiben kann. Nach diesem Aufruf gilt es, das neue Alarmereignis in der Listbox anzuzeigen.

Datenobjekte mit den Listeneinträgen verbinden *R48*

Hier kommt die Möglichkeit von *TListBox*, mit jedem String auch ein zugehöriges Objekt zu speichern, ins Spiel. Statt *Items.Add* oder *Items.Append* dient jetzt *AddObject* dazu, den Listentext (erster Parameter) zusammen mit dem Objekt (zweiter Parameter) zu speichern:

```
      ListBox1.Items.AddObject(NewData.ListText, NewData)
    end else
      NewData.Free; // das TAlarm-Objekt freigeben
  end;
```

Dabei ist *ListText* eine Methode von *TAlarm*, die einige der Alarmparameter zu einem String zusammenfasst, den das Beispielprogramm in der Listbox statt der reinen Alarmzeit anzeigt.

Die Methode des Schalters EDITIEREN verläuft ähnlich. Es gibt vor allem eine interessante Zeile in der Methode: Wenn ein bereits vorhandener Eintrag der Liste editiert werden soll, muss der Alarm-Dialog mit den Werten dieses Eintrages initialisiert werden, denn der Benutzer will ja die schon eingegebenen Werte ändern. Also muss das Programm das *TAlarm*-Objekt wiederfinden, das vorher einmal wie oben gezeigt mit *AddObject* zur Liste hinzugefügt wurde. Dafür bedient es sich des Properties *Items.Objects*, in dem zu jedem String der Liste an gleicher Position das zugehörige Objekt gespeichert ist:

```
ListBox1.Items.Objects[Listbox1.ItemIndex] // bezeichnet das Objekt für
                           // den gegenwärtig fokussierten Listeneintrag
```

Eine direkte Zuweisung dieses Objekts an eine *TAlarm*-Variable ist allerdings nicht möglich, da *Items.Objects[Index]* in der VCL als Typ *TObject* deklariert ist (die VCL kennt ja unseren Typ *TAlarm* noch nicht). Um dem Compiler mitzuteilen, dass es sich bei dem Objekt in Wirklichkeit um ein *TAlarm*-Objekt handelt, ist die folgende Typumwandlung notwendig.

```
ChangedAlarm := TAlarm(ListBox1.Items.Objects[Listbox1.ItemIndex]);
AlarmEditDlg.AlarmData := ChangedAlarm; // Initialisierung des Dialogs
  if AlarmEditDlg.ShowModal = mrOk then begin // Anzeige des Dialogs
```

Damit das Hauptformular wie gezeigt mit dem Dialogformular kommunizieren kann, muss Letzteres als Schnittstelle das Property *AlarmData* (mit der Schreibmethode *SetAlarmData*) und die Methode *GetAlarmData* implementieren. Da die beiden sich fast spiegelbildlich zueinander verhalten, genügt hier die oben bereits abgedruckte Methode *AlarmEditDlg.GetAlarmData*.

Hinweis: *GetAlarmData* hätte auch als Funktion geschrieben werden können, die ein *TAlarm*-Objekt zurückgibt, und diese Funktion hätte dann sogar als Lesemethode für das oben zum Schreiben verwendete *AlarmData*-Property dienen können, aber in diesen beiden Fällen wäre unklar gewesen, wer für die Initialisierung des *TAlarm*-Objektes zuständig gewesen wäre. Im Beispielprogramm gilt die Regel, dass ausschließlich das Hauptfenster die *TAlarm*-Objekte mit *Create* erzeugen und mit *Free* freigeben darf.

Zum Abschluss hier noch die vollständige Bearbeitungsmethode für den EDITIEREN-Schalter:

```
procedure TAlarmForm.EditButtonClick(Sender: TObject);
var
  Index: Integer;
  ChangedAlarm: TAlarm;
begin
  if ListBox1.ItemIndex <> -1 then begin
    Index := ListBox1.ItemIndex;
    ChangedAlarm := TAlarm(ListBox1.Items.Objects[Index]);
    AlarmEditDlg.AlarmData := ChangedAlarm;
    if AlarmEditDlg.ShowModal = mrOk then begin
      ListBox1.Items.Delete(Index);
      AlarmEditDlg.GetAlarmData(ChangedAlarm);
      ListBox1.Items.InsertObject(Index, ChangedAlarm.GetListText,
                                  ChangedAlarm);
      (* Die Markierung des Eintrages ging beim Löschen verloren
         und wird jetzt wiederhergestellt: *)
      ListBox1.ItemIndex := Index;
    end;
  end else
    ShowMessage('Kein Eintrag ausgewählt.');
end;
```

Speicherung der Daten *R117*

Für die Speicherung der Daten genügt es jetzt nicht mehr, das Property *CommaText* der ListBox in die Registry zu schreiben. *Wecker3d* schreibt alle Alarmdaten daher in binärer Form in die Registry. Dabei geht es so vor wie beim Erzeugen einer normalen binären Datei (siehe etwa Kapitel 5.3.3 unter *Dateioperationen*), verwendet aber als Ziel keine Datei auf der Festplatte, sondern einen Memory-Stream (zu Streams siehe Kapitel 4.3.1). Der Inhalt dieses Streams kann dann mit *TRegistry.WriteBinaryData* in die Registry geschrieben werden:

```
Registry.ReadBinaryData(REG_KEY, Stream.Memory^, Stream.Size);
```

Dabei verwendet *Wecker3d* für *REG_KEY* einen anderen Wert als *Wecker3c*, damit nicht eine dieser beiden Programmversionen versucht, die von der jeweils anderen Version geschriebenen und völlig unterschiedlich formatierten Daten zu lesen.

Hinweis: Durch die sehr einfache Speichermethode des Beispielprogramms werden pro Alarm über 500 Bytes gespeichert. Normalerweise sollten Sie mit dem Speicherplatz der Registry vorsichtig umgehen und keine größeren binären Dateien darin speichern.

Für weitere Details sei auch hier wieder auf den Quelltext der CD verwiesen.

Freigabe der Daten

Mit der Verwendung einer neuen Klasse zur Speicherung der Alarmdaten geht natürlich auch die »moralische« Verpflichtung einher, die neu konstruierten Datenobjekte auch wieder freizugeben. In den oben gezeigten Zeilen findet zum Beispiel eine solche Freigabe statt, wenn der Benutzer den Dialog zum Einfügen eines neuen Elements abbricht. Das Programm verwirft das vorher neu erzeugte Objekt dann wieder mit *Free*. Auch wenn der Benutzer Objekte aus der Liste löscht, ruft *Wecker3d* die Methode *Free* für die entsprechenden *TAlarm*-Objekte auf. Am Ende des Programms, also beim Ereignis *OnDestroy* des Hauptformulars, werden schließlich alle noch in der Liste befindlichen Alarm-Objekte freigegeben. Weitere Informationen zu *Free* und zur Freigabe von Objekten finden Sie in Kapitel 2.3.2.

1.9.5 Ausgabekomponenten

Mit den schon am Anfang von Kapitel 1.9.4 genannten Einschränkungen und Ausnahmen lassen sich die folgenden Komponenten als Ausgabekomponenten zusammenfassen:

Mit einer *TImage*-Komponente (Seite ZUSÄTZLICH) können Sie auf einfache Weise Grafiken in Ihre Formulare aufnehmen. Diese Bilder können schon zur Entwurfszeit festgelegt werden (Property *Image*), sie können aber auch zur Laufzeit aus Dateien oder der Zwischenablage geladen werden, und zwar über die in Kapitel 4.5.3 beschriebene *TPicture*-Schnittstelle (das Property *Image* hat den Typ *TPicture*).

TShape (Seite ZUSÄTZLICH): Wenn Sie in einem Formular eine einfache grafische Form wie einen Kreis oder ein Rechteck benötigen, könnten Sie diese Komponente benutzen. Möglicherweise ist jedoch der sinnvollste Einsatzzweck dieser Komponente, die Attribute der Pinsel und Stifte kennen zu lernen, mit denen Sie auf einem *Canvas*-Objekt zeichnen können (diese Attribute sind in *TShape* in den Properties *Pen* und *Brush* einstellbar und können schon zur Entwurfszeit getestet werden).

Die schon bekannte *TLabel*-Komponente sei hier der Vollständigkeit halber noch einmal mit aufgeführt, denn sie wird häufig nicht nur zur Beschriftung und Bereitstellung eines Tastenkürzels, sondern auch zur Ausgabe von Text verwendet (*Caption*-Property, eventuell auch mehrzeilig, wenn *WordWrap=True*).

TStaticText (Seite ZUSÄTZLICH) hat die gleiche Aufgabe wie *TLabel* und wurde nur für einen Spezialfall der ActiveX-Programmierung geschaffen.

TProgressBar (Seite WIN32) ist eine Klasse für Fortschrittsbalken, die von Installationsprozeduren und anderen zeitaufwändigen Vorgängen bekannt sind. Normalerweise haben Sie bei dieser Komponente nicht mehr zu tun, als ein paar Properties für Äußerlichkeiten einzustellen und zur Laufzeit das *Position*-Property gemäß dem zu beobachtenden Fortschritt zu setzen.

TAnimate (Seite WIN32) kapselt das in Windows eingebaute Animations-Steuerelement, das selbstständig AVI-Dateien abspielt. Wenn Sie nicht neben den vielen Icons auch noch eigene Animationen für Ihre Programme entwerfen wollen, so können Sie mit diesem Steuerelement zumindest die Standardanimationen abspielen, die Windows z.B. beim Kopieren, Suchen oder Löschen von Dateien anzeigt (siehe Online-Hilfe zum Property *TAnimate.CommonAVI*).

TPaintBox (Seite SYSTEM) definiert eine leere Zeichenfläche, die Ihnen im *OnPaint*-Ereignis die Gelegenheit gibt, Grafik über das *Canvas*-Property der Paintbox auszugeben. Die *PaintBox* ist immer dann ein guter Ersatz für die Zeichenfläche des gesamten Formulars, wenn die grafische Ausgabe nur in einem Teil dieses Formulars stattfinden soll. Der TreeDesigner in Kapitel 5 nutzt eine solche *TPaintBox* für seine Grafikausgabe.

TListView (Seite Win32) wurde bereits unter den Listenkomponenten in 1.9.3 vorgestellt. Die nächste Version des Beispielprogramms wird Ihnen diese Komponente genauer vorstellen.

TTreeView (Seite Win32) ist eine Komponente zur Anzeige hierarchischer Strukturen nach der Art der Verzeichnisbäume in modernen Datei-Browsern wie dem Windows-Explorer. Ihr ist das Kapitel 3.6.3 gewidmet, einige weitere Anwendungsbeispiele finden Sie in Kapitel 8.4.3 und in den Add-Ons zur Delphi-IDE, die im Anhang beschrieben werden.

THeaderControl (Seite Win32) stellt einen Tabellenkopf dar, wie er beispielsweise in der Detailansicht der Fenster des Windows-Explorers verwendet wird. Im Property *Sections* können Sie schon zur Entwurfszeit die Titel und Breiten der einzelnen Spalten angeben, zur Laufzeit kann der Benutzer die Spalten dann per Drag&Drop vertauschen, Spaltenbreiten verändern und die Spaltenüberschriften als Schaltflächen bedienen (wodurch er die Ereignisse *OnSectionDrag*, *OnSectionTrack/OnSectionResize* und *OnSectionClick* auslöst).

In der Komponente *TListView* ist ein Header bereits eingebaut, so dass Sie diese Komponente wahrscheinlich selten einzeln benötigen. Diese Komponente kommt im vorliegenden Buch und in den zugehörigen Beispielprogrammen nicht einzeln (d.h. außerhalb eines ListViews) vor.

Bei *TStatusBar* (Seite WIN32) handelt sich um eine typische Statuszeile, die in einzelne Abschnitte aufgeteilt werden kann – so wie die Statuszeile des Delphi-Editors. Zwar können Sie Statuszeilen auch einfach mit gewöhnlichen Panels aufbauen, *TStatusBar* bietet jedoch erheblich mehr Komfort bei der Unterteilung der Leiste (siehe den Editor des Properties *TStatusBar.Panels*).

TImageList (Seite WIN32) ist eine nicht-visuelle Komponente, die in *ToolBars*, *ListViews* und *TreeViews* zur Verwaltung der Icons verwendet wird. Die nächste Version des Beispielprogramms wird eine solche Komponente für sein ListView verwenden.

TDrawGrid (Seite ZUSÄTZLICH) stellt die gleiche Tabellenstruktur zur Verfügung wie *TStringGrid*, überlässt es aber Ihrer Verantwortung, die Zellen mit Inhalt zu füllen. *TDrawGrid* und *TStringGrid* sind über die gemeinsame Elternklasse *TCustomGrid* eng verwandt. Ein Beispiel zur Grafikausgabe in einem *TDrawGrid* finden Sie am Ende von Kapitel 4.4.3.

Die ListView-Version des Beispielprogramms

Die letzte Version des Beispielprogramms, *Wecker3e* (Abbildung 1.26), verwendet drei der genannten Ausgabekomponenten: *TListView*, *TStatusBar* und *TProgressBar*: Das Listview ersetzt die bisher verwendete Listbox und bietet eine mehrspaltige Darstellung verschiedener Alarmdaten sowie die Darstellung von Symbolen für verschiedene Alarmtypen. Die Statuszeile dient dazu, längere Hinweistexte zu der Komponente anzugeben, über der sich die Maus gerade befindet. Der Fortschrittsbalken soll anzeigen, wie viel der Zeit zwischen dem letzten und dem nächsten Alarmereignis bereits abgelaufen ist.

Abbildung 1.26: Zwei der vier möglichen Ansichten der abschließenden Programmversion

TStatusBar

Um das Aussehen einer Statuszeile festzulegen, müssen Sie im Wesentlichen die Properties *SizeGrip* und *Panels* setzen. Wenn *SizeGrip* eingeschaltet ist, wird die rechte untere Ecke mit einer Hervorhebung versehen, die an eine Vergrößerbarkeit des Fensters erinnert, selbst wenn die Statuszeile gar nicht zur Größeneinstellung dient. *Panels* ist ein strukturiertes Property, das Sie über einen Listeneditor editieren, in dem Sie die Bereiche definieren können, in welche die Statuszeile unterteilt werden soll. Wenn Sie Bereiche definieren, können Sie den Text jedes Bereichs über das Property *TStatusBar.Panels[x].Text* setzen.

Um einen möglichst breiten Platz zur Anzeige von Hinweisen zu erhalten, definiert *Wecker3e keine* Bereiche, sondern verwendet eine aus einem einzigen Bereich bestehende Statuszeile, deren Text sich über das Property *SimpleText* setzen lässt. Um dieses wirksam zu machen, muss allerdings vorher noch das *SimplePanel*-Property auf *True* gesetzt werden.

Ein Beispiel für die Anwendung der Statuszeile: Wenn der Benutzer die Alarmfunktion manuell deaktiviert, zeigt das Programm eine Rückmeldung in der Statuszeile an:

```
StatusBar1.SimpleText := 'Alarmfunktion deaktiviert.';
```

Hinweistexte in der Statuszeile anzeigen *R57*

Der Haupteinsatzzweck der Statuszeile in *Wecker3e* liegt jedoch im Anzeigen der Hinweise für die einzelnen Komponenten. Es gibt sogar ein Ereignis, das jedes Mal auftritt, wenn die Maus über eine beliebige Komponente Ihrer Anwendung bewegt wird. Dieses Ereignis heißt *OnHint*, befindet sich aber im *Application*-Objekt, so dass Sie das Ereignis nicht über den Objektinspektor mit einer Methode verknüpfen können. In *Wecker3e* wird diese Verknüpfung statt dessen in der *OnCreate*-Methode programmgesteuert durchgeführt:

```
procedure TAlarmForm.FormCreate(Sender: TObject);
begin
  ...
  Application.OnHint := ApplicationHint;
end;
```

Die Methode *ApplicationHint* muss nun lediglich den Hinweistext der aktuellen Komponente, den die VCL im Property *Application.Hint* bereitstellt, in die Statuszeile schreiben:

```
procedure TAlarmForm.ApplicationHint(Sender: TObject);
begin
  StatusBar1.SimpleText := Application.Hint;
end;
```

Hinweisfenster allgemein

Normalerweise zeigt die VCL die Hinweistexte automatisch in den bekannten gelben Hinweisfenstern an, wenn Sie das *ShowHint*-Property des Formulars auf *True* setzen. Alle Komponenten dieses Formulars »erben« dann standardmäßig dieses *ShowHint*-Verhalten. (Diese Vererbung können Sie bei jeder Komponente unterbinden, indem Sie ihr *ParentShowHint*-Property auf *False* setzen.) In diesem Beispielprogramm wurde *ShowHint* abgeschaltet, damit die Hinweise nicht gleich doppelt angezeigt werden.

Optional können Sie im *Hint*-Property sogar zwei Versionen des Hinweistextes angeben. Sie trennen beide Versionen durch das Zeichen »|« (ANSI-Code 124, üblicherweise mit »<« und »>« auf einer Taste). Delphi nimmt an, dass Sie vor diesem Zeichen eine kurze Version des Hinweistextes angeben, die sich sehr gut für die Hinweisfenster eignet, und danach eine ausführlichere Version, die beispielsweise in einer Statuszeile angezeigt werden könnte und die in *Wecker3e* über das Property *Application.Hint* abgerufen wird. Geben Sie nur eine Hinweis-Version ein, so verwendet die VCL diese sowohl für die automatischen Hinweisfenster als auch für *Application.Hint*.

Hinweis: Sie können das *Hint*-Property auch dynamisch zur Laufzeit ändern. So können Sie beispielsweise im *OnMouseMove*-Ereignis einer Zeichenfläche das *Hint*-Property auf die aktuelle Mausposition setzen, damit in einer Statuszeile immer die aktuelle Mausposition angezeigt wird. Die TreeDesigner-Beispielanwendung aus Kapitel 5 gibt mit ihrer Statuszeile ein praktisches Beispiel hierfür.

TProgressBar

Das Beispielprogramm übernimmt das voreingestellte Aussehen des Fortschrittsbalkens. Veränderungen an Properties werden daher nur zur Laufzeit vorgenommen. Der durch den Balken zu beobachtende »Fortschritt« ist der Verlauf der Zeit zwischen zwei Alarmereignissen. Der kleinste vom Programm messbare Fortschritt ist damit die Zeit zwischen zwei *OnTimer*-Ereignissen. So verwendet das Programm auch die *OnTimer*-Methode, um den Balken zu aktualisieren. Der aktuelle Fortschritt wird im Property *Position* angegeben und versteht sich relativ zu Start- und Endwert, die in den Properties *Min* und *Max* angegeben werden können. Da im Beispiel *Min* und *Max* nicht geändert wurden, stehen die Werte zwischen 0 und 100 zur Verfügung, um den Fortschritt darzustellen. Die *OnTimer*-Methode muss also die bisher abgelaufene Zeit lediglich in Prozent umrechnen, den dabei herauskommenden Fließkommawert mit der Standardfunktion *trunc* in eine ganze Zahl runden und dem *Position*-Property zuweisen:

```
procedure TAlarmForm.TimerTimer(Sender: TObject);
var
  NextAlarm: TDateTime;
begin
  ...
```

```
NextAlarm:=GetNextAlarm;
if NextAlarm > -0.1 then
  ProgressBar1.Position :=
    trunc ( (Time-LastAlarm) /
            (NextAlarm-LastAlarm) * 100
          )
else ProgressBar1.Position:=0;
```

TListView

Die vom Windows-Explorer oder auch von Delphis DATEI | NEU-Dialogbox[7] bekannten möglichen Ansichten eines ListViews heißen im Windows-Explorer *Große Symbole*, *Kleine Symbole*, *Liste* und *Details*. In Delphi stellen Sie die Ansichtsart über das Property *TListView.ViewStyle* wie folgt ein:

- *VsIcon* ist der Standardmodus, in dem jedes Listenelement mit einem Icon dargestellt wird, normalerweise mit einem großen, aber das hängt davon ab, wie groß die Icons in Ihren Bilderlisten sind. In einer Delphi-Anwendung sind diese Icons standardmäßig nicht verschiebbar. Wie Sie das ändern können, zeigt ein anderes Beispielprogramm in Kapitel 3.6.2.
- Im Stil *vsSmallIcon* werden statt dessen kleine Icons angezeigt; auch hier vorausgesetzt, dass Sie die Bilderlisten entsprechend initialisiert haben. Die Beschriftung befindet sich in diesem Modus rechts von den Icons.
- Bei der Einstellung *vsList* werden die Elemente ähnlich wie im Modus *vsSmallIcon* dargestellt, nämlich mit denselben Icons und derselben Art der Beschriftung; allerdings werden sie in festen Spalten angeordnet und können nicht mehr verschoben werden.
- Die vierte Darstellungsweise ist *vsRecord* (in der Windows-Shell unter dem Namen *Details* kursierend). Sie sieht so aus wie *vsList*, mit dem Unterschied, dass es nur eine Spalte von Elementen gibt, dafür aber pro Element mehrere Spalten von Informationen. Diese Spalten werden nach Art der Komponente *THeaderControl* beschriftet.

TListView im Beispielprogramm

Da die »Listen-Ansichten« seit ihrer Einführung im Explorer von Windows 95 eine sehr weite Verbreitung gefunden haben, soll das Beispielprogramm zum Abschluss auch noch durch ein solches Element aufgewertet werden. In Kapitel 3.6.2 werden Sie eine ausführlichere Besprechung dieser Komponente finden, an dieser Stelle geht es um einen ersten Einstieg in die Praxis mit dieser Komponente, wobei noch einige Merkmale von *TListView* außer Acht gelassen werden.

7 In diesem Dialog können Sie per lokalem Menü zwischen den vier Ansichtsarten wechseln.

Vor dem Schreiben von Code müssen die folgenden Vorbereitungen getroffen werden, um ein ListView zu erhalten, das wie im Beispielprogramm funktioniert:

- Zuerst gilt es, ein paar passende Bildsymbole zu finden. Wenn diese nicht aus Delphis MDI-Schablone oder anderen frei verwendbaren Vorlagen wie etwa denen aus dem Bilderordner von Delphi (BORLAND\GEMEINSAME DATEIEN\IMAGES) übernommen werden können, können Sie sich in Delphis Bildeditor oder mit jedem anderen Grafikprogramm, das BMP-Dateien erzeugen kann, eigene Bitmaps herstellen. Da Icons für die Funktion einer Listenansicht nicht *unbedingt* erforderlich sind, können Sie diesen Schritt natürlich auch überspringen oder zumindest auf später verschieben.
- Diese Icons werden dann unter Delphi in einer *TImageList*-Komponente zusammengefasst. Nachdem Sie eine solche Komponente in das Formular eingefügt haben, rufen Sie mit einem Doppelklick darauf einen Dialog auf, in dem Sie die Bilder aus BMP-Bilddateien laden können.
- Um die kleinen Versionen der Icons zu erhalten, wurden die großen Icons des Beispielprogramms einfach durch die Verkleinerungsfunktion des Micrografx Picture Publishers geschleust und unter neuem Namen abgespeichert. Diese Icons wurden dann in eine zweite Bilderliste aufgenommen.
- Nun können Sie die *TListView*-Komponente in das Formular einfügen und über das Property *LargeImages* mit der *TImageList* der großen Bilder, über *SmallImages* mit der anderen Liste verknüpfen.
- Auch die Tabellenspalten für die Report-Ansicht lassen sich bequem schon zur Entwurfszeit einfügen. Rufen Sie dazu im Objektinspektor beim *Columns*-Property des ListViews den Spalteneditor auf. Für das Beispielprogramm wurden die vier Spalten *Zeit*, *Aktion*, *Zu startendes Programm* und *Kommentar* erzeugt. Für jede Spalte lassen sich neben der Beschriftung einige weitere Properties einstellen, was im Beispielprogramm zur Einstellung der Spaltenbreite genutzt wurde.
- Um die Tabellenspalten schon zur Entwurfszeit überprüfen zu können, können Sie das ListView-Property *ViewStyle* auf *vsReport* setzen. Damit das Beispielprogramm trotzdem immer in der Ansicht der großen Icons gestartet wird, setzt es dieses Property beim *OnCreate*-Ereignis wieder auf *vsIcon* zurück.

Hinweis: Wenn Sie auf die Darstellung mit großen Icons verzichten, brauchen Sie nur *SmallImages* anzugeben; beschränken Sie sich auf die großen Icons und verzichten auf die drei verschiedenen Ansichten, in denen die kleinen Icons verwendet werden, lassen Sie statt dessen die *SmallImages* weg.

Elemente hinzufügen

Die Verwaltung der einzelnen Listeneinträge läuft in einem ListView grundlegend verschieden von der Verwaltung der Strings in einer ListBox ab. Im ListView wird jeder Eintrag durch ein eigenes Objekt repräsentiert, in dem alle wichtigen Daten zusammengefasst sind, die in einer Listbox noch vereinzelt vorliegen (so müssen Sie sich in einer ListBox beispielsweise die zum *i*-ten Eintrag gehörenden Daten aus folgenden Quellen zusammensuchen: *Selected[i]*, *Items[i]* und *Items.Objects[i]*, in *TListView* finden Sie sie als *Items[i].Selected*, *Items[i].Caption* und *Items[i].Data*).

Ähnlich wie bei *TListBox.Items.Append* erzeugen Sie mit *TListView.Items.Add* einen neuen Eintrag. Der wesentliche Unterschied ist, dass *Add* noch nicht den Inhalt des Eintrages festlegt, sondern nur ein *TListItem*-Objekt zurückliefert:

```
// siehe Beispielprogramm: Methode TAlarm.AddToListView
var
  NewItem: TListItem;
begin
  NewItem := ListView.Items.Add;
```

Dieses *TListItem*-Objekt verfügt nun seinerseits wieder über eine Reihe von Properties, über die Sie den Inhalt des Eintrags festlegen können, der nicht nur aus dem Beschriftungstext besteht, sondern auch aus dem Bildindex (die Bilderlisten werden ja schon in *TListView*-Properties angegeben, müssen hier also nicht nochmals gesetzt werden) und den Informationen, die in den Spalten der Report-Ansicht angezeigt werden sollen:

```
// siehe Beispielprogramm: Methode TAlarm.UpdateListItem
  NewItem.Caption := TimeToStr(Alarmzeit); // Text des Eintrages
  if Programm then // Index des Bildsymbols
    item.ImageIndex := 0
  else item.ImageIndex := 1;
  NewItem.SubItems.Clear; // Text für die einzelnen Spalten
  NewItem.SubItems.Add(GetEreignisText);
  NewItem.SubItems.Add(ProgrammPfad);
  NewItem.SubItems.Add(Kommentar);
```

Um den Zusammenhang zum Beispielprogramm darzustellen, hier noch ein Ausschnitt aus der *OnClick*-Methode für den Schalter EDITIEREN:

```
NewData := TAlarm.Create;
...
if AlarmEditDlg.ShowModal = mrOk then begin
  AlarmEditDlg.GetAlarmData(NewData);
  NewData.AddToListView(ListView1)
end else
  NewData.Free;
```

Um diese Zeilen vollständig verstehen zu können, sollten Sie die *TAlarm*-Klasse kennen, die in Kapitel 1.9.4 eingeführt wurde. Sie eignet sich hervorragend, um die Initialisierung eines Listeneintrages durchzuführen. *TAlarm.UpdateListItem* schreibt alle Alarmdaten in einen bestehenden *TListItem* und *TAlarm.AddToListView* erzeugt zusätzlich einen neuen Eintrag. Die entscheidenden Ausschnitte aus diesen beiden Methoden wurden oben abgedruckt.

Verknüpfung der Einträge mit Programmobjekten

Wie in der letzten Programmversion sollen auch in *Wecker3e* die einzelnen *TAlarm*-Objekte bei den zugehörigen Listeneinträgen gespeichert werden, damit sich die Daten später wieder so leicht in die Dialogbox zum Editieren kopieren lassen wie in Kapitel 1.9.4 gezeigt. Zu diesem Zweck kommt noch eine weitere Zuweisung an das *TListItem*-Objekt hinzu:

```
// siehe Beispielprogramm: Methode TAlarm.AddToListView
  NewItem.Data := self;
```

Diese Verknüpfung kann unter anderem dazu verwendet werden, um in der *OnTimer*-Methode die einzelnen Alarmdaten abzufragen:

```
with ListView1 do begin
  // Eine neu erreichte Alarmzeit suchen:
  for i := 0 to Items.Count-1 do begin
    TestAlarm := TAlarm(Items[i].Data).Alarmzeit;
    if IsTimeOver(TestAlarm) then begin
      LastAlarm := TestAlarm;
      AlarmAction(TAlarm(Items[i].Data));
    end;
  end;
```

Die Typumwandlung *TAlarm(Items[i].Data)* ist ebenfalls schon aus *Wecker3d* bekannt. Auch die Methoden zum Editieren eines Elements und zum Speichern der Daten in der Windows-Registry müssen bei einer Umstellung von *TListBox* auf *TListView* nur geringfügig angepasst werden. Wenn Sie eine solche Umstellung in einer Ihrer Anwendungen durchführen wollen, können Sie auch die Tabelle in Kapitel 1.9.3 zu Hilfe nehmen.

Editieren der Einträge *R51*

Ein ListView kann auch als Eingabekomponente verwendet werden. Solange Sie sein *ReadOnly*-Property nicht auf *True* setzen, wird die Editierfunktion automatisch von der Komponente bereitgestellt: Um einen Eintrag zu editieren, muss der Benutzer ihn mit der rechten Maustaste anklicken.

Hiermit ist es aber nicht getan, denn die Beschriftung der Elemente im ListView des Beispielprogramms gibt ja die Alarmzeit an, die jedoch eigentlich an einer anderen

Stelle gespeichert ist, und zwar in einem *TAlarm*-Objekt. Damit das Editieren im ListView Sinn macht, muss das Programm also nach jeder Umbenennung auch die mit dem umbenannten Eintrag gespeicherte Alarmzeit aktualisieren.

Natürlich gibt es für das Umbenennen eines Eintrages ein spezielles Ereignis, es heißt *OnEdited* und wird wie folgt bearbeitet:

```
procedure TAlarmForm.ListView1Edited(Sender: TObject; Item: TListItem;
  var S: String);
begin
  try
    TAlarm(Item.Data).Alarmzeit := StrToTime(S);
  except
    on E: EConvertError do
      S := TimeToStr(TAlarm(Item.Data)).Alarmzeit;
  end;
end;
```

Der editierte Eintrag muss nicht aus der *Items*-Liste herausgesucht werden, denn er wird der Methode direkt im Parameter *Item* übergeben. *S* enthält nahe liegenderweise den neuen Text für den Eintrag. Er ist als *var*-Parameter gehalten, so dass das Programm ihn noch einmal verändern kann.

In diesem Fall wird einfach das mit dem *Item* verknüpfte *TAlarm*-Objekt erfragt und dessen *Alarmzeit*-Variable auf den neuen Wert gesetzt. Dafür muss allerdings der editierte String in eine Zeitangabe umgewandelt werden, was wieder zu einem Konvertierungsfehler führen kann. Damit nun eine mögliche *EConvertError*-Exception nicht den Prozess des Editierens stört, fängt das Programm diese Exception ab. Es zeigt aber keine Fehlermeldung an, sondern sorgt dafür, dass die Umbenennung des Eintrags nicht durchgeführt wird. Dazu weist es dem *var*-Parameter *S* einfach wieder die bisherige Alarmzeit zu (umgewandelt in einen String).

Typische Merkmale eines Utilities

Als letztes Beispielprogramm dieser Serie wurde *Wecker3e* dazu auserwählt, zwei Funktionen zu demonstrieren, die in vielen derartigen Utility-Programmen verwendet werden:

- Unter der Annahme, dass es wenig sinnvoll ist, das Programm mehrmals gleichzeitig laufen zu lassen, bringt es bei einem zweiten Startversuch lediglich die erste, bereits laufende Instanz in den Vordergrund. Um festzustellen, ob es bereits gestartet wurde, verwendet es die Windows-API-Funktion *FindWindow*.
- Um dem Benutzer eine möglichst wenig Platz in Anspruch nehmende Information über seinen aktuellen Status zu liefern, zeigt es in der Task Notification Area der Windows-Taskbar (also in dem Bereich, in dem normalerweise auch die Uhrzeit angezeigt wird) ein Icon an.

Während hier für Details zur erstgenannten Funktion auf den Projektquelltext von *Wecker3e* verwiesen sei, befassen wir uns im Folgenden genauer mit der Einbindung des Icons in die TNA.

Einbinden von Icons in die Exe-Datei *R89*

Um gleich all das notwendige Werkzeug für die Anzeige des Icons zur Hand zu haben, sei als Erstes die Einbindung eines Icons in die EXE-Datei beschrieben. Hierzu wurde in Delphis Bildeditor eine Ressourcen-Datei im RES-Format und darin eine Icon-Ressource erstellt. Das Icon wurde unter dem Namen »ICONEIN« gespeichert, die Ressourcendatei wie in Kapitel 1.4.8 beschrieben mit der folgenden Anweisung zur EXE-Datei hinzugebunden:

```
{$R Icons.RES}
```

Im Programmcode (*OnCreate*-Ereignis des Formulars) kann aus dieser Ressource dann wie folgt ein *TIcon*-Objekt gemacht werden:

```
IconEin := TIcon.Create; // Weitere Details zu TIcon in Kapitel 4.5.1.
IconEin.Handle := LoadIcon(hInstance, 'ICONEIN');
```

Das Objekt *IconEin* wird im Folgenden an die Methode *ShowIcon* des Beispielprogramms übergeben und auf diese Weise in die TNA befördert.

Anzeige des Icons in der Task Notification Area *R163*

Das Beispielprogramm soll in der TNA anzeigen, ob die Alarmfunktion ein- oder ausgeschaltet ist. Ein Doppelklick auf das Icon soll außerdem zwischen den verschiedenen Zuständen umschalten. Während für das Icon lediglich der Aufruf einer API-Funktion erforderlich ist, muss das Programm zur Bearbeitung des Doppelklicks eine Nachrichtenbearbeitungsmethode bereitstellen, die allerdings nicht auf einem Delphi-Event, sondern auf einer Windows-Nachricht basiert. Sie wird wie folgt in der Formularklasse deklariert:

```
procedure WM_IconCallback(var Msg: TMessage); message MSG_IconCallback;
```

Dabei ist *MSG_IconCallback* eine selbst definierte Nachrichtennummer, im Beispielprogramm wie folgt festgelegt:

```
const // Die Nachrichten von WM_APP OxBFFF dürfen innerhalb einer...
  MSG_IconCallback = WM_APP + 100; // ...Anwendung frei verwendet werden.
```

Weitere Informationen über das Abfangen von Windows-Nachrichten gibt Kapitel 6.4.1 unter der Überschrift *Message-Methoden*.

Zur Anzeige des Icons dient die Shell-API-Funktion *Shell_NotifyIcon*, die mit einer Datenstruktur des Typs *TNotifyIconData* aufgerufen wird, welche im Beispielprogramm innerhalb der Methode *ShowIcon* initialisiert wird:

```
procedure TAlarmForm.ShowIcon(
  I: TIcon;      // VCL-Icon-Objekt, wird wie unten gezeigt geladen.
  Id: Integer;   // Kenn-Nummer des Icons, im Beispielprogramm immer 0
  Hint: String); // "Alarmfunktion einschalten" bzw. "... ausschalten"
var
  d: TNotifyIconData; // Datenstruktur des Windows-API
begin
  d.cbSize := sizeof(d); // Größe der Datenstruktur (zur Kontrolle)
  d.Wnd := Handle; // Handle des Fensters (siehe Kapitel 3.3.6)
  d.uID := Id; // Kenn-Nummer, im Beispielprogramm immer 0
  d.uCallbackMessage := MSG_IconCallBack; // Selbstdef. Nachricht
  d.hIcon := I.Handle;    // Handle des Icons
  strcopy(d.szTip, PChar(Hint)); // Maximal 63 Zeichen sind erlaubt.
  // Die folgenden Flags teilen der Shell mit, dass sie die optionalen
  // Felder d.uCallbackMessage, d.hIcon und d.szTip beachten soll:
  d.uFlags := NIF_MESSAGE or NIF_ICON or NIF_TIP;
  Shell_NotifyIcon(NIM_ADD, @d); // Aktion durchführen
end;
```

Beim Einschalten der Alarmfunktion wird diese Funktion beispielsweise wie folgt aufgerufen:

```
ShowIcon(IconEin, 0, 'Alarm ausschalten');
```

Wenn der Benutzer nun irgendwelche Mausaktionen über dem Icon in der TNA durchführt, wird *WM_IconCallback* aufgerufen. Das Beispielprogramm fragt ab, ob es sich um einen Doppelklick der linken Maustaste handelt, und schaltet gegebenenfalls die Alarmaktivierung um:

```
procedure TAlarmForm.WM_IconCallback(var Msg: TMessage);
begin
  if Msg.lParam = WM_LBUTTONDBLCLK then begin
    Alarmaktiviert1.Checked := not Alarmaktiviert1.Checked;
    ShowAlarmStatus;
  end;
end;
```

Zum Ausschalten des Icons wird ebenfalls *Shell_NotifyIcon* aufgerufen, allerdings mit der Kennung *NIM_DELETE* als erstem Parameter. Der *TNotifyIconData*-Record muss nur in drei Feldern initialisiert werden, und zwar mit den gleichen Werten wie zuvor:

```
d.cbSize := sizeof(d);
d.Wnd := Handle;
d.uID := Id;
Shell_NotifyIcon(NIM_DELETE, @d);
```

1.9.6 Komponenten zu Gestaltung und Strukturierung

Viele Komponenten dienen weder zur Ein- noch zur Ausgabe, noch dazu, eine Aktion auszulösen wie die Aktionsschalter, sondern sie haben eine rein gestalterische Funktion oder sind dazu gedacht, den Formularaufbau zu strukturieren:

TGroupBox hat die Aufgabe, mehrere Komponenten in einer Gruppe zusammenzufassen. Dabei bewirkt sie sowohl eine optische Gliederung, die dem Benutzer des Programms zugute kommt, als auch eine technische Gliederung, die dem Entwickler die Übersicht zu wahren hilft. *TGroupBox* ist ein Standardsteuerelement von Windows, das kaum Spielraum zu Variationen gibt, im Gegensatz zu *TPanel*. Tiefer gehende Details zu gruppierten Steuerelementen finden Sie in Kapitel 3.5.1.

TPanel ist eine von Borland programmierte Komponente, die Ihnen in den Properties *BevelInner, BevelOuter* und *BevelWidth* erlaubt, eine Umrandung ein- und abzuschalten sowie die Art und Dicke der Umrandung zu beeinflussen. Dafür gibt es bei einem *Panel*-Element nicht die von der *GroupBox* bekannte Möglichkeit, eine Überschrift anzugeben (die Beschriftung eines Panels wird vertikal zentriert ausgegeben). Ansonsten hat ein *Panel* die gleiche Funktion wie eine *GroupBox*.

TBevel (auf der Seite ZUSÄTZLICH) gibt Ihnen ähnliche Gestaltungsfunktionen wie *TPanel* (allerdings werden diese hier mit den Properties *Shape* und *Style* gesteuert) und kann zum Zeichnen von Trennlinien im Formular verwendet werden. *TBevel* gliedert das Formular aber *nur* optisch, eine Gruppierung von mehreren Steuerelementen zu einer im Programm besser handzuhabenden Einheit findet nicht statt. Der Vorteil davon ist, dass ein *TBevel* weniger Ressourcen des Betriebssystems verbraucht.

TTabControl (Seite WIN32) ist eine vielseitig verwendbare Komponente zum Umschalten zwischen Dateien, Dateibereichen, Seiten, zwischen verschiedenen Ansichten einer Seite oder was auch immer Sie über die angedeuteten Registerzungen dieser Komponente zugänglich machen wollen. Sie wird zusammen mit verwandten Komponenten in Kapitel 3.5.4 ausführlich behandelt.

TPageControl (in der Palette neben *TTabControl*) ermöglicht Ihnen die Gestaltung mehrseitiger Dialoge und entspricht einem *TTabControl* mit zusätzlicher Unterstützung mehrerer Seiten. Das Hinzufügen und Editieren von neuen Seiten ist zur Entwurfszeit auf eine sehr komfortable Weise möglich (siehe Kapitel 3.5.4).

TTabSet (Seite WIN 3.1) ist eine von Windows unabhängige, schon von Delphi 1 eingeführte Alternative zu *TTabControl*.

TNotebook (Seite WIN 3.1) ist eine zur Laufzeit unsichtbare Komponente, deren Aufgabe die Verwaltung mehrerer Seiten ist. Der Entwurf der Seiten im Formulardesigner läuft ähnlich ab wie bei einem *TPageControl*, ein Anwendungsbeispiel finden Sie in Kapitel 3.5.4.

Die Komponente *TTabbedNotebook* (Seite WIN 3.1) vereinigt die Funktion von *TNotebook* und *TTabSet* unter einer anderen optischen Gestaltung. Unter 32-Bit-Windows wird statt dessen üblicherweise die Komponente *TPageControl* verwendet. Zum schnellen Austauschen von *TTabbedNotebook*-Komponenten durch *TPageControls* (oder umgekehrt) siehe Kapitel 3.5.6.

TScrollBox (Seite STANDARD) dient dazu, eines oder mehrere Kindelemente, die eine größere Fläche beanspruchen, als der ScrollBox auf dem Bildschirm zur Verfügung steht, zu verwalten und zu scrollen. Die Komponente erzeugt automatisch Bildlaufleisten, falls die Kindelemente in horizontaler bzw. vertikaler Richtung mehr Platz benötigen. Auch das Scrollen wird von dieser Komponente automatisch abgewickelt. Außerdem können Sie zwischen normalen dreidimensionalen Bildlaufleisten und solchen im allerneuesten flachen Design wählen.

TPageScroller (Seite WIN32, ab Delphi 4) stellt eine praktische Scrollfunktion für schmale Komponenten dar, die nur in einer Richtung gescrollt zu werden brauchen. Um einen Eindruck von dieser Komponente zu bekommen, reduzieren Sie einfach die Breite des Menüfensters der Delphi-IDE so weit, dass nicht mehr alle Komponenten der Komponentenpalette sichtbar sind. Die standardmäßig versteckten Pfeilsymbole der *PageScroller*-Komponente treten dann zum Vorschein und ermöglichen Ihnen den Zugriff auf unsichtbare Komponenten.

Mit der *TScrollBar*-Komponente (Seite STANDARD) können Sie auch eine einzelne Bildlaufleiste in Ihr Formular aufnehmen und selbst festlegen, was durch sie gesteuert werden soll. Früher wurden solche Komponenten häufig zur Einstellung von Werten verwendet, für diesen Zweck ist heute aber die *TTrackBar*-Komponente eher zu empfehlen.

Wenn Fenster aus mehreren Bereichen bestehen, werden diese Bereiche häufig durch verschiebbare Balken getrennt, wie zum Beispiel in der Delphi-IDE die Fenster, die aneinander angedockt werden. Für solche Zwecke ist die *TSplitter*-Komponente vorgesehen (Seite ZUSÄTZLICH), allerdings sind bei ihrer Verwendung Besonderheiten zu beachten, die in Kapitel 3.3.1, R3 (Seite 997) näher beschrieben sind.

Diese Übersicht soll an dieser Stelle genügen. Das Beispielprogramm *Wecker3e* und seine Vorgänger haben ja bereits Komponenten der Klassen *TPanel* und *TGroupBox* verwendet, bieten jedoch noch kaum ein Betätigungsfeld für die anderen genannten Gestaltungskomponenten. Daher werden diese im Falle der Seitenauswahlkomponenten an einem eigenen Beispielprogramm im bereits erwähnten Kapitel 3.5.4 behandelt. Weitere Details zu *TScrollBox* finden Sie in Kapitel 3.4.1, Kapitel 5.5.4 beschreibt technische Details zum Scrolling der TreeDesigner-Beispielanwendung, und in Kapitel 6.5.2 wird *TScrollBox* als Basis für eine selbst definierte Komponentenklasse verwendet.

2 Die Sprache Object Pascal

Dieses Kapitel beschäftigt sich mit der Sprache Object Pascal, auf der Delphi aufbaut. Es legt besonderen Wert auf die neueren und objektorientierten Aspekte dieser Sprache, behandelt aber auch alle wichtigen anderen Eigenschaften, insbesondere natürlich alle Eigenschaften, die in diesem Buch verwendet werden.

Object Pascal ist eine Nachfolgesprache von »Borland Pascal mit Objekten«, die 1989 mit Turbo-Pascal 5.5 eingeführt wurde, einer Entwicklungsumgebung, die noch unter DOS lief. Wenig später machte die Sprache in »Turbo Pascal für Windows« Bekanntschaft mit einem grafischen Desktop und 1995 schließlich wurde sie durch umfangreiche Erweiterungen und auch Änderungen fit für ein neues Produkt namens »Delphi« gemacht. Bis Delphi 4 fügte Borland mit jeder Delphi-Version umfangreiche sprachliche Erweiterungen hinzu:

- Delphi 1 brachte die bisher grundlegendsten Veränderungen, denn es führte das derzeit vorherrschende Objektmodell (erkennbar am Schlüsselwort *class* im Gegensatz zu *object*, siehe Kapitel 2.2 und 2.3) sowie die Properties und Exceptions ein (Kapitel 2.2.5 bzw. 2.6). Der Name der Sprache wurde von Borland Pascal with Objects auf Object Pascal geändert.
- Delphi 2 stand ganz im Zeichen der Umstellung des Compilers von 16- auf 32-Bit, die meisten Änderungen fanden im Bereich der Typen statt (Kapitel 2.4). Außerdem »legalisierte« es initialisierte Variablen (Abschnitt 2.1.5) und führte neue Aufrufkonventionen für Prozeduren und Funktionen ein (Abschnitt 2.5.2).
- Die wichtigste Neuerung in Delphi 3 waren die Interfaces (Kapitel 2.7), ergänzt von den für die Fehlersuche vorgesehenen Assertions (Kapitel 1.7.6) und weiteren Details wie neuen Aufrufkonventionen und Typen.
- Zu den Spracherweiterungen von Delphi 4 gehören das Überladen von Funktionen, Prozeduren und Methoden, die Definition von Standardwerten für ihre Parameter (zu beidem siehe Kapitel 2.5.4) und die dynamischen Arrays (Kapitel 2.4.3). Auch die schon von Delphi 3 bekannte Unterstützung der COM-Interfaces wurde noch weiter verfeinert (siehe *Delegierte Implementierung* in Kapitel 2.7.5). Schließlich lässt es sich auch Delphi 4 nicht nehmen, ein paar neue Typen einzuführen (Kapitel 2.4).

- Nachdem Object Pascal in Delphi 4 einen sehr hohen Entwicklungsstand erreicht hat, blieb der Sprachumfang in den folgenden Versionen weitgehend stabil. Erst in Kylix und Delphi 6 kamen einige kleinere Erweiterungen hinzu, etwa neue Formen der bedingten Kompilierung und neue Compiler-Direktiven (siehe Kapitel 2.1.3; beides ist für die Plattform-übergreifende Programmierung nützlich) sowie neue *$A*-Optionen für das Ausrichten von Datenfeldern und eine erweiterte Syntax für Aufzählungstypen (siehe Kapitel 2.1.3 und 2.4.1; beides ist für das Zusammenspiel mit C/C++-Code wichtig und wurde offenbar vor allem unter Linux benötigt). Außerdem gibt es sehr spezielle Neuerungen wie selbst definierte Varianten und einen neuen eingebauten Assembler, die in diesem Kapitel aber nicht genauer untersucht werden (siehe Online-Hilfe).

Wir beginnen in diesem Kapitel nicht im Kleinen bei den Typen, Variablen und Anweisungen von Object Pascal, sondern im Großen bei der objektorientierten Sprachstruktur, mit der Sie bei der Verwendung von Delphi schon von Anfang an konfrontiert werden, zu allererst jedoch mit einem Überblick.

2.1 Überblick

Bevor wir richtig in die Welt der Objekte einsteigen, soll dieses Kapitel einen Überblick über das Modulkonzept und über einige Kleinigkeiten wie Schlüsselwörter, Kommentare und andere lexikalische Elemente geben.

2.1.1 Object Pascal für Umsteiger

Dieses Kapitel enthält kurze Vergleiche von Object Pascal und Sprachen, die mächtiger sind als Object Pascal (C++) oder weniger mächtig (Basic-, Datenbank- und Makrosprachen). Für Umsteiger von Sprachen der letztgenannten Gruppe beschreibt es kurz die Konzepte, die Object Pascal diesen Sprachen voraus hat. Falls Sie einen informellen Überblick über die Möglichkeiten suchen, die Sie innerhalb einer Ereignisbearbeitungsmethode haben, lesen Sie bitte das Kapitel 1.5.2.

Wenn Sie sich bereits mit der objektorientierten Programmierung in Turbo- oder Borland Pascal auskennen, benötigen Sie von diesem zweiten Kapitel wahrscheinlich nur die Abschnitte, in denen es um Neuerungen geht. Im folgenden Vergleich zwischen Object Pascal und C++ werden zahlreiche dieser Neuerungen erwähnt. Einige Neuerungen, die speziell für die Programmierung von Komponenten gedacht sind, werden erst in Kapitel 6 besprochen.

Object Pascal und C++

Object Pascal enthält gegenüber Borland Pascal einige Neuerungen, die speziell für C++-Programmierer (und aufgrund der Ähnlichkeit der Sprache zu C++ natürlich auch für Java-Programmierer) interessant sind:

- virtuelle Konstruktoren, mit denen Object Pascal gegenüber C++ auf dem Gebiet der Polymorphie einen interessanten Vorteil besitzt (Kapitel 2.3.4),
- Methodenzeiger, die praktischer und effektiver sind als die Methodenzeiger in C++ (Kapitel 2.5.5),
- Exceptions, die den Exceptions im alten C-Stil entsprechen – ein Vorteil gegenüber den C++-Exceptions ist die *finally*-Anweisung (die in C++ wegen der automatischen Destruktoraufrufe natürlich nicht wirklich benötigt wird) (Kapitel 2.6),
- Typinformationen, die über die RTTI von C++ hinausgehen (Kapitel 2.3.6),
- offene Array-Konstruktoren, mit denen Sie praktisch variable Parameterlisten erhalten (Kapitel 2.5.3),
- Neben den Schutzebenen *private* und *public* kennt Object Pascal nun auch die *protected*-Ebene. Sogar zur *friend*-Deklaration von C++ gibt es eine ähnliche Alternative: Alle Klassen innerhalb einer Unit sind »friends«, die Schutzbestimmungen werden vom Compiler nur zwischen verschiedenen Modulen kontrolliert (Kapitel 2.2.6).
- C++-Programmierer, die schlechte Erfahrungen mit der Mehrfachvererbung gemacht haben, finden vielleicht in den Interfaces von Object Pascal eine saubere Lösung für ihr Programmierproblem. Wie die Interfaces der Sprache Java erreichen sie viele Ziele, die man in C++ nur mit Mehrfachvererbung erreichen konnte, sind aber frei von den großen Risiken dieser C++-Spracheigenschaft (Kapitel 2.7).

Auch wenn Object Pascal einiges zu C++ aufgeholt hat, bleiben einige grundsätzliche Vorteile von C++, die wahrscheinlich auch in Zukunft nicht in Object Pascal ausgeglichen werden können, da dazu die Grundfesten von Object Pascal erschüttert werden müssten. So müssen Sie in Object Pascal des Öfteren zu umständlichen Ausdrucksweisen greifen. Object Pascal enthält manche Schlüsselwörter, die in C++ gar nicht benötigt werden, da sich der Sinn dort von alleine ergibt (z.B. *then*, *var*, *type*, *to* in einer *for*-Schleife), und Operatoren und andere in C++ aus ein oder zwei Zeichen bestehende Symbole beanspruchen in Pascal ein ganzes Schlüsselwort (z.B. *begin*, *end*, *div*, *mod*).

Um die Verzichtsliste noch eine Weile fortzusetzen: In Object Pascal gibt es außerdem keinen Zuweisungsoperator, der flexibel wie jeder andere Operator einsetzbar ist, keinen »?:«-Operator, keinen Methoden-Code in der Klassendeklaration (Inline-Methoden), keine automatischen Konstruktoren/Destruktoren, keine Makros, keine selbst überladbaren Operatoren, keine Mehrfachvererbung von Klassen und auch keine Templates.

Neben den am Anfang schon genannten Highlights von Object Pascal gibt es übrigens noch einige weitere Vorteile, die schon etwas älter sind: Mengen (Kapitel 2.4.5), offene Arrays (Kapitel 2.5.3) und der in diesem Buch wahrscheinlich am häufigsten verwendete Vorteil von Pascal gegenüber C++, die *with*-Anweisung (siehe Kapitel 2.5.1).

Umsteiger von Basic- und Makrosprachen

Wenn Sie bisher mit Visual Basic, einem anderen Basic oder einer Makrosprache gearbeitet haben, werden Sie in Delphi auf viele konzeptuelle Unterschiede treffen, die im Prinzip immer dieselben Ziele haben: die Erstellung und Wartung größerer Programme zu vereinfachen sowie deren Lesbarkeit und Überschaubarkeit zu verbessern. Dabei spielen verschiedene Konzepte wie das Aufteilen einer Programmieraufgabe in überschaubare und möglichst unabhängige Einheiten, das Verbergen von Informationen (Kapselung, Information Hiding) und das Wiederverwenden von Code eine Rolle, die hier jedoch nicht in Einzelheiten erläutert werden können.

Module

Object-Pascal-Module (*Units*) unterscheiden sich von den Modulen vieler anderer Sprachen durch ihre besondere Eigenständigkeit. Während Module in Visual Basic mit dem Formular verschmelzen und Dateien in C++ auf die verschiedensten Arten zusammengebunden werden können, ist eine Unit in Object Pascal ein klar definierter und von Formularen unabhängig gespeicherter Baustein. Der gesamte Quelltext, den Sie in Delphi schreiben, inklusive aller Ereignisbearbeitungsmethoden, wird in Units gespeichert, die im normalen Textfformat vorliegen und daher nach außen völlig offen sind. Der Aufbau eines Formulars wird in einer eigenständigen, wahlweise ebenfalls im Textformat speicherbaren `.dfm`-Datei abgelegt, die Sie auch in lesbarer Form in den Quelltexteditor laden können.

Ein weiterer Vorteil von Units ist ihre klare Schnittstellendefinition. Im *interface*-Teil einer Unit werden alle nach außen hin sichtbaren Elemente der Unit deklariert, alles andere bleibt in der Unit verborgen. Die Einbindung einer Unit findet lokal statt, d.h. eingebundene Units sind nur in den Units sichtbar, in denen sie auch eingebunden werden.

Typen

Besonders wichtig ist, dass Pascal-Variablen immer zuerst deklariert werden müssen, bevor sie verwendet werden. Bei dieser Deklaration müssen Sie sich auf einen bestimmten Variablentyp festlegen, allerdings haben Sie mit untypisierten Variablenparametern oder untypisierten Zeigern auch in Object Pascal sehr große Flexibilität. Schließlich findet die Deklaration von Variable in einem (oder mehreren) eigenen *var*-Blöcken statt, die klar vom Programmcode abgetrennt sind, was zwar unbequem sein kann, aber sehr zur Übersichtlichkeit beiträgt.

Unterschiede im Quelltext

Die Übersichtlichkeit der Pascal-Quelltexte erleichtert das Erlernen der Sprache in besonderer Weise: Zwischen verschiedenen Programmteilen findet eine klare Trennung statt. So besteht eine Unit aus den beiden Teilen *Interface* und *Implementation*, innerhalb derer sich wieder kleinere Blöcke befinden, die mit eigenen Überschriften versehen sind (*var*, *type*, *const*). Im Implementationsteil kommen schließlich die Methoden (Prozeduren und Funktionen) hinzu, deren Aufbau wieder ähnlich strukturiert ist: Zuerst folgen *var*-, *type*- und *const*-Abschnitte, dann der in *begin* und *end* eingeschlossene Programmcode.

Auch innerhalb des Quelltextes ist (Object) Pascal sehr strukturiert. Anweisungsblöcke von Funktionen, Prozeduren und Methoden werden immer durch *begin* und *end* umgeben, bei vielen Kontrollstrukturen wie *if...then*, *while*- und *for*-Schleifen usw. verwenden Sie ebenfalls *begin* und *end*, um mehrere Anweisungen zusammenzufassen. Schließlich gibt es im Aufruf von Prozeduren und Funktionen keinen syntaktischen Unterschied – bei beiden werden die Parameter in Klammern angegeben (was doch eigentlich einfacher ist als die Anweisungs-/Funktions-Unterscheidung im angeblich so einfachen Basic).

Für einen groben praktischen Überblick über einfachen Object-Pascal-Quelltext sei hier noch einmal auf das Kapitel 1.5.2 verwiesen.

OOP

Um mehr über die objektorientierte Programmierung in Object Pascal zu erfahren, sollten Sie das Kapitel 2.2 und folgende lesen. Die Unterschiede zwischen der objektorientierten Programmierung in Object Pascal und den Ansätzen zur objektorientierten Programmierung, die vielleicht in einigen Makrosprachen zu finden sind, sind zu groß, als dass sie einzeln beschrieben werden sollten.

2.1.2 Lexikalische Elemente

Der gesamte Quelltext eines Object Pascal-Programms darf nur aus folgenden Elementen bestehen:

- aus den von der Sprache vorgegebenen Schlüsselwörtern, Operatoren und Interpunktionszeichen
- direkter Angabe von Werten (Konstanten) in verschiedenen Typen (Zahlen, Zeichen und Strings)
- Bezeichnern
- Kommentaren

Wir gehen diese Punkte in umgekehrter Reihenfolge durch.

Kommentare dürfen überall zwischen den anderen syntaktischen Elementen stehen und dürfen alle Arten von Zeichen enthalten, außer natürlich der Zeichenfolge, die als Kommentarende-Markierung verwendet wird. Sie haben die Wahl zwischen drei Begrenzungsarten:

```
{Kommentare können so eingeklammert sein, }
(* oder so. Eine Vermischung der Begrenzer ist nicht erlaubt*)
  x := 0; // Dieser Kommentar endet "automatisch" am Zeilenende
```

Es ist möglich, Kommentare der ersten beiden Formen in zwei Ebenen zu schachteln, sofern Sie für jede Ebene eine andere der beiden Arten von Begrenzungszeichen verwenden.

Bezeichner sind aus den Buchstaben des Alphabets, aus Ziffern und aus dem Unterstrich aufgebaut, dürfen aber aus Gründen der Eindeutigkeit nicht mit einer Ziffer beginnen. Umlaute und andere länderspezifische Zeichen wie é, û oder å sind nur im Zusammenhang mit COM-Automation erlaubt (siehe Kapitel 8.6). Im folgenden Beispiel-Bezeichner ist eine weitere Regel versteckt:

```
Zwischen_Gross_und_Kleinschreibung_wird_nicht_unterschieden (* . *)
```

Die verschiedenen Typen konstanter Werte von Object Pascal werden im folgenden zusammengefasst.

Art der Konstanten-angabe	**Aufbau**	**Beispiel**
ganze Zahlen	Folge aus 0...9	9876543210
Hexadezimalzahlen	Hexadezimalzahl mit vorangestelltem '$'-Zeichen	$F0D9
Fließkommazahlen	Kommazahl+Exponent	3.4e5 – 1.4e+100 – 49e-4
einzelne Zeichen	Zeichen, eingeschlossen wie folgt:	'X'
einzelne Zeichen	#+Zahlenwert	#65 (entspricht in ASCII- und ANSI-Code dem 'A')
einzelne Steuerzeichen	^A	entspricht dem Zeichen #1
Strings	Zeichenfolgen, eingeschlossen wie im Beispiel	"XYZ"
Mengen	Liste von Elementen, eingeschlossen in eckigen Klammern	[biSystemMenu, biMinimize, biMaximize]

Sonderfälle sind, wenn Sie Begrenzungszeichen selbst als Zeichen oder String angeben sollen: Sie werden dann einfach zweimal hintereinander geschrieben (ein Hochkomma als Zeichen wird geschrieben als »''''« und ein einzelnes Anführungszeichen als String sieht so aus: »'"'«).

Schlüsselwörter: Die mittlerweile schon 64 Schlüsselwörter von Object Pascal, die Sie nicht für eigene Bezeichnernamen verwenden dürfen, sind:

and	array	as	asm	begin	class
case	const	constructor	destructor	dispinterface	div
do	downto	else	end	except	exports
file	finalization	finally	for	function	goto
if	implementation	in	inherited	initialization	inline
interface	is	label	library	mod	nil
not	object	of	on	or	packed
procedure	program	property	raise	record	repeat
set	shl	shr	string	then	threadvar
to	try	type	unit	until	uses
var	while	with	xor		

Ebenfalls zur Sprache Object Pascal gehören die folgenden Standarddirektiven:

absolute	abstract	assembler	automated	cdecl	default
deprecated	dispid	dynamic	export	external	forward
implements	index	library	message	name	nodefault
overload	override	pascal	platform	private	protected
public	published	read	readonly	register	reintroduce
resident	safecall	stdcall	stored	virtual	write
writeonly					

Wegen der Kompatibilität zu 16-Bit-Pascal unterstützt Delphi außerdem noch die Direktiven *near* und *far*, auch wenn sie im flachen 32-Bit-Speichermodell keine Wirkung haben. In der Online-Hilfe finden Sie schließlich noch einige Direktiven, die nur für den Quelltext von Packages benötigt werden und die hier nicht aufgeführt wurden (*package*, *contains* und *requires*).

Die Besonderheit der Standarddirektiven gegenüber den Schlüsselwörtern ist, dass sie nicht ausschließlich für den Compiler reserviert sind. Sie dürfen diese Direktiven ohne Einschränkung für eigene Bezeichner verwenden, weil keine Zweideutigkeiten entstehen können: Da, wo die meisten der Standarddirektiven stehen – am Ende von Funk-

tionsköpfen –, können keine Bezeichner stehen und umgekehrt kommen die Standarddirektiven nie in Funktionsrümpfen vor. Die Worte *private*, *protected*, *public* und *published* sind allerdings innerhalb einer Objekttyp-Deklaration reserviert, können also nicht als Name von Methoden oder Variablen einer Objektklasse verwendet werden, weil der Compiler dann wieder mehr überlegen müsste, ob nun ein Bezeichner oder die Direktive gemeint ist.

Bei Standarddirektiven und Schlüsselwörtern unterscheidet Object Pascal ebenso wenig zwischen Groß- und Kleinschreibung wie bei Bezeichnern.

Semikolons

Wie schon in der Schnellübersicht aus Kapitel 1.5.2 gezeigt, besteht der Programmcode auf höherer Ebene aus Zuweisungen, Aufrufen von Methoden / Funktionen / Prozeduren und Kontrollstrukturen. Jeder dieser drei Anweisungstypen wird an geeigneter Stelle beschrieben (wobei die Zuweisung kaum über das Kapitel 1.5.2 hinauskommt), so dass hier nur eine grundsätzliche Bemerkung zu den Semikolons übrig bleibt.

Semikolons werden in Pascal grundsätzlich dazu verwendet, Anweisungen zu trennen, das heißt, dass Sie die letzte Anweisung eines Anweisungsblocks nicht durch ein Semikolon abschließen müssen. Da Pascal jedoch eine leere Anweisung akzeptiert, können Sie nach der letzten Anweisung (und an anderen Stellen) beliebig viele weitere Semikolons setzen. In den Programmen dieses Buchs werden der Einfachheit halber alle Anweisungen durch ein Semikolon abgeschlossen.

Zur Verwendung der Semikolons in Konstanten-, Typen- und Variablendeklarationen und an anderen Stellen nehmen die entsprechenden Abschnitte dieses Kapitels Stellung.

2.1.3 Compileranweisungen

Innerhalb eines Quelltextes können Sie nicht nur Anweisungen für das Programm, sondern auch für den Compiler unterbringen. Da diese nicht zur eigentlichen Programmiersprache gehören, haben sie sich ein Versteck gesucht, wo sie nicht mit der Sprache verwechselt werden können: die Kommentare. Zur Unterscheidung von normalen Kommentaren beginnen sie mit einem »$« -Zeichen. Um also beispielsweise den Compiler-Schalter *$B* einzuschalten, genügt es, ein »*{$B+}*« in den Quelltext zu schreiben.

Schalter

Die einfachste Anweisung an den Compiler ist es, eine der Optionen zu setzen, die Sie auch in der Dialogbox PROJEKT | OPTIONEN finden. Zu jeder Dialogboxoption gibt es eine entsprechende Compileranweisung und darüber hinaus einige weitere Compiler-

schalter. Manche davon dürfen nur am Anfang einer Quelltextdatei gesetzt werden. Genauere Informationen zu den einzelnen Optionen finden Sie in der Online-Hilfe. Einige wichtige Schalter, die auch in diesem Buch verwendet oder an anderer Stelle erwähnt werden, sind:

- $B: Auswertung von booleschen Ausdrücken, siehe *Auswertung von booleschen Ausdrücken* in Kapitel 2.4.2
- $O: Optimierung an / aus (siehe Kapitel 1.7.1)
- $C: Assertion-Auswertung an / aus (siehe Kapitel 1.7.6)
- $H: steuert Schalter die Verwendung der langen Strings (siehe Kapitel 2.4.4).
- $I: I / O-Fehlerbehandlung mit Exceptions oder mit *IoResult* (siehe Kapitel 2.6.5)
- $J: macht typisierte Konstanten zu echten Konstanten (siehe Kapitel 2.1.5).
- $R: Bereichsüberprüfung (siehe Kapitel 2.6.5)
- $S: Stacküberprüfung, bricht das Programm bei einem Stack-Überlauf ab.
- $D, $L, $Y: Informationen für Browser und Debugger in die DCU-Datei aufnehmen (Kapitel 1.7.1)
- $P: Aktivierung von offenen Arrays (Kapitel 2.5.3)
- $N: Einschalten der erweiterten Fließkommatypen (Kapitel 2.4.1)
- $A1, $A2, $A4 und $A8: Stellt die Ausrichtung der einzelnen Felder von Records auf Adressen mit Vielfachen von 1, 2, 4 oder 8 ein, was einem schnelleren Zugriff durch moderne Prozessoren dient. Während die Voreinstellung in Delphi 6 der Einstellung *$A8* entspricht, wird beispielsweise unter Kylix für die Deklaration von aus der *libc* stammenden Records eine Ausrichtung an DWord-Grenzen (*$A4*) benötigt. Die Einstellung von *$A8* bewirkt beispielsweise, dass ein Record mit zwei *Char*-Variablen für jede Variable 8 Bytes zur Verfügung stellt, obwohl bei knapper Kalkulation auch jeweils ein Byte genügen würde (mit der Deklaration als *packed record* können Sie das Ausrichten der Felder verhindern und erhalten so Records mit minimalem Platzbedarf).

Alternativ zu diesen Kürzeln können Sie auch eine ausführlichere Schreibweise wählen, z.B. *$BoolEval On* statt *$B+* und *$Assertions Off* statt *$C-*. Eine Übersicht finden Sie in der Online-Hilfe.

Neben den Schaltern gibt es einige Anweisungen, von denen hier nur die *$I-* und die *$*R-Anweisung erwähnt werden sollen.

Ressourcen und Include-Dateien

Mit der folgenden Compileranweisung können Sie den Compiler veranlassen, temporär zum Zeitpunkt der Übersetzung eine andere Datei in den Quelltext einzufügen, im folgenden Fall `incl.pas`:

```
(*$I incl.pas*)
```

Dies war früher eine Möglichkeit, sich gegen das Fehlen des Unit-Konzepts zu wehren und einen Quelltext auf mehrere Dateien zu verteilen.

Ähnlich und zurzeit noch häufiger verwendet wird die Anweisung *$R Datei*. Eine solche finden Sie in jeder Formular- und in jeder Projektdatei. Sie bindet Ressourcendateien (`.res`) oder Formulardateien (`.dfm`) zur EXE-Datei hinzu. Ressourcen, die in Delphi nicht in idealer Weise editiert werden können, wie z.B. Bitmaps, können Sie so mit einem externen Tool erstellen und in Ihr Projekt einbinden (siehe Kapitel 1.4.8).

Bedingte Compilierung

Durch bedingte Compilierung ist es möglich, mehrere Versionen eines Programms mit nur einer Quelltextdatei zu generieren. Dies lohnt sich dann, wenn die Unterschiede zwischen beiden Versionen nur aus wenigen Zeilen bestehen. Beispielsweise könnte es eine Programmversion für Windows geben und eine andere für Linux, oder Sie könnten zusätzlichen Code für die Fehlersuche einfügen, der jedoch in der endgültigen Programmversion nicht mitübersetzt werden soll:

```
{$ifdef CompilerBedingung}
   ... Programmcode ...
{$else}
   ... alternativer Programmcode ...
{$endif}
```

In diesem Beispiel würde der Compiler den ersten Programmteil übersetzen, wenn das Symbol *CompilerBedingung* definiert ist, ansonsten würde er sich für den zweiten Teil entscheiden. Dieser *else*-Teil ist optional. Statt dem *ifdef* könnten Sie auch ein *ifndef* verwenden, was soviel bedeutet wie »*wenn nicht*«. Das Symbol *CompilerBedingung* hat nichts mit irgendwelchen Bezeichnern des Quelltextes zu tun, sondern muss auf andere Weise definiert werden:

- durch die Compileranweisung *{$define CompilerBedingung}*,
- durch Angabe des Symbols in den Projektoptionen, Seite VERZEICHNISSE / BEDINGUNGEN, Feld DEFINITION,
- durch den Compiler selbst. Die vom Compiler vordefinierten Symbole ermöglichen es, zwischen verschiedenen Versionen zu unterscheiden und je nach Version unterschiedlichen Code auszuführen. Die Zählung der Compiler-Version geht dabei auf

Turbo Pascal 1.0 zurück, so dass die aktuellen Entwicklungssysteme hier schon lange in den zweistelligen Versionsnummern sind. Die folgende Tabelle zeigt die wichtigsten Symbole der bisherigen »Delphi-kompatiblen Produkte« von Borland:

Produkt	Vordefinierte Symbole
Delphi 1	VER80, Windows, CPU86
Delphi 2	VER90, Win32, CPU386
Delphi 3	VER100, Win32, CPU386
Delphi 4	VER120, Win32, CPU386
Delphi 5	VER130, Win32, CPU386
Delphi 6	VER140, Win32, MSWINDOWS, CPU386
Kylix 1	VER140, Linux, CPU386

Neue Bedingungs-Direktiven

Ab der Version »Ver140« unterstützt der Compiler noch ein alternatives Set von Direktiven zur bedingten Übersetzung: *$if, $elseif* und *$ifend*. Über das Abfragen einfacher Symbole hinaus können Sie mit *$if* und *$elseif* auch Ausdrücke auswerten:

```
const MeineKonstante = 'LN';
{$if MeineKonstante = 'XA'}
```

Dabei können Sie auf deklarierte Objekt Pascal-Konstanten wie oben auf *MeineKonstante* zugreifen. Besonders interessant ist die Abfrage der vordefinierten Konstante *RTLVersion*, die in Delphi 6 und Kylix als »14.0« definiert ist und damit der Compiler-Version entspricht. Der Unterschied zur Abfrage des Compiler-Symbols *VER140* liegt darin, dass Sie auch alle nachfolgenden Versionen in der Bedingung berücksichtigen können:

```
{$ifdef VER140} // in nachfolgender Version nicht mehr definiert
{$if RTLVersion >= 14} // in nachfolgenden Versionen definiert
```

Schließlich bietet Delphi Ihnen für die Ausdrücke in *$if* noch zwei spezielle »Funktionen« an: Mit *Declared(MeinSymbol)* können Sie abfragen, ob ein Object-Pascal-Symbol deklariert wurde, mit *Defined(Compilersymbol)* fragen Sie ab, ob ein Compiler-Symbol, wie es auch mit der alten Direktive *$ifdef* abgefragt werden kann, definiert ist.

Für die Portierung hilfreiche Warnungen

Ebenfalls ab der Compilerversion *VER140* werden drei neue Direktiven unterstützt, mit denen beliebige Deklarationen (von Variablen, Typen, Funktionen, Klassen oder Units) gewissermaßen als »nicht uneingeschränkt zur Verwendung empfohlen« deklariert werden können:

- *deprecated* markiert Deklarationen als veraltet, wenn statt dessen lieber eine neue Alternative verwendet werden sollte.
- *platform* markiert Deklarationen als plattformspezifisch und warnt somit vor Portierungsproblemen, sollte die Anwendung einmal auf eine andere Plattform portiert werden.
- *library* markiert die Deklaration als abhängig von der verwendeten Bibliothek (wie VCL unter Windows, CLX unter Linux).

Es können sowohl einfache Programmobjekte wie Konstanten und Funktionen markiert werden ...

```
const
  // Beispiel aus SysUtils.pas: Das ReadOnly-Flag ist spezifisch für
  // die Dateisysteme von Windows (unter Linux ist der Zugriffsschutz
  // auf den drei Ebenen Benutzer/Gruppe/Alle geregelt):
  faReadOnly = $00000001 platform;
// Beispiel aus SysUtils.pas - Diese Funktion gibt es nur unter Windows:
function FileGetAttr(const FileName: string): Integer; platform;
```

... als auch ganze Records oder Klassen ...

```
type
  // Beispiel aus System.pas: Die Felder von TInitContext sind teilweise
  // von der verwendeten Plattform abhängig:
  TInitContext = record
    ..
  end platform;
```

... oder Units:

```
unit WindowsFunktionsSammlung platform;
```

Die Wirkung der Direktiven besteht darin, dass der Compiler Warnungen ausgibt, wenn Sie auf ein mit diesen Direktiven versehenes Programmobjekt zugreifen. Die Ausgabe der Warnungen kann mit den Compiler-Direktiven *$HINTS ON/OFF* und *$WARNINGS* gesteuert werden (für Detailfragen diesbezüglich siehe Online-Hilfe).

2.1.4 Typen und Variablen

In der visuellen Umgebung der Delphi-IDE kommen Sie zwar vorwiegend mit Formularen, Komponenten und Properties in Berührung, auf Methodenebene wird in Delphi jedoch mit den Variablen- und Typenkonzepten programmiert, die überwiegend schon in der 1972 von Niklaus Wirth entwickelten Ur-Pascal-Version vorhanden waren. Klassen und Objekte sind in dieses Konzept eingebaut: So entpuppen sich die Klassen syntaktisch als Erweiterungen der Record*typen*, und dementsprechend sind Objekte *Variablen*, denen als Typ eine Objektklasse zugewiesen wurde. In diesem Abschnitt sol-

len nur die allgemeinen Regeln für Typen und Variablen zusammengefasst werden, in Kapitel 2.4 finden Sie einen Überblick über alle Typen.

Deklarationsblöcke

Durch die übersichtliche Struktur eines Pascal-Programms lassen sich Variablen, Typen und Konstanten einfacher unterscheiden als beispielsweise in C++ oder Java: Jeder der drei Bezeichnertypen muss in einem Abschnitt mit spezieller Überschrift deklariert werden. Diese Abschnitte befinden sich immer außerhalb der Abschnitte, die den Programmcode enthalten (also außerhalb jeglicher *begin...end*-Blöcke). So gehören die Variablen in den *var*-Abschnitt:

```
var
  EineZahl: Integer;
  Taste1, Taste2: Char;
begin
  ...
```

Eine Variablendeklaration weist einer oder mehrerer durch Kommata getrennten Variablen den hinter dem Doppelpunkt stehenden Typ zu. Die oben verwendeten Typen *Integer* und *Char* sind bereits vordefiniert.

Typendeklarationen

In jeder Formulardatei befindet sich bereits die Deklaration eines Formulartyps bzw. einer Formularklasse, die somit ein besonders umfangreiches Beispiel einer Typendeklaration gibt. Die einfachste Art, einen eigenen Typ zu deklarieren, besteht darin, ihn einfach als Ersatzbezeichner für einen der vordefinierten Standardtypen zu verwenden:

```
type
  TTabellenIndex = Integer; { deklariert den neuen Typ "TTabellenIndex" }
  TSeitenIndex = Integer;
  TBuchIndex = Byte;
```

Sie verwenden selbst definierte Typen genauso wie die vordefinierten:

```
var
  EineSeitenzahl: TSeitenIndex;
```

Die Unterschiede eines Typendeklarationsblocks zum Variablenblock liegen in der Überschrift *type*, in der Beschränkung auf einen Bezeichner pro Deklaration und in dem Gleichheitszeichen anstelle des Doppelpunkts. Mit der obigen Typendefinition haben Sie später beispielsweise die Möglichkeit, durch Ändern des *TBuchIndex*-Typs in *Word* alle *Byte*-Variablen, die für den Buchindex stehen, auf den größeren Wertebereich von *Word* zu setzen, während alle anderen *Byte*-Variablen unverändert bleiben – eine Maßnahme, die Ihnen mit Suchen&Ersetzen viel Arbeit gemacht hätte.

Weitere Deklarationsblöcke

Neben den Abschnitten *var* und *type* gibt es noch *const* (Kapitel 2.1.5), *exports* (Kapitel 8.3.1) und *label* (für *goto*-Sprünge, die in diesem Buch nicht näher erläutert werden).

Die Reihenfolge der Abschnitte spielt in Object Pascal keine Rolle: Sie können beliebig viele Variablen- und Typenblöcke miteinander abwechseln. Wenn Sie sie erst nach bestimmten Prozeduren, Funktionen und Methoden aufführen, schränkt das lediglich die Sichtbarkeit der Bezeichner ein, denn diese dürfen erst in den Prozeduren verwendet werden, die unterhalb der Bezeichnerdeklaration stehen.

2.1.5 Konstanten und initialisierte Variablen

Kapitel 2.1.2 hat die direkte Angabe von konstanten Werten bereits als elementaren Bestandteil des Quelltextes genannt. Sie können solchen konstanten Werten auch einen Namen geben und sie dadurch leichter mehrfach verwenden. Auch ohne diesen Namen könnte man einen Wert bereits als Konstante bezeichnen, in Object Pascal meint die Bezeichnung *Konstante* aber einen *Bezeichner*, der für einen konstanten Wert steht.

In Object Pascal gibt es zwei verschiedene Arten von Konstanten: In echten Konstanten geben Sie einem Bezeichner einen festen Wert, den der Compiler überall dort einsetzt, wo Sie den Bezeichner verwenden:

```
const
  SampleRate = 44100;
  BitAnzahl = 16;
  Kanaele = 2;
  BytesProSek = SampleRate*(BitAnzahl div 8)*Kanaele;
begin
  Sekundendaten := GetMem(BytesProSek);
```

Wie Sie dem Beispiel entnehmen können, akzeptiert der Compiler bei Konstanten auch Ausdrücke, solange in diesen wiederum nur Konstanten vorkommen.

Typisierte Konstanten

Zusätzlich zur direkten Wertangabe können Sie die Konstantendeklaration auch noch um eine Typenangabe erweitern:

```
{$J+}   // In Delphi 6 und Kylix erforderlich, wenn der Wert
        // von SampleRate im Code durch Zuweisung geändert werden soll.
const
   SampleRate: Word = 44100;
begin
   SampleRate := 48000;
```

Bis Delphi 5 wurden solche Deklarationen vom Compiler nicht wie echte (unveränderbare) Konstanten, sondern wie initialisierte (veränderbare) Variablen behandelt, was der Compileroption *{$J+}* entspricht. In der Voreinstellung von Delphi 6 *{$J-}* würde der Compiler die obige Zuweisung nicht akzeptieren (in den Projektoptionen finden Sie diese Option auf der Compiler-Seite unter dem Namen ZUWEISBARE TYPISIERTE KONSTANTEN; die ausführliche Schreibweise der *$J*-Compileroption im Code lautet *$WRITEABLECONST*).

Initialisierte Variablen

Es ist jedoch nicht notwendig, den Umweg über typisierte Konstanten zu gehen, um eine initialisierte globale Variable zu erhalten. Sie können hier allen globalen Variablen Initialisierungswerte zuweisen, die diese bei Programmstart anstatt der voreingestellten Null-Werte erhalten. Die Syntax stimmt mit der Konstantendeklaration überein, einziger Unterschied ist die Überschrift *var*:

```
var
  SampleRate: Word = 44100;
```

Lokale typisierte Konstanten

Lokale Variablen bleiben weiterhin immer uninitialisiert. Sie dürfen in Object Pascal jedoch typisierte Konstanten lokal zu einer Funktion deklarieren. Die Konstante ist dann wie eine lokale Variable nur innerhalb dieser Funktion ansprechbar, obwohl es sich eigentlich um eine initialisierte globale Variable handelt.

```
procedure CriticalProcedure;
const
  CallCounter: LongInt = 0;
begin
  inc(CallCounter);
```

In diesem Beispiel zählt die Prozedur *CriticalProcedure* ihre Aufrufe in einer »privaten« Variablen *CallCounter,* die bei Programmstart mit Null initialisiert ist.

2.1.6 Gültigkeitsbereiche und lokale Variablen

Jeden Bezeichner, den Sie in Object Pascal verwenden, indem Sie ihm einen Wert zuweisen oder in einen Ausdruck aufnehmen, müssen Sie vorher deklarieren. Ab dieser Deklaration ist der Bezeichner bis zum Ende seines Gültigkeitsbereichs verwendbar. Als Gültigkeitsbereiche zählen die Bereiche von Modulen, Funktionen und Klassen:

Bezeichner, die innerhalb einer Funktion, Prozedur oder Methode deklariert werden, sind nur innerhalb dieser gültig und werden als *lokal* bezeichnet:

```
procedure TForm1.FormCreate(Sender: TObject);
var { Deklaration lokaler Variablen }
  Start, LokalerZaehler: Integer;
begin
  LokalerZaehler := 0; { Verwendung der Variable }
  ...
```

Wenn Sie Funktionen schachteln, kann die geschachtelte Funktion auf alle vorher deklarierten lokalen Variablen der umgebenden Funktion zugreifen.

Bezeichner, die Sie in einer Klassendeklaration deklarieren, sind in allen Methoden der Klasse gültig und darüber hinaus in abgeleiteten Klassen, wobei hier die Schutzbestimmungen beachtet werden müssen (siehe Kapitel 2.2.6). Auch der Gültigkeitsbereich von Bezeichnern in Records ist auf den Record beschränkt. Innerhalb des Records gibt es zwar niemanden (keine Methode), der dadurch einen Vorteil hätte, jedoch können Sie mit der *with*-Anweisung in den Gültigkeitsbereich eines Records »hinabsteigen« (siehe Kapitel 2.5.1).

Bezeichner, die außerhalb der genannten Bereiche deklariert sind, heißen *globale* Bezeichner. Sie sind im gesamten Modul gültig, beginnend mit der Zeile ihrer Deklaration.

Verdeckungen

Globale Bezeichner können zeitweise durch lokale Bezeichner verdeckt werden. Gibt es im obigen Beispiel neben der lokalen auch eine globale Variable mit dem Namen *Start*, so wird diese von der lokalen Variablen *Start* verdeckt. Sie können die globale Variable aber dennoch ansprechen, indem Sie deren Gültigkeitsbereich direkt angeben. Der globale Gültigkeitsbereich hat den Namen des Moduls – wenn dieser beispielsweise *Unit1* ist, könnte die obige Methode auf die globale Variable wie folgt zugreifen:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Start: Integer;
begin
  Unit1.Start := 0; { spricht die globale Variable an }
  Start := 0; { spricht die lokale Variable an }
  ...
```

2.1.7 Das Modulkonzept

In Delphi gibt es drei verschiedene Modultypen: *library*, *program* und *unit*. Die Aufgaben der letzten beiden sind in Zusammenhang mit den Dateien eines Projekts schon in Kapitel 1.6.2 erläutert worden: *program*-Module dienen in Delphi als Projektdatei mit der Endung `.dpr`, in die mehrere *unit*-Module (Units) mit der Endung `.pas` eingebunden werden können. Eine *library* ist eine dynamische Link-Bibliothek (DLL).

Die Projektdatei (bzw. das Hauptmodul)

Am einfachsten ist das Hauptmodul aufgebaut:

```
[program ModulName;]
[uses SysUtils, ...;]
[beliebige Folge von var-, const-, type-
   Blöcken und Definitionen von Methoden, Prozeduren und Funktionen]
begin
  [Anweisungsfolge]
end.
```

Alles außer dem *begin...end*-Block (dem so genannten *Hauptprogramm*) ist optional. Unmittelbar nach dem abschließenden Punkt hinter dem *end* wendet sich der Compiler von der Datei ab, unabhängig davon, ob noch etwas folgt oder nicht.

Zu den optionalen Elementen des Programms gehört der Name des Moduls in der ersten Zeile (fehlt die *program*-Zeile, übernimmt der Compiler den Dateinamen). Dieser Name kann z.B. bei Zugriff auf den äußeren Gültigkeitsbereich des Moduls aus dem Innern des Moduls (von innerhalb einer Funktion) oder von einem äußeren Modul aus verwendet werden, um globale Bezeichner des Moduls anzusprechen, wenn diese durch andere Bezeichner verdeckt sind (siehe Kapitel 2.1.6).

In der *uses*-Klausel können Sie Units in das Programm einbinden (dazu später mehr). In einer nicht festgelegten Reihenfolge folgen darauf Typen, Variablen, Konstanten und Methoden.

Units

Die *unit* weist einen weniger freien Aufbau auf, besteht sie doch aus zwei Teilen namens *interface* und *implementation*.

```
unit ModulName; { Modulname muss angegeben werden }

interface
[uses-Klausel]
[Deklarationen]

implementation
[uses-Klausel]
[Deklarationen und Definitionen]

Unit-Hauptprogramm, abkürzbar als "end."
```

Eine Unit kann alle Arten von Object-Pascal-Objekten enthalten, wobei mit »Objekt« hier allgemein die Objekte des Programmierens, also Prozeduren, Funktionen, Variablen, Klassen etc. gemeint sind. Sinn der Unit ist es, diese Objekte teilweise oder ganz anderen Modulen zur Verfügung zu stellen. Die Unit ist also durchaus berechtigt, einige Bestandteile geheim zu halten.

Alle Objekte, die in anderen Modulen benutzt werden dürfen, müssen im *Interface* deklariert werden. Dadurch werden die entsprechenden Bezeichner öffentlich (*public*), also nach außen hin sichtbar. Bei Variablen genügt diese einmalige Deklaration, alle Funktionen des Programms und der Objekte müssen außerdem noch definiert werden, und zwar im Abschnitt *Implementation*. Alle weiteren Deklarationen, die innerhalb dieses Teils stehen, sind privat.

Ähnlich verhält es sich mit den *uses*-Klauseln: Nur die öffentlichen Symbole der Units, die Sie im Interface einbinden, werden automatisch auch den äußeren Modulen bekannt. Der eigentliche Sinn zweier getrennter *uses*-Anweisungen ist jedoch, zirkuläre Nutzungsverhältnisse festzulegen – wenn sich beispielsweise zwei Units gegenseitig benutzen wollen. Das folgende Beispiel würde den Compiler in eine Endlosschleife stürzen, wenn dieser den Fehler nicht erkennen würde:

```
{ Dateibeginn der ersten Unit: }
unit Unit1; interface uses Unit2;
{ Dateibeginn der zweiten Unit: }
unit Unit2; interface uses Unit1;
```

In diesem Fall muss eine der *uses*-Klauseln auf den Implementationsteil der Unit verschoben werden, beispielsweise:

```
{ Beginn der ersten Unit: }
unit Unit1; uses Unit2;
---
{ Beginn der zweiten Unit: }
unit Unit2;
interface
implementation
uses Unit1;
```

Wenn der Compiler nun *Unit1* übersetzt, kann er das Interface der *Unit2* einlesen, ohne dort sofort aufgetragen zu bekommen, sich zuerst dem Interface von *Unit1* zuzuwenden. Die obige Ausweichlösung bringt natürlich als Einschränkung mit sich, dass Sie im Interface von *Unit2* keine Bezeichner aus *Unit1* mehr verwenden können; diese Einschränkung dürfte sich in der Praxis jedoch kaum auswirken.

Units initialisieren und verlassen

Der in der obigen Übersicht als »Unit-Hauptprogramm« ausgewiesene Abschnitt wurde in Object Pascal bisher von Version zu Version leicht erweitert. Da der in diesem Abschnitt ausgewiesene Code beim Programmstart ausgeführt wird und die Unit so Gelegenheit hat, sich selbst zu initialisieren, können Sie diesen Abschnitt statt mit *begin* auch mit *initialization* einleiten.

Passend dazu gibt es noch das Schlüsselwort *finalization*, das als Überschrift für den Code dient, der bei Beendigung des Programms auf jeden Fall aufgerufen werden soll. Das Ende einer Unit kann damit wie folgt aussehen:

```
initialization
  { Code zum Initialisieren der Unit }
finalization
  { Code zum Aufräumen nach der Programmausführung }
end.
```

Konnte man in 16-Bit-Zeiten als Ersatz für eine *finalization* noch eine Exit-Prozedur installieren, so verbietet sich die Anwendung von *AddExitProc* mittlerweile, da sie seit Delphi 3 mit der Package-Technologie kollidiert.

2.2 Objekte und Klassen in Object Pascal

Das zentrale Konzept der objektorientierten Programmierung ist die Zusammenfassung von Daten und Code – in der prozeduralen Programmierung noch strikt getrennt – zu abgeschlossenen Strukturen namens *Objekten*. Die Objekte, mit denen Sie im ersten Kapitel Bekanntschaft gemacht haben, waren in erster Linie Komponenten und Formulare. Auch die Bestandteile der einzelnen Komponenten, die Sie im Objektinspektor angezeigt finden, sind nicht immer nur einfache Werte wie Zahlen oder Strings, sondern mitunter auch Objekte, beispielsweise die *Font*-Objekte verschiedener Komponenten.

Aus der Vereinigung von Daten und Code ergeben sich viele Vorteile wie z.B.:

- Innerhalb einer Methode haben Sie direkten Zugriff auf alle Elemente des Objekts, ohne dass Sie mit einer *with*-Anweisung oder auf andere Arten auf eine »fremde« Datenstruktur zugreifen müssen (siehe Kapitel 2.5.1).
- Es steht (normalerweise) immer fest, welche Prozeduren zu welchen Daten gehören.
- Alle Probleme der globalen Variablen können vermieden werden.
- Sie können die Daten eines Objekts verbergen, so dass der Nutzer des Objekts nur die Methoden, aber nicht die Implementierung des Objekts zu kennen braucht (insofern ist ein Objekt so etwas wie eine eigene Unit mit Interface- und Implementationsteil).

Hinzu kommen natürlich die Vorteile der objektorientierten Programmierung im Allgemeinen, zu denen die leichtere Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit von Code und die bessere Strukturierung der Programmieraufgabe gehören.

In diesem Kapitel sehen wir uns zuerst das Innenleben der Objekte an, bevor wir in Kapitel 2.3 untersuchen, wo die Objekte entstehen und wo sie enden.

2.2.1 Der Aufbau von Objekten

Für Object Pascal bedeutet die Vereinigung von Code und Daten konkret, dass ein Objekt aus Properties, Variablen und Methoden besteht, wobei von außen betrachtet Properties und Variablen den Datenanteil des Objekts bilden und der Code sich in den Methoden befindet. (Von innen betrachtet handelt es sich bei den Properties ja um eine Kombination von Code und Daten.)

OOP und visuelle Programmierung

In der visuellen Programmierumgebung von Delphi haben Sie Zugriff auf den Teil der Properties und Methoden, den die Programmierer der entsprechenden Komponente für die Delphi-Oberfläche freigegeben haben (siehe Kapitel 6.4.3).

Wie Sie schon in Kapitel 1 erfahren haben, gestalten Sie beim Entwurf des Formulars nicht die eigentlichen Fensterobjekte, sondern nur die Schablonen dafür. Dass von der Schablone des Hauptformulars beim Programmstart nur ein Fenster geöffnet wird, ist lediglich eine sinnvolle Voreinstellung von Delphi, denn von den Formularschablonen können Sie theoretisch beliebig viele Exemplare des gleichen Fensters erzeugen.

2.2.2 Klassen und Instanzen

Das Formular, das am Anfang der Unit deklariert ist, wird in Object Pascal als Objekt*klasse* bezeichnet. Das Fenster, das zur Laufzeit am Bildschirm zu sehen ist, liegt im Programm als Objekt*instanz* vor. Diese Unterscheidung von Objektinstanz und Objektklasse gilt auch für alle anderen Objekte in Object Pascal, die nichts mit Fenstern zu tun haben. Mit der Kurzbezeichnung *Objekt* ist in diesem Buch immer die Objektinstanz, also die Variable gemeint; Objektklassen lassen sich in Object Pascal nun ohne Bedenken kurz als *Klassen* bezeichnen, da sie mit dem Schlüsselwort *class* deklariert werden, und nicht wie unter Borland Pascal mit dem Schlüsselwort *object* (eine alternative Bezeichnung für die Klasse ist *Objekttyp*). Dennoch werden Sie feststellen, dass Borlands Online-Hilfe es hier nicht so genau nimmt und Klassen, die keine Komponentenklassen sind, auch als »Objekte« bezeichnet.

Ein wichtiger Unterschied zum *Record* liegt bei den Klassen darin, dass Sie sie nicht direkt einer Variablen zuweisen können. Record-Variablen können Sie noch wie folgt deklarieren:

```
var
  Struktur: record
    Element1, Element2: Char;
  end;
```

Dies ist bei Objekten nicht mehr möglich. Da Sie von diesen meist mehrere Exemplare benötigen, ist es auch sinnvoll, für jedes Objekt zuerst eine Klasse und davon dann

eine Objektinstanz anzulegen. Falls die obige Variable *Struktur* also keine normale Variable, sondern ein Objekt sein soll, müsste die Deklaration wie folgt aussehen:

```
type
  TStruktur: class
    Element1, Element2: Char;
  end;

var
  Struktur: TStruktur;
```

2.2.3 Die Klassendeklaration

Sehen wir uns eine beispielhafte Objektklasse an: Angenommen, Sie haben Ihrem Formular den Namen *MainWindow* gegeben, so erzeugt Delphi im dazugehörigen Programmcode eine Klasse mit dem Namen *TMainWindow*. Diese könnte beispielsweise wie folgt aussehen:

```
type
  TMainWindow = class(TForm)
    TabSet1: TTabSet;
    ColorGrid1: TColorGrid;
    ScrollBar1: TScrollBar;
    CheckBox1: TCheckBox;
    procedure FormPaint(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure TabSet1Change(Sender: TObject; NewTab: Integer;
      var AllowChange: Boolean);
  private
    { Private-Deklarationen }
    x3, y3, x4, y4: Integer;
  public
    { Public-Deklarationen }
  end;
```

Da es sich bei jeder neuen Klasse um einen »selbst definierten Typ« handelt, befindet sich diese Deklaration in einem Abschnitt, der durch das Schlüsselwort *type* eingeleitet wird (allerdings würde die Bezeichnung als »Datentyp« dem Wesen der Objekte nicht mehr gerecht, da sie die Methoden nicht angemessen berücksichtigen würde). Das wichtigste Schlüsselwort bei der Deklaration ist das Wort *class*. Es unterscheidet die ihm folgende Struktur von den Records und den alten Objekttypen, die statt dessen mit *record* bzw. mit *object* eingeleitet werden. Wir verwenden hier ausschließlich Objektklassen, die mit dem Schlüsselwort *class* deklariert werden; auf Unterschiede zu den mit *object* deklarierten Verwandten geht Kapitel 2.3 (der Lebenslauf von Objekten)

ein. Nach *class* wird in Klammern die Basisklasse angegeben, von der die Klasse *erbt*. Mit dieser Vererbung beschäftigt sich Kapitel 2.2.7.

Darauf folgen die Bestandteile der Klasse, und zwar in Abschnitten mit den Überschriften *public*, *published*, *private* und *protected* (siehe Kapitel 2.2.6), die in einer beliebigen Reihenfolge auftreten können, allerdings müssen Sie bei den Formularklassen den ersten, unbetitelten Abschnitt für die automatische Code-Erzeugung von Delphi reservieren.

Innerhalb eines Abschnitts müssen Sie zuerst die Variablen und dann die Methoden aufführen. Sobald die erste Methode erscheint, dürfen Variablen erst wieder nach einer neuen Überschrift wie z.B. *public* folgen. Die einzelnen Deklarationen von Methoden und Variablen halten sich an dieselben Regeln wie normale Variablen, Prozeduren und Funktionen.

Mit einem *end* und einem Semikolon schließen Sie die Klassendeklaration ab. Weitere Informationen speziell zum automatisch erzeugten Anteil der Formularklassen finden Sie in Kapitel 1.5.7.

2.2.4 Zugriff auf die Klassenelemente

Wir haben im Laufe der ersten Kapitel schon des Öfteren auf die Elemente einer Klasse zugegriffen, es wird also Zeit, zu untersuchen, von wo aus die Elemente eines Objekts auf welche Art angesprochen werden können.

Der self-Parameter

Für den Einstieg in die OOP kann das Verständnis des *self*-Parameters wichtig sein, wenn Sie bisher zu stark an die Verwendung von nicht-objektorientierten Strukturen und damit an die strikte Trennung von Code und Daten gewöhnt sind. Da der *self*-Parameter an sich etwas sehr »Natürliches« ist, wird er Ihnen wahrscheinlich bald völlig selbstverständlich vorkommen.

Ein Objekt soll natürlich seine Elemente gut kennen, daher ist es logisch, dass Sie in einer Objektmethode nur den Namen des Elements anzugeben brauchen, um sich auf dieses zu beziehen. So können Sie innerhalb einer Ereignisbearbeitungsmethode eines Formulars direkt auf die Properties des Formulars zugreifen:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Color := clRed+clGreen+clBlue;
```

Im Falle eines einzelnen Fensters scheint es dafür noch keine Probleme zu geben. Wenn nun aber viele Fenster derselben Formularklasse z.B. als MDI-Kinder existieren, dann verfügt jedes dieser Fenster über seine eigene Farbe. Wie kann nun die obige Zeile wissen, wessen *Color*-Property gemeint ist?

Die Antwort liegt im unsichtbaren *self*-Parameter, den jede Methode automatisch zugeordnet bekommt. *Self* gibt das Objekt an, mit dem die Methode arbeitet. Der Compiler stellt jedem Objektelement, das Sie innerhalb der Methode verwenden, automatisch diesen *Self*-Parameter voran. Jede Methode ist also intern so organisiert:

```
procedure TForm1.
  Methode(self: TForm1; ... hier folgen die sichtbaren Parameter);
begin
  with self do begin
    ... hier kommen die sichtbaren Anweisungen
  end;
end;
```

Sie finden *self* weder unter den reservierten Wörtern noch unter den Standarddirektiven, können diesen Namen also uneingeschränkt für eigene Variablen verwenden. Wenn Sie *keine* Variable mit dem Namen *self* deklarieren, können Sie die vordefinierte Version von *self* in Ihrem Code ansprechen, z.B.:

```
with FarbigeKomponente do
  Color := self.Color;
{ ist das gleiche wie: }
FarbigeKomponente.Color := Color;
```

Zugriff von außen auf die Objektelemente

Wenn nun Programmcode von außen auf Elemente des Objekts zugreifen soll, müssen Sie zuerst die Objektinstanz angeben und das gewünschte Element hinter einem Punkt anschließen, um z.B. die *Execute*-Methode aufzurufen:

```
EingabeDialog.Execute;
```

Diese Schreibweise können Sie immer verwenden, wenn Sie in Formularmethoden auf die Komponenten des Formulars zugreifen, eine Alternative dazu ist die *with*-Anweisung (siehe Kapitel 2.5.1).

2.2.5 Properties

Die Properties sind eine der besonders wichtigen sprachlichen Neuerungen von Delphi gegenüber Borland Pascal. Sie scheinen auf den ersten Blick das OOP-Konzept, die Daten eines Objekts nach außen nicht zu zeigen (geschweige denn, sie von dort verändern zu lassen), zu durchkreuzen. Wie Sie in Kapitel 2.2.5 jedoch genauer sehen können, sind die Daten in den Properties mit speziellen Methoden verbunden, die diese Daten verändern. Jedes Mal, wenn Sie einem Property einen Wert zuweisen oder das Property nur lesen (einer anderen Variable zuweisen), wird eine Methode des Objekts aufgerufen, so dass dieses von jeder Änderung unterrichtet wird. Tatsächlich sind die Properties also nur vereinfachte Methodenaufrufe. Ein Beispiel soll den Unterschied erläutern:

Früher hätte man von außen die Farbe eines Objekts so ändern können:

```
Object.Color := clRed;
Object.Paint; (* neue Farbe am Bildschirm ausgeben *)
```

Besser wäre es gewesen, die Objektdaten (*Color*) nicht direkt zu verändern, sondern statt dessen eine Methode aufzurufen:

```
Object.SetColor(clRed);
(* Object.Paint könnte von SetColor
   automatisch aufgerufen werden *)
```

Handelt es sich bei *Color* jedoch um ein Property, weisen Sie ihm einfach einen Wert zu wie im oben gezeigten, damals noch schlechten Beispiel:

```
Object.Color := clRed;
```

Die Methode, die bei dieser Zuweisung automatisch aufgerufen wird, sollte die Farbänderung auch am Bildschirm sichtbar machen. Bei den Komponentenklassen der VCL können Sie sich jedenfalls darauf verlassen, dass diese Aufgaben wie das Aktualisieren der Bildschirmanzeige von alleine erledigen.

Eigene Properties deklarieren *R119*

Properties eignen sich auch außerhalb von Komponenten in Formularen und eigenen Nicht-Komponenten-Klassen hervorragend zur Vereinfachung der Programmierung. Properties werden an derselben Stelle einer Klassendeklaration deklariert wie Methoden, dürfen also erst nach allen Variablen desselben Abschnitts genannt werden.

Die Deklaration eines Properties sieht wie folgt aus:

```
property Color: TColor read GetColor write SetColor;
```

Dabei ist *Color* der Name, unter dem Sie das Property ansprechen, die dahinter stehenden *read* und *write*-Direktiven geben die Methoden an, die beim Lese- bzw. beim Schreibzugriff aufgerufen werden. Sie können eine der beiden Direktiven auch weglassen, um ein Property zu erhalten, das nur gelesen bzw. nur beschrieben werden kann.

Die Lesemethode eines Properties ist eine Funktion ohne Parameter, deren Ergebnistyp mit dem Typ des Properties übereinstimmen muss, während die Schreibmethode eine Prozedur ist, die eine Variable des Propertytyps als Parameter hat, beispielsweise:

```
function TEineKlasse.GetColor: TColor;
begin
  Result := FColor;
end;

procedure TEineKlasse.SetColor(x: TColor);
begin
  FColor := x;
```

```
    Invalidate;
  end;
```

Wie in diesem Beispiel liegen einem Property oft Variablen desselben Typs zugrunde und es ist die Aufgabe der Property-Methoden, besondere Aktionen mit dem Lesen und Schreiben dieser Variable zu verknüpfen. Meistens wird diese Variable aus einem vorangestellten »F« und dem Namen des Properties gebildet (in diesem Beispiel *FColor*).

Variablen in Property-Verkleidung

Alternativ zu den Methoden können Sie in den *read*- und *write*-Direktiven auch Variablen angeben, die direkt gelesen oder geschrieben werden können. Im obigen Beispiel, in dem die Methode *GetColor* nur die Variable *FColor* ausliest, wäre es beispielsweise sinnvoller, das Property wie folgt zu deklarieren:

```
{ alle für das Property notwendigen Deklarationen: }
  FColor: TColor;
  procedure SetColor(x: TColor);
  property Color: TColor; read FColor write SetColor;
```

Property-Beispiele

Für vollständige und größere Property-Beispiele sei an dieser Stelle auf andere Teile dieses Buchs verwiesen, hier folgt eine Liste ohne Vollständigkeitsanspruch:

- Formular *TDocumentForm*: *ZoomFactor* (Kapitel 5.6.3) und *FileName* (Kapitel 5.2.5)
- Klasse TGraphicDoc: BrushColor und BrushName (Kapitel 5.3.2)
- Klasse *TGraphicElement*: *PrevNode* (Kapitel 5.3.4)
- Klasse *THistoryList*: *MaxLen* (Kapitel 4.1.2)

Darüber hinaus definieren natürlich die in Kapitel 6 entwickelten Komponenten Properties, die Sie auch zur Entwurfszeit im Objektinspektor editieren können.

Array-Properties

Ein Array-Property funktioniert nach außen hin so wie ein normales Array, Sie greifen also beispielsweise mit

```
Colors[0] := clWhite;
```

auf das nullte Element des Properties *Colors* zu. Bei der Deklaration geben Sie die Bereiche einer oder mehrerer Dimensionen nicht wie in einem normalen Array an, sondern schreiben jede Dimension des Arrays wie eine Variablendeklaration (z.B. *I: Byte*).

Auch wird der Typ eines Array-Properties nicht wie bei Arrays hinter einem *of*, sondern hinter einem Doppelpunkt angegeben:

```
property Colors[i: Byte]: TColor read GetColor write SetColor;
property DreiD[x, y, z: Integer]: Elementtyp; read Read3D write Set3D;
```

Die Schreib- und Lesemethoden sind nun jeweils um einen oder mehrere Index-Parameter erweitert, für die obigen Properties könnten sie folgendermaßen deklariert sein:

```
function GetColor(i: Byte): TColor;
procedure SetColor(i: Byte; val: TColor);
function Get3D(x, y, z: Integer): Elementtyp;
procedure Set3D(x, y, z: Integer; val: Elementtyp);
```

Im schon erwähnten Kapitel 5.3.2 gibt es mit dem von *TList* überschriebenen *Items*-Property auch ein Array-Property-Beispiel.

Das Standard-Property

Wenn Sie die Deklaration eines Array-Properties mit der Direktive *default* abschließen, wie beispielsweise das Property *TList.Items*, wird dieses Property zum *Standard-Property* der Klasse:

```
property Items[Index: Integer]: Pointer read Get write Put; default;
```

Für die Klasse *TList* bedeutet das beispielsweise, dass Sie ein ganzes Objekt dieser Klasse behandeln können wie dieses Array, ohne das Property extra nennen zu müssen. Die Ausdrücke *Liste.Items[i]* und *Liste[i]* haben daher dieselbe Wirkung. Es versteht sich von selbst, dass jede Klasse nur ein derartiges Standard-Property haben kann.

Eine Methode – mehrere Properties

Indizierte Properties sind vergleichbar mit Array-Properties, bei denen Sie nicht über einen Index, sondern über einen Namen auf die einzelnen Elemente zugreifen. Die Properties *Left*, *Right*, *Top* und *Bottom* der Klasse *TGraphicElement* aus Kapitel 5.3.2 beispielsweise sind wie folgt deklariert:

```
property Left: Integer index 1 read GetCoord;
property Right: Integer index 2 read GetCoord;
property Top: Integer index 3 read GetCoord;
property Bottom: Integer index 4 read GetCoord;
```

Alle vier Properties werden mit derselben Methode gelesen, wobei diese die Properties über den Index unterscheidet. Der Compiler fügt automatisch den passenden Index in den Methodenaufruf ein, wenn Sie eines dieser Properties ansprechen:

```
function TGraphicElement.GetCoord(Index: Integer): Integer;
begin
```

```
  with Points do
    case Index of
      1: Result := Left;
      2: Result := Right;
      3: Result := Top;
      4: Result := Bottom;
    end;
end;
```

Gleichermaßen funktionieren die Schreibprozeduren für ein solches Property, die natürlich wieder einen zusätzlichen Parameter für den Wert benötigen. Ein weiteres Beispiel für eine Methode, die mehrere Properties behandelt, gibt Kapitel 6.5.3 mit *SetIniData*.

Vererbung von Properties

Sie können Properties in abgeleiteten Klassen überschreiben, wobei Sie den Zugriffsschutz, in dem sie deklariert sind, sowie die Lese- und Schreibmethoden verändern können. Diese und einige weitere Variationen der Properties, die bei der Komponentenprogrammierung verwendet werden, sind in den Kapiteln 6.3.1 und 6.4.3 besprochen.

2.2.6 Zugriffsbeschränkungen

Durch zusätzliche Angaben in der Klassendeklaration können Sie bestimmte Elemente der Klasse vor dem Zugriff von außen schützen. Sie haben die folgenden vier Schutzebenen zur Verfügung (zur Anwendung der Direktiven siehe *Die Klassendeklaration*, Kapitel 2.2.3):

- *public* (öffentlich) lässt die Elemente eines Objekts vollkommen ungeschützt und erlaubt es, dass von außen darauf zugegriffen wird. Optimal ist es, wenn nur Methoden und Properties, aber keine Daten öffentlich sind.
- *protected* schützt die Elemente vor dem Zugriff von außen, was bedeutet, dass beispielsweise eine Variable, die im *protected*-Bereich der *Button*-Komponente deklariert ist, nicht von einer Ereignisbearbeitungsmethode des Formulars aus angesprochen werden kann. Sie dürfen die mit *protected* geschützten Elemente einer Klasse jedoch in von dieser abgeleiteten Klassen ohne Einschränkung verwenden.
- Die Schutzebene *private* versagt sogar abgeleiteten Klassen die Verwendung der so gekennzeichneten Elemente, so dass die besitzende Klasse die alleinige Kontrolle darüber hat, was mit diesen Elementen geschieht.
- Schließlich gehört noch eine vierte Direktive in die Reihe von *private*, *protected* und *public*: Die Schutzebene *published* (veröffentlicht), die noch niedriger als *public* ist, wird nur für Komponenten benötigt, um deren Properties sogar noch außerhalb

des Programms sichtbar zu machen, nämlich für die Delphi-IDE bzw. deren Objektinspektor. Standardmäßig sind alle Elemente einer Klasse dieser Schutzebene zugeordnet – solange Sie also keine der drei anderen Schutzebenen angeben, bleiben alle Elemente veröffentlicht. In der Formulardeklaration sind dies beispielsweise alle Elemente, die Delphi automatisch hinzufügt.

Kontrolle der Schutzbeschränkungen

Die Schutzebenen *private* und *protected* sind allerdings nicht ganz so streng, wie es zuerst aussieht: Sie gelten nur für den Zugriff aus einem anderen Modul heraus. Wenn Sie in einer Klasse *private*- und *protected*-Elemente deklarieren, können Sie aus anderen Klassen beliebig auf diese Elemente zugreifen, wenn sich diese anderen Klassen in derselben Unit befinden. Mit anderen Worten sind also alle Klassen eines Moduls miteinander befreundet, als wären sie in C++ als *friend* deklariert.

Da sich alle Klassen der VCL in eigenen Modulen befinden, können Sie auf die privaten Elemente der Komponenten gar nicht zugreifen, auf die *protected*-deklarierten Elemente nur innerhalb eigener, abgeleiteter Komponentenklassen.

2.2.7 Vererbung

Vielleicht fragen Sie sich, wo denn in der Formulardeklaration die Properties des Formulars zu finden sind. Die Antwort ist: Sie sind alle in der ersten Deklarationszeile versteckt, in dem Fragment *TForm1 = class (TForm)*, denn auf diese Weise erbt *TForm1* alle Elemente, die schon in der Klasse *TForm* deklariert sind (allerdings dürfen Sie von *TForm1* aus nicht auf die mit *private* geschützten Klassenelemente zugreifen):

In *TForm* implementiert die VCL die grundlegenden Eigenschaften und Fähigkeiten eines Formulars, daher ist ein leeres Formular, zu dem Sie noch keine Ereignisbearbeitungsmethoden hinzugefügt haben, sofort lauffähig. Durch die Vererbung müssen Sie das Formular in eigenen Anwendungen nicht vollkommen neu implementieren, sondern können die bestehende Formularklasse erweitern.

Natürlich gibt es auch Bibliotheken, die nicht objektorientiert sind und mit denen Sie ein Fenster ebenfalls nicht völlig neu implementieren müssen, jedoch vereinfachen die Eigenschaften des OOP die Nutzung bestehenden Codes und dessen Erweiterung enorm und machen sie elegant und leicht verständlich.

Der Prozess, aus einer bestehenden Klasse eine neue zu machen, wird als *Ableiten* bezeichnet. Im obigen Beispiel ist also *TForm1* von *TForm* abgeleitet. Die Klasse, von der abgeleitet wird, heißt *Eltern*- oder *Basisklasse*. Der Prozess des Ableitens ist auch in mehreren Ebenen wiederholbar. Auf diese Weise lassen sich Klassenhierarchien aufbauen, in denen es allgemeine Klassen gibt, die hauptsächlich als Basisklassen für speziellere Klassen dienen. Alle davon abgeleiteten Klassen müssen nur noch den ihnen

eigenen Funktionsanteil realisieren und können sich für den Funktionsbereich, in dem Übereinstimmung herrscht, auf die gemeinsame Basisklasse berufen.

Ein sehr gutes Beispiel für eine Klassenhierarchie ist die VCL selbst. So implementiert beispielsweise die VCL-Klasse TWinControl alle wichtigen Fähigkeiten, die eine Komponente, die mit einem Windows-Fensterelement verbunden ist, haben muss. Alle visuellen Steuerelement-Komponenten, die von dieser Klasse abgeleitet sind, erben bereits ohne eigenes Zutun die Verbindung zum Windows-Fensterelement. Sie können sich daher auf ihre eigenen speziellen Funktionen konzentrieren.

2.2.8 Vorwärtsdeklaration von Klassen

Damit zwei Klassen sich auch gegenseitig benutzen können, erlaubt Object Pascal die Vorwärtsdeklaration von Klassen. Um dem Compiler beispielsweise die Klasse *TDocument* bekannt zu machen, genügt die folgende Deklaration:

```
type
  TDocument = class;
```

Von da an können Sie Objekte dieser Klasse deklarieren. Die gegenseitige Benutzung zweier Klassen sähe dann so aus:

```
type
  TDocument = class; { Klasse für ein Dokument }
  TItem = class { Klasse für einen Bestandteil des Dokuments }
    { Der Bestandteil weiß, in welchem Dokument er sich befindet: }
    ParentDocument: TDocument;
  end;
  TDocument = class
    { Das Dokument kennt z.B. seinen ersten Bestandteil: }
    FirstItem: TItem;
  end;
```

2.3 Der Lebenslauf von Objekten

Wir untersuchen nun die Vorgänge, die zur Laufzeit des Programms mit einem Objekt passieren (können). Dazu gehören Initialisierung und Löschung mit Konstruktoren und Destruktoren sowie das sehr wichtige OOP-Thema der Polymorphie, bei der sich erst zur Programmlaufzeit entscheidet, welche Methode aufgerufen wird.

2.3.1 Initialisierung von Objekten: Konstruktoren

Wie Sie eine Instanz einer Klasse anlegen, zeigt Delphi Ihnen ebenfalls schon am Beispiel des Hauptformulars: Delphi erzeugt automatisch eine Variable des neuen Formulartyps, die den von Ihnen im Formular-Eigenschaftsfeld *Name* angegebenen Namen erhält, per Voreinstellung also den Namen *Form1*:

```
var
  Form1: TForm1;
```

In Verbindung mit dieser Variablen existiert genau ein Fenster der Klasse *TForm1*.

Wofür der von Delphi automatisch erzeugte Code kein gutes Beispiel gibt, ist die Initialisierung eines Objekts. Nachdem Sie eine Objektvariable wie oben gezeigt deklariert haben, können Sie sie noch nicht benutzen, ohne einen Laufzeitfehler oder eine Schutzverletzungs-Exception (*EAccessViolation*) zu erzeugen, denn jedes Objekt muss zuerst initialisiert werden, indem Sie seinen Konstruktor aufrufen (für Borland Pascal-Gewöhnte: wirklich *jedes* Objekt, das einen *class*-Typ hat, denn diese Objektvariablen bestehen nach ihrer Deklaration intern immer aus einem uninitialisierten Zeiger).

In den meisten Fällen sorgt jedoch die VCL für die Initialisierung der Objekte. Das Formular selbst wird in der Methode *CreateForm* initialisiert, die in der automatisch erzeugten Projektdatei aufgerufen wird. Zur Initialisierung des Formulars gehört auch die Initialisierung all seiner Komponenten, und diese Komponenten sind ja bereits alle Objekte, die in einer einfachen Delphi-Anwendung vorkommen.

Wenn Sie jedoch selbst dynamisch Komponenten oder andere Objekte erzeugen oder Objekte verwenden möchten, die unabhängig von einem Formular sind, müssen Sie Ihre Objekte selbst initialisieren.

Wie schon erwähnt, benötigen Sie dazu einen neuen Konstruktor. In diesem erhält das Objekt Gelegenheit, sich auf die Wirren der Programmlaufzeit vorzubereiten und sich selbst in einen angemessenen Anfangszustand zu versetzen. Das Mindeste, was im Konstruktor geschieht, ist in jedem Fall die Reservierung von Speicher, denn alle *class*-Objekte sind dynamisch und belegen erst nach dem Konstruktoraufruf nennenswerte Speichermengen. Der Konstruktor des Formulars heißt *Create* und ist von *TForm* geerbt.

Bei Formularen ist es normalerweise jedoch nicht notwendig, einen eigenen Konstruktor zu schreiben, denn das Initialisieren des Fensters ist auch ein Ereignis, das Sie im Objektinspektor unter dem Namen *OnCreate* finden. Verknüpfen Sie mit diesem eine Methode, in der Sie Ihre Initialisierungen durchführen.

Konstruktoren aufrufen

Konstruktoren haben nicht nur die Aufgabe, das Objekt zu initialisieren, sondern auch, das Objekt überhaupt zu erzeugen (Speicher für es zu reservieren). Um beispielsweise das Objekt *DynamicForm* zu erzeugen, können Sie daher nicht die folgende Anweisung verwenden:

```
DynamicForm.Create(ParentForm); { keine Objekterzeugung! }
```

Diese Anweisung ruft den Konstruktor *Create* mit einem Objekt (*DynamicForm*) auf, das noch gar nicht existiert. Um ein Objekt neu zu erzeugen, müssen Sie der Objektvariablen (hier *DynamicForm*) ein neues Objekt *zuweisen* (was in Borland Pascal über die *new*-Anweisung gemacht wurde). Das neue Objekt erhalten Sie, indem Sie den Konstruktor nicht auf die Objektvariable, sondern auf die Klasse anwenden. *TForm1.Create(ParentForm)* reserviert den Speicher für ein neues dynamisches Objekt, initialisiert das Objekt und liefert es als Ergebnis zurück, so dass Sie es der Objektvariablen zuweisen können:

```
DynamicForm := TForm1.Create(ParentForm);
```

Konstruktoraufruf ohne Objekterzeugung

Sie können einen Konstruktor auch wie oben als Element eines bestehenden Objekts aufrufen (*DynamicForm.Create*). In diesem Fall wird kein neues Objekt erstellt, sondern der Konstruktor erhält *DynamicForm* als *self*-Parameter. In diesem Fall muss das Objekt also schon vorher mit einem Konstruktor initialisiert worden sein. Normalerweise ist es nur dann notwendig, auf solche Weise einen Konstruktor aufzurufen, wenn Sie selbst einen neuen Konstruktor anlegen und in diesem den geerbten Konstruktor aufrufen. In diesem Fall müssen Sie den Aufruf jedoch mit Hilfe des Schlüsselworts *inherited* durchführen, wie der nächste Abschnitt zeigen wird.

Eigene Konstruktoren schreiben

Wenn Sie eigene nichtvisuelle Objektklassen schreiben, benötigen Sie statt der Bearbeitung des *OnCreate*-Ereignisses einen eigenen Konstruktor. Sie deklarieren diesen wie eine Methode, verwenden allerdings das Schlüsselwort *constructor*:

```
type
  TGraphicElement = class(TPersistent)
    constructor Create(InitRect: TRect);
     { der Name "Create" ist nicht vorgeschrieben }
```

Während die meisten Konstruktoren der VCL die Besitzerkomponente als Parameter erwarten, können Sie Ihrem Konstruktor die Parameter geben, die Sie zur Objektinitialisierung benötigen. Im obigen Beispiel ist das ein Rechteck, dessen Koordinaten zur Initialisierung des *TGraphicElement*-Objekts verwendet werden sollen.

Innerhalb eines Konstruktors müssen Sie den Konstruktor der Basisklasse aufrufen, damit auch diese sich richtig initialisieren kann:

```
constructor TGraphicElement.Create(InitRect: TRect);
begin
  inherited Create;
  ...
```

Der Aufruf des geerbten Konstruktors erzeugt kein neues Objekt, da er ohne die Voranstellung der Klasse stattfand (siehe voriger Abschnitt *Konstruktoraufruf ohne Objekterzeugung*).

Hinweis: Wenn die Klasse, von der Sie eine neue Klasse ableiten, einen virtuellen Konstruktor deklariert, sollten Sie nicht irgendeinen neuen Konstruktor schreiben, sondern den virtuellen Konstruktor der Basisklasse überschreiben. Virtuelle Methoden und Konstruktoren werden später in diesem Kapitel behandelt, das Überschreiben virtueller Konstruktoren ist beispielsweise bei der Entwicklung eigener Komponentenklassen notwendig (siehe z.B. Kapitel 6.5.2, Konstruktor *TScrollBoxEx.Create*).

Hinweis für Borland-Pascal-Umsteiger: In Borland Pascal hätten Sie statt *inherited Create* auch *TPersistent.Create* schreiben können, um den geerbten Konstruktor aufzurufen, ohne dass dadurch ein weiteres neues Objekt konstruiert worden wäre. In Object Pascal würde der Aufruf *TPersistent.Create* ein neues *TPersistent*-Objekt erzeugen und als Funktionsergebnis zurückliefern (wobei dieses Ergebnis verloren gehen würde, weil Sie es keiner Variablen zugewiesen haben). In Object Pascal *müssen* Sie also mit der Direktive *inherited* arbeiten.

Der oben gezeigte Unterschied zwischen dem Aufruf *TClass.Create* und *ObjectVar.Create* wird von dem Code, den der Compiler erzeugt, automatisch berücksichtigt. Wenn die erste Zeile Ihres Konstruktors erreicht ist, ist der Speicher für das Objekt in jedem Fall bereits reserviert. Falls mit dem Konstruktoraufruf ein neues Objekt erstellt wurde, ist dieser Speicher außerdem komplett mit 0 initialisiert. Sie brauchen daher Variablen, deren binäre Form nur aus Nullen bestehen soll (z.B. Zahlen, die 0 sind, leere Mengen und *False*-Werte), im Konstruktor nicht einzeln zu initialisieren.

2.3.2 Aufräumen mit Destruktoren

Das Gegenstück zu den Konstruktoren bilden die Destruktoren. Grundsätzlich sollte jedes Objekt, für das in irgendeinem Teil des Programms Speicher reserviert wurde, an einer anderen Stelle wieder freigegeben werden.

In der Praxis müssen Sie Destruktoren jedoch noch seltener von Hand aufrufen als Konstruktoren. Selbstverständlich gibt die VCL alle Objekte wieder frei, die sie selbst erzeugt hat (also die Komponenten des Formulars und das Formular selber). Darüber hinaus gibt sie auch die von Ihnen erzeugten dynamischen Objekte frei, wenn es sich um Komponenten handelt und Sie diese in die Besitzhierarchie eines Formulars eingefügt haben. Wenn nämlich eine Komponente (oder ein Formular) gelöscht wird, gibt die VCL alle von dieser Komponente besessenen Komponenten frei, indem sie ihren Destruktor aufruft.

Sie müssen also nur für die Objekte einen Destruktor aufrufen, für die Sie einen Konstruktor aufgerufen haben, die Sie aber nicht in andere Komponenten, die von der VCL verwaltet werden, eingefügt haben.

Destruktoren aufrufen

Wenn die genannte Voraussetzung nicht zutrifft und Sie das Objekt selber freigeben müssen, können Sie den Destruktor direkt aufrufen:

```
DynamicObject.Destroy;
```

Empfehlenswerter ist es jedoch, statt dessen die Methode *Free* aufzurufen. Diese testet, ob das Objekt überhaupt initialisiert wurde, und ruft den Destruktor nur dann auf, wenn das der Fall ist. Sie können auch selbst überprüfen, ob ein Objekt bereits initialisiert wurde, indem Sie die Standardprozedur *Assigned* verwenden:

```
{ entspricht dem Aufruf von "Free": }
if Assigned(DynamicObject)
   then DynamicObject.Destroy;
```

Assigned funktioniert nur dann zuverlässig, wenn alle nicht-initialisierten Objekte auch am Wert *nil* erkennbar sind. Objekte, die Teil eines Formulars oder anderen Objekts sind, werden immer mit *nil* vorbelegt, da der Speicher eines Objekts mit Nullen initialisiert wird. Nachdem Sie aber ein Objekt mit *Free* freigegeben haben, wird es nicht automatisch auf *nil* zurückgesetzt. Dies müssen Sie manuell erledigen, falls Sie möglicherweise später mit *Assigned* abfragen wollen, ob das Objekt initialisiert ist:

```
DynamicObject := nil;
```

Destruktoren schreiben

Um einen Destruktor selbst zu schreiben, deklarieren Sie eine Methode mit dem Schlüsselwort *destructor*. Die Sprache Object Pascal erlaubt hier zwar wieder einen beliebigen Namen, da jede Klasse in Object Pascal aber ein Nachfahre der vordefinierten Klasse *TObject* ist, empfiehlt es sich jedoch, den schon von *TObject* vordefinierten virtuellen Destruktor *Destroy* zu überschreiben. In jedem Fall sollte ein Destruktor am Schluss auch einen geerbten Konstruktor aufrufen, im Normalfall also *Destroy*:

```
destructor TGraphicElement.Destroy;
begin
  ...
  inherited Destroy;
end;
```

Destruktoren haben weniger zu tun als Konstruktoren, da sie oft nur die Speicherreservierungen rückgängig machen, die im Konstruktor explizit vorgenommen wurden. Wenn Ihr Konstruktor nur einige Variablen mit Standardeinstellungen belegt, benöti-

gen Sie oft keinen eigenen Destruktor. Ausnahmen von dieser Regel: Wenn andere Methoden des Objekts Speicher zur privaten Verwendung innerhalb des Objekts reservieren, ihn aber selbst nicht freigeben, sollte dies spätestens im Destruktor nachgeholt werden.

Auch für andere Arten von Systemressourcen wie z.B. geöffnete Dateien gilt, dass sie (spätestens) im Destruktor freigegeben werden sollten. Wenn Sie die Klassen der VCL verwenden, sind diese Systemressourcen jedoch meistens in Objekten gekapselt, so dass Sie sich um die Ressourcen selbst nicht zu kümmern brauchen (Dateien können z.B. mit *TStream*-Objekten verwaltet werden).

Automatischer Destruktoraufruf

Object Pascal unterstützt zwar nicht die automatische Konstruktion und Destruktion von Objekten wie C++, in einem speziellen Fall kommt es jedoch zu einem automatischen Aufruf des Destruktors: Wenn innerhalb des Konstruktors eine Exception auftritt, wird automatisch der Destruktor aufgerufen, um den Speicher wieder freizugeben. (Dieser Mechanismus ist nur aktiv, wenn der Konstruktor auf eine Klasse (genauer: Klassenreferenz) angewendet wurde, um ein neues Objekt zu initialisieren.)

Ihr Destruktor sollte auf den Fall, dass der Konstruktor durch eine Exception unterbrochen wird, vorbereitet sein. Im folgenden Beispiel könnte es Probleme geben:

```
constructor DemoObject.Create;
begin
  inherited.Create;
  OpenFile;
  GetMem(DynArray, 20000);
end;

destructor DemoObject.Destroy;
begin
   { der folgende Code ist nicht sicher,
    wenn der Konstruktor durch eine Exception unterbrochen wird:
  CloseFile;
  FreeMem(DynArray, 20000);
  inherited Destroy;
end;
```

Falls in *Create* während der Operation *OpenFile* eine Exception auftritt, wird der Konstruktor sofort abgebrochen und der Destruktor aufgerufen. Die Variable *DynArray* ist zu diesem Zeitpunkt noch nicht initialisiert. Der Aufruf von *FreeMem* würde also nicht erforderlich sein und würde sogar zu einem Fehler führen. Statt dessen könnte der Destruktor überprüfen, ob die Initialisierungen überhaupt vorgenommen worden sind:

```
destructor DemoObject.Destroy;
begin
  if FileIsOpen then
    CloseFile;
  if Assigned(DynArray) then
    FreeMem(DynArray, 20000);
  inherited Destroy;
end;
```

Freigabe von aktiven Objekten *R31*

Problematisch ist es auch, Objekte zu löschen, die noch aktiv sind. So können Sie in einer *OnClick*-Bearbeitung für einen Schalter diesen Schalter nicht löschen, da die VCL den Schalterdruck eventuell noch weiter bearbeiten will und nicht damit rechnet, dass der Schalter plötzlich verschwindet.

Ebenso wenig dürfen Sie die Methode *Free* eines Formulars innerhalb einer Ereignisbearbeitungsmethode dieses Formulars aufrufen. Es ist jedoch möglich, aktive Formulare zu schließen, indem Sie die Methode *Close* oder (in Spezialfällen) *Release* verwenden.

2.3.3 Polymorphie durch virtuelle Methoden

Im Laufe dieses Kapitels werden wir noch des Öfteren auf Spracheigenschaften stoßen, die von der Komponentenbibliothek aufgewertet – oder die im Gegenteil durch die Nutzung dieser Bibliothek weniger wichtig geworden sind. Die virtuellen Methoden erhalten in Object Pascal gleich mehrfach Konkurrenz: Zum einen können Sie Methoden explizit als *dynamic* deklarieren, was jedoch am hier beschriebenen Prinzip nichts ändert, zum anderen macht die VCL bei den Events massiven Gebrauch von den neuen Methodenzeigern (siehe Kapitel 2.5.5).

Methodenzeiger oder virtuelle Methoden?

Das Ergebnis dieses Konkurrenzkampfs ist, dass die virtuellen Methoden für den Entwurf eines Formulars keine Rolle mehr spielen. Statt dessen werden die mit Hilfe der visuellen Tools einfacher handzuhabenden Ereignisbearbeitungsmethoden verwendet. In allen anderen Bereichen bleiben die virtuellen Methoden nahezu unverzichtbar. So brauchen Sie als Komponentenprogrammierer ein fundiertes Wissen über die Polymorphie mit virtuellen oder dynamischen Methoden. Auch wenn Sie fernab jeglicher Komponenten eigene Objektklassen entwickeln wollen, sind virtuelle Methoden meist erheblich besser geeignet als Methodenzeiger. Der große Vorteil der Methodenzeiger bei der visuellen Programmierung resultiert auch erst daraus, dass Delphi Ihnen bei ihrer an sich umständlichen Verwaltung eine Menge Routinearbeit abnimmt.

Zuweisungskompatibilität von Objekten

Die erste Voraussetzung für die Polymorphie (Vielgestaltigkeit) ist, dass ein Objekt nicht nur zu den Objekten kompatibel ist, die von derselben Klasse sind, sondern auch zu den Objekten, die eine von deren Vorfahr-Klassen haben.

Die Kompatibilität gilt nur in einer Richtung: Sie können einer Variablen einer bestimmten Klasse beliebige Objekte von abgeleiteten Klassen zuweisen:

```
var
  Object: TObject;  { TObject ist die oberste aller Klassen }
  Component: TComponent; { TComponent ist abgeleitet von TObject. }
  Form: TForm;  { TForm ist indirekt abgeleitet von TComponent. }
  Button: TButton; { TButton stammt ebenfalls von TComponent ab. }
...
Component := Form;
Component := Button;
Object := Component;
```

Da Formulare und Schalter alle Fähigkeiten von *TComponent* erben, können sie anstelle eines *TComponent*-Objekts (hier die Variable *Component*) verwendet werden. Da außerdem *TComponent* eine Nachfahr-Klasse von *TObject* ist, kann sie alles tun, was ein *TObject* auch tun kann, daher ist auch die letzte Zuweisung des obigen Beispiels erlaubt. Der umgekehrte Weg ist nicht möglich:

```
Form := Object; { Typ-Fehler }
```

Von einem Formular werden sehr viele spezielle Dinge erwartet, eines davon ist z.B., dass es auf dem Bildschirm sichtbar ist. Da *Object* aber irgendein Objekt sein kann, ist es sehr unwahrscheinlich, dass es allen Anforderungen der Klasse *TForm* genügt. Für die Zuweisung an eine *TForm*-Variable kommen also nur Objekte von *TForm* oder von davon abgeleiteten Klassen in Frage.

Das Ableiten einer Klasse von einer anderen führt also nicht nur dazu, dass die abgeleitete Klasse die Elemente der Basisklasse erbt, sondern es führt zu Verwandtschaftsverhältnissen, die bei der Polymorphie eine große Rolle spielen, wie wir gleich sehen werden.

Hinweis: In diesem Zusammenhang ist es wichtig, dass Objektvariablen nur Zeiger sind. Bei den Zuweisungen im obigen Beispiel werden nur diese Zeiger kopiert, die Objekte selbst werden nicht verändert. Andernfalls würden ja einige Daten verloren gehen, wenn Sie z.B. versuchten, den relativ kleinen Speicherbereich eines *TComponent*-Objekts mit einem umfangreichen *TForm*-Objekt zu füllen, das ja außer den von *TComponent* geerbten Daten viele weitere Daten enthält.

Rückblick auf die statischen Methoden

Zunächst einmal ist festzustellen, dass jede Methode, Prozedur, Funktion (und jeder beliebige Programmabschnitt) eine Adresse im Speicher des Computers hat. Die Methoden, die in den Beispielen des Kapitels 2.2 vorgekommen sind, waren alle statisch. Wenn Sie eine solche Methode aufrufen, generiert der Compiler Maschinenbefehle für einen direkten Sprung zu dieser Adresse wie bei einem normalen Prozedur- oder Funktionsaufruf. Sobald Vererbung ins Spiel kommt, kann dieses Verhalten jedoch unerwünscht werden und sogar den Sinn der Vererbung gefährden.

Nehmen wir an, Sie haben verschiedene Klassen, die alle auf eine verschiedene Weise arbeiten. Die Arbeit ist in einer Prozedur *Arbeite* definiert. Jede Klasse besitzt eine eigene Prozedur *Arbeite*, die eine individuelle Arbeit verrichtet:

```
type
  AbstrakteKlasse = class(TObject)
  { Konstruktor Create wird von TObject geerbt }
    procedure Arbeite;
  end;
  Klasse1 = class(AbstrakteKlasse) procedure Arbeite; end;
  Klasse2 = class(AbstrakteKlasse) procedure Arbeite; end;
  ...
  Klasse10 = class(AbstrakteKlasse) procedure Arbeite; end;
```

Sie lassen nun den Benutzer Ihrer Anwendung auswählen, welche Klasse er verwenden möchte. Je nach Auswahl des Benutzers erzeugen Sie dynamisch ein Objekt der gewünschten Klasse und weisen es der Variablen *Objekt* zu:

```
var
  Objekt: AbstrakteKlasse;
begin
  case Benutzerauswahl of
    { "Benutzerauswahl" kann z.B. sein: (Sender as TButton).Tag }
     Button1: Objekt := Klasse1.Create;
     Button2: Objekt := Klasse2.Create;
  ...
     Button10: Objekt := Klasse10.Create;
```

Wir nehmen weiter an, dass Sie an einer anderen Stelle die Arbeit der Klasse ausführen wollen, die der Benutzer ausgewählt hat:

```
Objekt.Arbeite;
```

Wie soll der Compiler nun diesen Methodenaufruf in Maschinenbefehle umwandeln, bzw. welche *Arbeiten*-Methode soll er aufrufen? Schließlich kann er nicht wissen, ob *Objekt* zur Laufzeit den Typ *Klasse1*, *Klasse2* oder sonst einen Typen hat. Da *Objekt* als Instanz der Klasse *AbstrakteKlasse* deklariert wurde, ruft der Compiler in diesem Beispiel die Methode *AbstrakteKlasse.Arbeite* auf, obwohl *Objekt* vorher mit einer anderen Klasse initialisiert wurde. Es gibt jedoch eine Lösung für dieses Problem.

Virtuelle Methoden

Wenn Sie bisher nur die Methodenzeiger und Ereignisproperties kennen, werden Ihnen diese sicher als Lösungsmöglichkeit einfallen: Die Basisklasse (*AbstrakteKlasse*) muss lediglich einen Methodenzeiger, z.B. *OnArbeite*, beinhalten, der von den abgeleiteten Klassen entsprechend gesetzt wird. Der obige Methodenaufruf müsste dann mit Hilfe dieses Zeigers stattfinden:

```
Objekt.OnArbeite;
```

Tatsächlich funktionieren die virtuellen Methoden sehr ähnlich wie in diesem Konzept, sind aber viel einfacher zu handhaben: Das Einzige, was Sie im obigen Beispiel ändern müssen, um von der statischen Bindung zur dynamischen überzugehen, ist, in der Basisklasse ein *virtual* hinter den Methodenkopf von *Arbeite* zu schreiben und bei den abgeleiteten Klassen ein *override*:

```
type
  AbstrakteKlasse = class
  { Konstruktor Create wird von TObject geerbt }
    procedure Arbeite; virtual;
  end;
  Klasse1 = class(AbstrakteKlasse)
    procedure Arbeite; override;
  end;
  ...
.. weiterer Code wie oben ...
..
  Objekt.Arbeite; { ruft immer die richtige Methode auf }
```

Je nachdem, welches Objekt also in der oben abgedruckten *case*-Abfrage in der Variablen *Objekt* gespeichert wurde, wird für die Anweisung *Objekt.Arbeite* jetzt eine der Methoden *Klasse1.Create, Klasse2.Create* usw. aufgerufen. Die Methode *AbstrakteKlasse.Create* wird überhaupt nicht mehr verwendet (sie könnte daher als *abstract* deklariert werden, wie einer der folgenden Abschnitte zeigen wird).

Override

Der Vorgang, eine geerbte virtuelle Methode neu zu definieren, wird als *Überschreiben* der geerbten Methode bezeichnet. Die Angabe der Direktive *override* ist zwingend erforderlich. Wenn Sie sie weglassen, erzeugt der Compiler eine eigenständige Methode, die nichts mit der geerbten Methode zu tun hat und die im obigen Beispiel (in dem die Variable *Objekt* von einer abstrakten Klasse ist) nicht aufgerufen worden wäre. Aus der Sicht der neuen Klasse verdeckt eine solche, nicht mit *override* deklarierte Methode die geerbte Methode gleichen Namens.

Dies ist besonders wichtig für Benutzer von Borland Pascal, da es dort das Wort *override* noch nicht gab und virtuelle Methoden *immer* durch gleichnamige Methoden in

abgeleiteten Klassen überschrieben wurden. Allerdings gibt Delphi per Voreinstellung eine Warnung aus, wenn Sie eine geerbte virtuelle Methode verdecken.

Verwechseln Sie *override* außerdem nicht mit der Direktive *overload*. *Overload* wird beim Definieren mehrerer gleichnamiger Methoden, die als Alternativen verwendbar sein sollen, benötigt und hat nichts mit Vererbung zu tun (siehe Kapitel 2.5.4).

Reintroduce

Außerdem gibt es durch die Direktive *Reintroduce* noch eine Möglichkeit, eine geerbte virtuelle Methode im oben beschriebenen Sinne zu verstecken, ohne dass der Compiler eine Warnung ausgibt (ab Delphi 4). Fügen Sie dazu einfach den Zusatz »*Reintroduce;*« an das Ende der Methodendeklaration an.

Wenn Sie also in einer abgeleiteten Klasse eine Methode definieren, deren Name mit dem einer geerbten Methode übereinstimmt, sollten Sie entweder *Reintroduce* oder *Override* angeben, damit der Compiler keine Warnung erzeugt und auch einem Menschen beim Lesen des Quelltextes sofort deutlich wird, wie sich die Deklaration der neuen Methode auswirkt. Die *Override*-Direktive ist natürlich nur erlaubt, wenn Parameterliste und Rückgabetyp mit der überschriebenen Methode übereinstimmen, diese Forderung gilt für *reintroduce* nicht.

Überschreiben von Methoden und Inherited

Alles, was Sie tun müssen, um den Mechanismus der virtuellen Methoden zu aktivieren, ist, die Methode dort, wo sie zum ersten Mal in der Hierarchie auftritt, als *virtual* zu deklarieren und in allen Ableitungen als *override*. Die Methode, die durch die neue überschrieben wird, sollten Sie in den meisten Fällen innerhalb der neuen aufrufen, damit deren Funktionalität nicht verloren geht. Dazu verwenden Sie den Namen der Methode, stellen ihm aber das Schlüsselwort *inherited* voran:

```
procedure TForm1.WndProc(var Message: TMessage);
begin
  inherited WndProc(Message);
  {... eigene Anweisungen }
```

Hinweis: Während Sie in der Deklaration der Methode *WndProc* die Direktive *override* nicht vergessen dürfen, fehlt diese Direktive hinter dem obigen Methodenkopf. Auch für alle anderen Direktiven gilt, dass Sie sie nur bei der erstmaligen Deklaration, also innerhalb der Klassendeklaration, anzugeben brauchen.

Vorteile virtueller Methoden gegenüber einfachen Methodenzeigern

Mit Methodenzeigern hätten Sie für jede virtuelle Methode eine zusätzliche Variable deklarieren und im Konstruktor der abgeleiteten Objekte diese auf die richtige Methode setzen müssen (auf diese Weise könnte man dann gleich auf OOP-Sprachen verzichten und das OOP in rein prozeduralen Sprachen wie Pascal oder C simulieren).

Virtuelle Methoden haben noch einen weiteren Vorteil: Wenn Sie mehrere virtuelle Methoden in einer Klasse deklarieren, wird nicht für jede davon ein Methodenzeiger in jeder Objektinstanz abgelegt. Statt dessen erhält jede Objektinstanz einen Zeiger auf eine *VMT* (Virtual Method Table), die nur ein einziges Mal für alle Objekte derselben Klasse gespeichert werden muss. In dieser Tabelle sind sämtliche virtuelle Methoden, die geerbt und / oder überschrieben sind, so abgelegt, dass der Compiler sie schnell finden kann. Der Aufruf einer virtuellen Methode läuft damit intern so ab:

```
{ Der Aufruf ... }
   Objekt.Methode(Parameter);
{ ... wird ähnlich behandelt wie folgt: }
   Objekt.VMT^[MethodenIndex](Parameter);
{ allerdings ist der VMT-Zeiger nicht mit
  normalen Zeigern kompatibel und kann nicht
  unter dem Namen "VMT" angesprochen werden. }
```

Der VMT-Zeiger des Objekts wird wie ein normales Datenelement desselben gelesen. Zu dieser VMT-Adresse rechnet das Programm den Index, an dem sich die Adresse der für dieses Objekt gültigen virtuellen Methode befindet, hinzu. Dieser Tabelleneintrag wird ausgelesen und das Programm springt an die darin angegebene Adresse (nachdem die Parameter für die Methode inklusive des *self*-Parameters auf den Stack gelegt wurden).

Dynamische Methoden

Dynamische Methoden sind von der Verwendung her mit den virtuellen Methoden identisch. Statt der Direktive *virtual* verwenden Sie einfach die Direktive *dynamic*, um eine dynamische Methode zu erhalten. Alles andere inklusive des Überschreibens der Methode mit *override* bleibt.

Die dynamische Art, die Methodenadressen zu verwalten, ist noch speicherplatzsparender als die VMTs. Der Compiler legt für die dynamischen Methoden jeder Klasse eine eigene Tabelle (DMT) an. Die Platzersparnis resultiert aus der Tatsache, dass dort Methoden, die zwar geerbt, aber nicht überschrieben werden, nicht aufgeführt werden. Vielleicht fragen Sie sich, ob es überhaupt notwendig ist, den geringen Speicherbedarf von VMTs weiter zu reduzieren. Erst bei 100 Klassen mit jeweils 100 Methoden kämen 10000 VMT-Zeiger, also ca. 40 KByte VMTs zusammen, in 32-Bit-Umgebungen immer noch ein Klacks.

Die wichtigste Aufgabe der dynamischen Methoden ist daher nicht das Einsparen von Speicher, sondern die Bearbeitung von Windows-Nachrichten. Hier kommt der entscheidende Vorteil der dynamischen Methoden ins Spiel: Sie können mit einer Integer-Konstante verknüpft sein. Jeder dynamischen Methode, die eine Windows-Botschaft bearbeiten soll, wird ein Index zugeordnet, der dem Nachrichtencode der Windows-Nachricht entspricht. Diese Methoden werden allerdings nicht mit *dynamic*, sondern mit der *message*-Direktive deklariert, siehe Kapitel 6.4.1.

Wenn Sie nicht mit derartigen indizierten Methoden arbeiten, sind wahrscheinlich immer die schnellen virtual-Methoden günstiger, es sei denn, Sie haben davon mehrere 1000 und arbeiten noch unter Windows 3.1.

Polymorphie

Das Gesamtkonzept, das auf der Vererbung und der dynamischen Bindung basiert, heißt *Polymorphie* (Vielgestaltigkeit). Ein polymorphes Objekt ist eine Objektvariable, von der der Compiler nicht weiß, welche Klasse diese Variable zur Laufzeit haben wird. Tatsächlich kann die Klasse sich zur Laufzeit mehrfach ändern, wenn Sie einem polymorphen Objekt Objekte anderer Klassen zuweisen.

Ein polymorphes Objekt kann also in vielen Gestalten auftreten, wie im obigen Beispiel die Variable *Objekt*. Polymorphe Objekte kommen bei verschiedenen Gelegenheiten vor:

- als Properties: z.B. einige Properties, die Stringlisten enthalten, wie etwa *TListBox.Items*. Dieses Property ist als *TStrings* deklariert, basiert aber auf einem Objekt einer davon abgeleiteten Klasse.
- als Parameter von Ereignisbearbeitungsmethoden: z.B. der Parameter *Sender*, der jedes beliebige Objekt transportieren kann.
- als Parameter von Methoden, die Sie aufrufen können: z.B. die Methode *TComponent.InsertComponent*, mit der Sie eine beliebige Komponente zum Besitz einer anderen Komponente hinzufügen können.

Eine weitere Aufwertung erfährt die Polymorphie durch virtuelle Konstruktoren, die in Kapitel 2.3.4 besprochen werden.

Abstrakte Klassen *R129*

In großen Vererbungshierarchien wie der VCL kommt es häufig vor, dass Klassen nur dazu da sind, die Gemeinsamkeiten verschiedener anderer Klassen zu definieren. Dabei können diese Klassen so weit gehen, dass sie zwar eine virtuelle Methode deklarieren, die in allen abgeleiteten Klassen überschrieben wird, dass sie diese selbst jedoch gar nicht implementieren.

Um eine derartige Methode zu deklarieren, die nur als Platzhalter für die Methoden abgeleiteter Klassen dient, können Sie sie mit der Direktive *abstract* als *abstrakte Methode* deklarieren. Dieser Direktive muss entweder *dynamic* oder *virtual* vorangehen, da eine abstrakte Methode, die nicht überschrieben werden kann, keinen Sinn ergibt. Die Methode *Arbeite* der Klasse *AbstrakteKlasse* aus dem obigen Beispiel könnte also auch so deklariert worden sein:

```
procedure Arbeite; virtual; abstract;
```

Diese Deklaration hat den Vorteil, dass Sie die Methode nicht im Implementationsteil der Unit definieren müssen und dass ein Aufruf dieser Methode zu einem Laufzeitfehler mit der Nummer 210 führt, wodurch sichergestellt ist, dass diese Methode nicht versehentlich aufgerufen wird. In Methoden abgeleiteter Klassen, die abstrakte Methoden überschreiben, dürfen Sie daher die geerbte abstrakte Methode auch nicht mit *inherited* aufrufen.

Beispiele für abstrakte Klassen der VCL sind *TStream*, *TStrings* und *TPersistent*.

Laufzeit-Typinformation

Das Konzept der Polymorphie ist sicher faszinierend, vor allem, wenn man vorher nur prozedural programmiert hat und sich langsam an die Vielgestaltigkeit gewöhnt; aber manchmal ist es vielleicht doch etwas unheimlich, dass man überhaupt nicht weiß, welche Klasse sich hinter dem vielgestaltigen Objekt zur Laufzeit tatsächlich verbirgt.

Anders gesagt: Manchmal ist es notwendig, hilfreich oder auch nur sicherer, wenn zur Laufzeit abgefragt werden kann, welchen Typ ein Objekt hat. Dies wäre zwar schon mit einer virtuellen Methode wie *WelchenTypHabeIch* realisierbar, aber die Typinformation, die Sie zur Laufzeit erhalten können, ist noch weitaus umfangreicher. Die Typinformationen in Object Pascal hängen eng mit Klassenreferenzen und Klassenfunktionen zusammen, die das Thema der nächsten beiden Abschnitte sind.

2.3.4 TClass – die Klasse der Klassen

Für die Implementation von virtuellen Konstruktoren müssen in Object Pascal nicht nur die Objekte einer Klasse, sondern auch die Klasse selbst zur Laufzeit ansprechbar sein. Hierzu gibt es die *Klassenreferenz* (in der deutschen Online-Dokumentation finden Sie den Begriff *Objektbezug*, der hier aus Gründen der Verständlichkeit vermieden wird). In manchen anderen OOP-Sprachen finden Sie für die Klassenreferenz die Bezeichnung *Metaklasse*.

Klassenreferenzen *R128*

Die Unit *System* enthält die Deklaration des Typs für eine grundlegende Klassenreferenz:

```
type
  TClass = class of TObject;
```

Verwechseln Sie *TClass* nicht mit anderen Klassen, die mit *class*, aber ohne *of* deklariert werden. Eine Klassenreferenz erhalten Sie, indem Sie einer Variablen den Klassenreferenz-Typ zuweisen:

```
var
  ClassRef: TClass;
```

Sie können sich eine Klassenreferenz als Objekt vorstellen, das verschiedene Informationen über die Klasse speichert, wie z.B. den Namen der Klasse, ihre Basisklasse usw.

Grundsätzlich können Sie eine Klassenreferenz im Programmcode so verwenden wie einen Klassennamen. Die Vorteile einer Klassenreferenz gegenüber einer direkten Klassenangabe liegen wieder einmal in der Polymorphie. In Kapitel 2.3.3 hatten wir das folgende Beispiel, in dem je nach Wahl des Benutzers eine Instanz einer bestimmten Klasse erzeugt werden sollte:

```
case Benutzerauswahl of
  { Benutzerauswahl kann z.B. sein: (Sender as TButton).Tag }
   Button1: Objekt := Klasse1.Create;
   Button2: Objekt := Klasse2.Create;
...
   Button10: Objekt := Klasse10.Create;
```

Unbefriedigend ist dabei, dass wir zwar jedes beliebige Objekt erzeugen können, dass wir aber die Klasse, die der Benutzer ausgewählt hat, nicht in einer Variablen speichern können. Nachdem die Variable *Objekt* erzeugt ist, stehen deren virtuelle Funktionen zwar wie gewünscht zur Verfügung und begründen die Polymorphie des Objekts *Objekt*; wenn nun aber an einer völlig anderen Stelle des Programms ein zweites Objekt derselben (von der Benutzerauswahl abhängenden) Klasse erzeugt werden soll, wäre erneut eine lange *case*-Anweisung wie oben erforderlich (oder eine eigene Prozedur, die diese *case*-Anweisung zum mehrmaligen Gebrauch zur Verfügung stellt).

Verwendungsbeispiel

Mit den Klassenreferenzen können Sie das Problem in Object Pascal jedoch auf viel elegantere Weise lösen:

```
type
  AbstrakteKlasseRef = class of AbstrakteKlasse;
var
  GewaehlteKlasse: AbstrakteKlasseRef;
```

```
  Objekt: AbstrakteKlasse;
begin
  Objekt := GewaehlteKlasse.Create;
```

Mit der obigen Anweisung erreichen Sie dasselbe wie mit dem langen *case*-Block aus vorhergehenden Beispielen. Natürlich muss *ClassRef* schon vorher den gewünschten Wert erhalten haben. Wenn dieser wieder von der Benutzerauswahl abhinge, wäre dazu immer noch eine *case*-Anweisung wie die obige erforderlich:

```
case Benutzerauswahl of
  Button1: GewaehlteKlasse := Klasse1;
  Button2: GewaehlteKlasse := Klasse2;
usw.
```

Es ist jedoch ein klarer Vorteil, dass diese *case*-Anweisung nur ein einziges Mal erforderlich ist und die Wahl des Benutzers als *ClassRef* gespeichert werden kann. Eine Klassenreferenz ist also auch vorstellbar als eine Variable, die kein Objekt, sondern eine Klasse enthält.

Eigene Klassenreferenzen deklarieren

Sie können – wie im letzten Beispiel geschehen – auch selbst Klassenreferenztypen der Art *class of* deklarieren:

```
type
  TMyClassReference = class of TMyClass;
  AbstrakteKlasseRef = class of AbstrakteKlasse;
```

Eine solche Deklaration dient lediglich dazu, einen festen Namen für einen Klassenreferenztypen zu vergeben. Der Compiler erzeugt sowieso zu jeder Klasse automatisch eine Klassenreferenz, womit auch der Typ dieser Klassenreferenz existiert. Mit der *class-of*-Deklaration weisen Sie diesem Typen *zusätzlich* einen eigenen Typenbezeichner zu. Einen solchen Typenbezeichner benötigen Sie immer, wenn Sie eine Klassenreferenz-Variable deklarieren wollen, wie in einem früheren Beispiel die Variable *GewaehlteKlasse*. Mit der Methode *TObject.ClassType* können Sie übrigens auch ohne eine solche Deklaration jederzeit in Besitz einer Klassenreferenz für die Klasse eines beliebigen Objekts gelangen, siehe dazu Kapitel 2.3.6.

Virtuelle Konstruktoren

Der Code im Abschnitt *Verwendungsbeispiel* bringt noch eine Gefahr mit sich: Zwar wird mit der Zeile

```
Objekt := GewaehlteKlasse.Create;
```

immer ein Objekt der richtigen Klasse erzeugt, trotzdem wird jedoch nur der von der Klasse *AbstrakteKlasse* bereitgestellte Konstruktor aufgerufen, der an die abgeleiteten Klassen vererbt wird. Falls diese abgeleiteten Klassen wichtige eigene Initialisierungen

in eigenen Konstruktoren vornehmen müssen, so sollten diese mit der obigen Zeile ebenfalls aufgerufen werden. Damit das geschieht, muss *AbstrakteKlasse* den *Create*-Konstruktor als *virtual* deklarieren, und die abgeleiteten Klassen (*Klasse1*, *Klasse2* usw., siehe oben) müssten diesen mit *override* überschreiben.

Virtuelle Konstruktoren sind auch nur dann sinnvoll einsetzbar, wenn sie wie in diesem Beispiel über eine Klassenreferenz aufgerufen werden. In allen anderen Fällen steht schon bei der Übersetzung fest, welcher Konstruktor aufgerufen werden soll, weshalb dann die statische Bindung genügt.

Konstante Klassenreferenzen

Der normale Konstruktoraufruf mit Hilfe einer Klassenangabe, der in diesem Kapitel schon mehrfach angesprochen wurde, macht ebenfalls Gebrauch von einer Klassenreferenz, allerdings ist die Klassenreferenz hier nicht in einer Variablen gespeichert, sondern wird direkt über den Klassennamen angegeben:

```
Stringliste := TStringList.Create;
```

Kompatibilität

Für Klassenreferenzen gelten dieselben Kompatibilitätsregeln wie für Objekte. Das bedeutet, dass Sie einer Klassenreferenz vom Typ *TClass* jede beliebige Klassenreferenz zuweisen können, genauso wie Sie einem Objekt vom Typ *TObject* jedes beliebige Objekt zuweisen können, da *TObject* eine Vorfahrenklasse aller anderen Klassen ist.

Die Unit *Forms* deklariert beispielsweise den folgenden Klassenreferenztypen:

```
type
  FormClass = class of TForm;
```

Einer Variablen dieses Typs dürfen Sie nur eine Klasse zuweisen, die von *TForm* abgeleitet ist, beispielsweise die Klasse eines Ihrer Formulare oder die Klasse *TForm* selber.

2.3.5 Klassenmethoden

Eine interessante Erweiterung der OOP-Fähigkeiten früherer Pascal-Versionen stellen die Klassenmethoden dar. Die Bezeichnung *Klassenmethode* meint hier weniger »Methoden der Klasse« als eher »Methoden für die Klasse«, also Methoden, die nicht für spezifische Objekte, sondern für die Klasse selbst aufgerufen werden (C++-Programmierer werden dies von den statischen Elementen der C++-Klassen kennen). Ein rein fiktives Beispiel ist:

```
type
  TDemoClass = class
    class function Arbeitsgeschwindigkeit: Byte; virtual;
```

```
  { mit dieser Funktion kann die Klasse versprechen, besonders
    schnell zu arbeiten (falls Rückgabewert > 100) }
```

Klassenmethoden erhalten als *self*-Parameter kein Objekt ihres Klassentyps, sondern ein Objekt ihres Klassenreferenztyps (also eine Klassenreferenz). Als Konsequenz daraus können Klassenmethoden nicht auf die Variablen eines Objekts zugreifen. Dies sollte normalerweise auch nicht nötig sein, da es ja die Aufgabe der Klassenmethoden ist, Informationen über die gesamte Klasse zu liefern und nicht über einzelne Objekte. Sie können sich Klassenmethoden also auch als Methoden der Klassenreferenz vorstellen.

Oft sind Klassenmethoden nur sinnvoll, wenn sie auch virtuell sind. Die Methode im obigen Beispiel könnte beispielsweise die Unterschiede zwischen verschiedenen abgeleiteten Klassen deutlich machen, muss sich also in abgeleiteten Klassen überschreiben lassen. Wenn sie nicht virtuell wäre, würde sie kaum noch einen Sinn haben, da ihr Ergebnis vorhersehbar wäre.

Simulation von Klassenvariablen *R127*

Sinnvolle Beispiele für Klassenmethoden lernen Sie im nächsten Kapitel kennen. All diese Beispiele geben Informationen über die Klasse zurück; es wäre jedoch auch vorstellbar, dass Sie dadurch Einstellungen an der Klasse vornehmen. So könnte beispielsweise eine Rechteckklasse nicht nur über eine voreingestellte Breite und Höhe verfügen, sondern eine Klassenmethode anbieten, mit der Sie diese Voreinstellung ändern können:

```
type
  TRechteck = class
    class procedure SetStandardWH(W, H: Integer);
    ...
```

Leider hat diese Methode jedoch keine Möglichkeit, diese Daten direkt in der Klasse zu speichern, da es in Object Pascal keine Klassen*variablen* gibt. Allerdings können Sie sich in einem solchen Fall leicht mit globalen Variablen helfen, die Sie im Implementationsteil der Unit verbergen. Da die Klasse nur einmal vorkommt, sind globale Variablen hier zwar nicht elegant, aber aus technischer Sicht mit Klassenvariablen nahezu gleichwertig.

2.3.6 Typinformationen zur Laufzeit

Einen großen Fortschritt gegenüber der *Pascal-with-Objects*-Version vor Delphi stellen die in diesem Abschnitt vorgestellten Erweiterungen der Laufzeit-Typinformation dar. Der englische Fachbegriff für diese Typinformationen ist RTTI (Run Time Type Information) und ist aus der C++-Welt importiert. Als Delphi die RTTI in Object Pascal ein-

führte, war RTTI in der C++-Welt noch so neu, dass es noch nicht überall unterstützt wurde. Dabei enthalten die RTTI von Object Pascal noch deutlich mehr Informationen als die von C++.

TObject-Methoden

In Object Pascal erhalten Sie die Typinformationen, indem Sie Methoden der Klasse *TObject* aufrufen. Die folgende Tabelle fasst diese Methoden zusammen.

Funktionsart	Funktion	Ergebnistyp	Ergebnis
class function	ClassName	String	Name des Objekts oder der Klasse
class function	ClassNameIs(Str)	Boolean	Besagt, ob der Klassenname mit dem Parameterstring übereinstimmt
class function	ClassParent	TClass	Elternklasse des Objekts oder der Klasse
function	ClassType	TClass	Klasse des Objekts
class function	ClassInfo	Pointer	Zeiger auf ausführliche Informationen
class function	InheritsFrom(AClass)	Boolean	*True*, wenn die Klasse von *AClass* abgeleitet ist
class function	GetInterfaceTable	Zeiger auf *TInterfaceTable*	Informationen über die von einer Klasse implementierten Interfaces, siehe in der Online-Referenz auch die Funktion *GetInterfaceEntry*
function	GetInterface(IID, out)	Boolean	Ein Interface-Zeiger für ein bestimmtes, z.B. mit *IUnknown::QueryInterface* erfragtes Interface (siehe Kapitel 2.7.3)
class function	InstanceSize	Word	Größe einer Objektinstanz der Klasse

ClassName

Bei fünf der sechs Methoden handelt es sich um Klassenfunktionen, das heißt, dass sie (eigentlich) unabhängig von Objektinstanzen aufgerufen werden:

```
Name := TButton.ClassName;
```

In diesem Beispiel erhält *Name* den String mit dem Inhalt *'TButton'* zugewiesen. Interessant wird es erst, wenn das Ergebnis nicht schon von vornherein feststeht, wenn Sie also Klassennamen einer *polymorphen* Objektinstanz abfragen, z.B. in einer Ereignisbearbeitungsmethode, deren Parameter als *TObject* deklariert ist:

```
procedure TForm1.ButtonClick(Sender: TObject);
begin
  if Sender.ClassName = 'TSpeedButton' then
  ... else if Sender.ClassName = 'TButton' then ...
  { effizientere Abfragen mit dem is-Operator! }
```

Das Ergebnis von *ClassName* hängt nun von dem Objekt ab, das tatsächlich als Parameter *Sender* übergeben wurde. Die Methode *ButtonClick* kann auf diese Weise gleichzeitig Ergebnisse von zwei verschiedenen Schaltertypen verarbeiten und doch zwischen beiden unterscheiden (abgesehen davon, dass dieser spezielle Fall in der Praxis eher unwahrscheinlich ist).

Wie erklärt sich nun aber der Widerspruch zwischen der Aussage, dass Klassenmethoden unabhängig von Objektinstanzen sind, und der Tatsache, dass im letzten Beispiel ein Objekt und keine Klasse abgefragt wurde? Die Antwort ist einfach: Das Programm stellt zur Laufzeit fest, welche Klasse zur Instanz *Sender* gehört, und *für diese Klasse* wird dann die Methode *ClassName* aufgerufen. Diese ist dann wirklich nur noch von der Klasse abhängig. Tatsächlich sieht die Abfrage also wie folgt aus:

```
if Sender.ClassType.ClassName = 'TSpeedButton'
```

ClassType

Mit *ClassType* erhalten Sie die Referenz auf die Klasse selbst, also z.B.:

```
Ref := TObject.ClassType;
  { ergibt eine Referenz des Typs TClass }

Ref := TForm1.ClassType;  { oder }
Ref := Form1.ClassType;
  { ergibt eine Referenz des Typs "class of TForm1" }
```

ClassType ist keine Klassenmethode, d.h. dass sie nur mit Objekten verwendet werden kann. Eine Verwendung beispielsweise als *TClass.ClassType* wäre auch nicht sinnvoll, denn dann müssten Sie eine Referenz auf die Referenz erhalten.

ClassParent

Die Methode *ClassParent* liefert Ihnen eine Referenz auf die Elternklasse einer Klasse. Da sie nur den direkten Vorfahren einer Klasse angibt, können Sie damit nicht in einem Schritt ermitteln, ob eine Klasse von einer anderen abgeleitet ist (direkt oder indirekt). Das folgende Beispiel demonstriert die Verwendung von *ClassParent*:

```
function BaumEbene(Klasse: TClass): Integer;
begin
  Result := 0;
  { Wiederholen, bis die Klasse TObject erreicht ist: }
  while Klasse.ClassName <> 'TObject' do begin
    { Klassenreferenz der Basisklasse ermitteln: }
    Klasse := Klasse.ClassParent;
    inc(Result);
  end;
end;
```

Diese Funktion ermittelt, wie viele Ebenen sich die als Parameter übergebene Klasse unter der Klasse *TObject* befindet, sie gibt damit quasi die Entfernung der *Klasse* von *TObject* im Hierarchiebaum an. Diese Funktion ist im Beispielprogramm *DragDrop-Demo* eingebaut.

InheritsFrom

InheritsFrom beseitigt die Nachteile der Methode *ClassParent*, indem sie angibt, ob eine Klasse auf irgendeine Weise von einer anderen Klasse abgeleitet ist, ob sie also von dieser erbt. Die folgenden Ausdrücke sind alle wahr:

```
TButton.InheritsFrom(TControl) = True
TForm.InheritsFrom(TGraphicControl) = False
EineKlasseX.InheritsFrom(EineKlasseX) = True
```

Die Operatoren is und as

Die beiden Operatoren *is* und *as* sind dazu da, Typenumwandlungen und Abfragen von Vererbungsverhältnissen auf besonders elegante Weise durchführen zu können. Auf der rechten Seite beider Operatoren steht immer eine Objektinstanz oder eine Klassenreferenz, während auf der linken Seite nur eine Klassenreferenz stehen darf. *Is* testet, ob die Klasse des rechten Operanden ein Nachfahre der auf der linken Seite angegebenen Klasse ist. Die beiden folgenden Ausdrücke sind also gleichbedeutend:

```
(Object Is Class) = Object.InheritsFrom(Class)
```

As hat gegenüber einer herkömmlichen Typenumwandlung einen technischen Vorteil: Bevor dieser Operator einen Typ in einen anderen umwandelt, testet er, ob die Umwandlung überhaupt zulässig ist. Dabei testet *as*, ob der als Ergebnis gewünschte Typ kompatibel zum Typ vor der Umwandlung ist, es läuft also dieselbe Prüfung ab, die vom *is*-Operator durchgeführt wird (es muss gelten: »Ergebnistyp *is* Ausgangstyp«). Sind die Typen nicht kompatibel, erzeugt *as* eine *EInvalidCast*-Exception. Damit ist die Umwandlung *A := B as ClassType* gleichbedeutend mit:

```
if B is ClassType then
  A := ClassType(B) { normale Typenumwandlung durchführen }
else
  raise EInvalidCast; { Ausnahmebedingung ausrufen }
```

RTTI

Wie Sie oben gesehen haben, enthalten die Klassenreferenzen selbst keine Informationen außer der Klasse, zu der sie gehören. Über die Methoden von *TObject* können Sie Informationen über den Klassennamen und Vererbungsverhältnisse erhalten, Vergleiche zwischen Klassen sind über Vergleich der Referenzen oder mit dem *is*-Operator

möglich. Normalerweise werden Sie nicht mehr Informationen zur Laufzeit benötigen, denn welche Elemente Ihre Klassen enthalten, können Sie ja aus den Klassendeklarationen ablesen.

Trotzdem sind Informationen über veröffentlichte Properties und Methoden auch zur Laufzeit abfragbar. Die VCL selbst verwendet diese Informationen beispielsweise zur Speicherung der Properties, wenn Komponenten in eine Datei geschrieben werden.

Schlüssel zu diesen Informationen ist die Methode *ClassInfo*. Diese liefert zwar einen untypisierten Zeiger zurück, die Datenstruktur, auf die dieser Zeiger weist, ist aber nicht undokumentiert, sondern nur versteckt, da sie wie gesagt kaum gebraucht wird. Aus demselben Grund soll sie hier nur kurz erwähnt werden:

Die Unit TypInfo

Um einen bequemen Zugriff auf die Informationen von *ClassInfo* zu erhalten, müssen Sie die Unit *TypInfo* einbinden und den *Pointer*, den *ClassInfo* zurückliefert, in *PTypeInfo* umwandeln, z.B.:

```
uses TypInfo;

var
  TypeInfo: PTypeInfo;
  TypeData: PTypeData;
begin
  TypeInfo := PTypeInfo(TForm.ClassInfo)
  TypeData := GetTypeData(TypeInfo);
```

Schon haben Sie mit *TypeData* einen Zeiger auf einen höchst informativen Datenblock, dessen Struktur vom Wert von *TypeInfo.Kind* abhängt. Ist dieser beispielsweise *tkClass*, dann befindet sich in *TypeData*^ neben dem Verweis auf die Basisklasse und anderen Daten eine Liste von Properties, die in dieser Klasse deklariert sind. Auch der Aufbau dieser Liste ist in der Unit *TypInfo* in Form von Typendeklarationen dokumentiert. Wenn Sie also tief greifende RTTI-Informationen benötigen, sollten Sie sich an das Interface dieser Unit wenden (in der Standard-Version von Delphi unter DOC\TYPINFO.INT, sonst in SOURCE\VCL\TYPINFO.PAS).

2.3.7 Kompatibilität mit Borland Pascal

Das Objektmodel von Object Pascal führt dazu, dass die Deklaration der Variablen *Form: TForm* nicht, wie bei Borland Pascal, Platz für die gesamten Objektdaten von *TForm* reserviert. Diese Deklaration entspricht vielmehr der BP-Deklaration *Form: ^TForm* – es werden also genau vier Bytes für einen Zeiger belegt.

Dadurch ändert sich auch der Zugriff auf Methoden solcher dynamischer Objekte: Statt der Dereferenzierung wie *Objekt^.ZaehleFenster* findet jetzt ein einfacher Zugriff mit *Objekt.ZaehleFenster* statt, als ob das Objekt statisch wäre. Der Konstruktoraufruf ist bei Object-Pascal-Objekten auch dann erforderlich, wenn Sie eine statische Methode aufrufen wollen.

Der Weg zu den von Borland Pascal gewohnten Objekten ist jedoch nicht weit: Benutzen Sie statt des Schlüsselworts *class* wie gewohnt *object*, so erzeugt der Compiler Objekte nach dem alten Muster.

2.4 Typen

Auf dem Gebiet der Variablenverwaltung führt Object Pascal gegenüber dem Ur-Pascal keine neuen Vereinfachungen oder gar visuelle Hilfsmittel ein, d.h. dass Sie lokale Variablen nach wie vor selbst deklarieren müssen, bevor Sie sie benutzen (siehe Kapitel 2.1.4). Die einfachen Variablentypen von Object Pascal entsprechen grundsätzlich den verschiedenen Wertetypen für Properties des Objektinspektors, weisen aber weitere Variationsmöglichkeiten auf.

Neuigkeiten bei den Typen

Die ständige Erweiterung des Typvorrats von Delphi hat sich seit der Umstellung auf das 32-Bit-System in Delphi 2 etwas verlangsamt. Die folgende Tabelle zeigt die Neuerungen der letzten Delphi-Versionen, wobei in Delphi 5 und 6 keine neuen Typen einführt wurden:

Delphi 2	**Delphi 3**	**Delphi 4**
AnsiChar	WideString	Int64
WideChar	ByteBool (geändert)	LongWord
Currency	WordBool (geändert)	Real (geändert)
Variant	LongBool (geändert)	
String (geändert)		
SmallInt		
Cardinal (vergrößert)		
Integer (vergrößert)		
WideString (»Anfänge«)		

All diese Typen werden in den folgenden Abschnitten behandelt, bis auf den Typ *Variant*, der eher ein Spezialtyp für die COM-Automation und die Datenbankprogrammierung ist und uns daher erst in den Kapiteln 7.4.1 und 8.6.1 begegnen wird.

2.4.1 Einfache Typen

Zu den einfachen Typen gehören die Fließkommatypen und die Typen, deren Werte als ganze Zahlen darstellbar sind, die so genannten *ordinalen* Typen. Die wichtigste Gruppe der ordinalen Typen sind die Integertypen.

Integertypen

Die verschiedenen Integertypen unterscheiden sich nach Speicherbedarf und durch die Tatsache, ob ein Bit für das Vorzeichen verwendet wird oder ob nur positive Zahlen gespeichert werden können:

Typ	Wertebereich	Speicherbedarf	versionsspezifisch
ShortInt	–128..127	1 Byte	
Byte	0..255	1 Byte	
SmallInt	–32768-32767	2 Bytes	ab Delphi 2
Word	0..65535	2 Bytes	
LongWord	0..42949672950	4 Bytes	ab Delphi 4
LongInt	–2147483648..2147483647	4 Bytes	
Int64	$-2^{63}..2^{63}-1$	8 Bytes	ab Delphi 4
generische Typen:			
Integer	–2147483648..2147483647	4 Bytes	unter Delphi 1 nur 2 Bytes
Cardinal	0..42949672950	4 Bytes	unter Delphi 1 nur 2 Bytes, unter Delphi 2 und 3 ist der maximale Wert 2147483647

Da die Typen *Cardinal* und *Integer* keine fest definierte Größe haben, sondern sich gegebenenfalls an die Prozessor-Architektur anpassen (beim Schritt von 16- zu 32-Bit-Code wurde die Größe von *Integer* beispielsweise verdoppelt), werden sie auch als *generische* Typen bezeichnet, im Gegensatz zu den Typen, die sich auf neuen Prozessor-Architekturen garantiert nicht ändern, den *fundamentalen* Datentypen. (Auf zukünftigen 64-Bit-Plattformen werden die *Integers* sich übrigens nicht weiter verbreitern; wer 64-bittige Integers haben möchte, wird dann wohl auf den heute schon existierenden Typ *Int64* zurückgreifen müssen.)

Int64 und LongWord in der Praxis

Die Typen *LongWord* und *Int64* (eingeführt in Delphi 4) lassen sich im Prinzip genau wie die alten Typen verwenden. Da sich *LongWord* zu *LongInt* genauso verhält wie *Word* zu *SmallInt*, werfen wir hier nur noch einen kurzen Blick auf die Besonderheiten des Typs *Int64*. Die gute Nachricht ist zunächst einmal, dass der Compiler seit der Ein-

führung von *Int64* auch direkt angegebene Konstanten dieses Typs, also etwa ganze Zahlen mit 18 Nullen akzeptiert, beispielsweise so:

```
ShowMessage('Hello Int64: ' + IntToStr(1000000000000000000));
```

Ein weiteres Beispiel zeigt allerdings, dass Sie beim Typ *Int64* eventuell beachten müssen, dass der Standard-Integer von Delphi nach wie vor ein 32-Bit-Integer ist. Der folgende Aufruf, den Sie in eine beliebige *OnClick*-Methode einfügen können, erzeugt nämlich einen Überlauf beim Übersetzen des Programms:

```
ShowMessage('Int64-Test ergab: ' + IntToStr(1000000000 * 1000000000));
```

Dies liegt daran, dass Delphi jede der beiden Zahlen als 32-Bit-Integer interpretiert, da sie sich noch als solche darstellen lassen. Damit erhält aber auch das Ergebnis der Multiplikation vom Compiler nur noch einen 32-Bit-Integer zur Verfügung gestellt, wo ein 64-Bit-Integer notwendig wäre. Um das Problem zu beheben, müssen Sie die beiden Zahlen vor der Multiplikation manuell in *Int64*-Werte umwandeln:

```
ShowMessage('Int64-Test ergab: ' +
            IntToStr(Int64(1000000000) * Int64(1000000000)));
```

Hinweis: Für die Umrechnung von Strings in *Int64*-Werte gibt es die Funktion *StrToInt64*. Bei den Funktionen der Laufzeitbibliothek (und allgemein bei allen Funktionen und Methoden), die nur 32-Bit-Werte akzeptieren, werden 64-Bit-Integer-Werte auf 32 Bit gekürzt, wenn sie an diese Funktionen übergeben werden. Zu den Funktionen, die auch mit *Int64*-Werten umgehen können –, wie z.B. die oben verwendete *IntToStr* – siehe die Online-Hilfe unter dem Stichwort »Int64« .

Currency

Der Typ *Currency* (engl.: Währung) ist mit einem maximal darstellbaren Wert von 922.337.203.685.477,5807 (im negativen Zahlenbereich dehnt er sich sogar noch um 0,0001 weiter aus) auch für ausgefallene geldbezogene Berechnungen geeignet. Trotz der vorhandenen Nachkommastellen handelt es sich nicht um einen Fließkommatypen, denn der Kommabereich ist auf vier Stellen fixiert.

Vielmehr ist *Currency* bei Vernachlässigung des Kommas ein 64-Bit-Integer-Wert, der die Zehntausendstel einer (Geld-)Einheit angibt. Berechnungen, die nur *Currency*-Werte enthalten, lassen sich so vom Prozessor als schnelle Integer-Operation ausführen (wenn auch etwas langsamer als die »natürlichen« 32-Bit-Operationen).

Der Typ *Currency* reiht sich artig in die Reihe der anderen Typen ein, lässt sich also mit denselben Operatoren verarbeiten und wird bei Bedarf in Fließkommawerte umgewandelt (wobei ein Genauigkeitsverlust möglich ist), so dass Sie ihn wie eine Fließ-

kommazahl behandeln und wie eine solche auch mit Integer-Typen kombinieren können (indem Sie den Integertyp in eine Fließkommazahl umwandeln oder den *Currency*-Typ in einen Integer, z.B. mit den Funktionen *Trunc* oder *Round*).

Fließkommatypen

Die Fließkommatypen von Delphi entsprechen den Fließkommaformaten der Intel-Prozessoren. Der Standard-Fließkommatyp ist *Double*, die Alternativen zu ihm verbrauchen mit 4 Bytes entweder weniger Speicher (*Single*) oder weisen mit ihren 10 Bytes eine höhere Genauigkeit und einen größeren Zahlenbereich auf (*Extended*). Der Typ *Comp* macht eine Ausnahme und bietet bei relativ kleinem Zahlenbereich die Genauigkeit des Typs *Extended*, verbraucht aber nicht so viel Speicher wie dieser.

Typ	Bereich	Genauigkeit	Speicherbedarf
Single	1.5*10^-45..3.4*10^38	7–8 Stellen	4 Bytes
Double	5*10^-324..1.7*10^308	15–16 Stellen	8 Bytes
Extended	3.4*10^-4932..1.1*10^4932	19–20 Stellen	10 Bytes
Comp	–9.2*10^18..9.2*10^18	19–20 Stellen	8 Bytes
Real	wie *double*		

Bis Delphi 3 hatte der Typ *real* noch sein aus alten Zeiten geerbtes eigenes 6-Byte-Format und daher natürlich einen kleineren Wertebereich. Da dieses Format aber von den Fließkommaeinheiten der aktuellen Prozessoren nicht unterstützt wird, war es trotz der handlicheren Ausmaße der Daten langsamer, weshalb Delphi seit der Version 4 per Voreinstellung *Real*-Variablen wie ein *Double* behandelt. Über die Compilerdirektive {$RealCompatibility ON} bringen Sie auch die neueste Delphi-Version noch dazu, die 6-Byte-Version von Real zu verwenden.

Zeichen: AnsiChar, WideChar und Char

Der Typ eines Zeichens heißt in Pascal *Char*. In einer *Char*-Variablen können alle Zeichen des ANSI-Zeichensatzes (bzw. ASCII, falls z.B. alte DOS-Dateien gelesen werden) gespeichert werden, *Char*-Konstanten werden in Hochkommata eingeschlossen. Zeichen, die sich nicht auf der Tastatur befinden, können Sie mit ihrem Zahlenwert (z.B. *#10*) oder als Steuerzeichen (z.B. ^*A*) angeben (siehe Übersicht in Kapitel 2.1.2).

Delphi unterstützt auch Zeichen des Unicode-Zeichensatzes, die zwei Bytes benötigen. Der Typenbezeichner hierfür lautet *WideChar*. Da die ersten 256 Zeichen des Unicode-Zeichensatzes mit denen des ANSI-Zeichensatzes übereinstimmen, können Sie einer *WideChar*-Variablen ohne Probleme einen *Char*-Wert zuweisen.

Technisch gesehen sind *Char* und *WideChar* die einzigen Zeichentypen von Delphi 6. Allerdings sollten Sie sich auch mit dem Typ *AnsiChar* befassen. Dieser Typ bezeichnet ebenfalls nichts weiter als ein Zeichen nach dem ANSI-Zeichensatz, wie es auch in *Char* enthalten ist. Allerdings verhält es sich mit dem Typ *Char* so wie mit dem Typ *Integer* (es handelt sich also um einen generischen Typen): Er kann sich bei der Weiterentwicklung von Delphi ändern, was spätestens dann denkbar wäre, wenn Delphi auf einem Betriebssystem läuft, das nur noch mit dem Unicode-Zeichensatz arbeitet. Um auf jeden Fall mit einem nur ein Byte beanspruchenden ANSI-Zeichen zu arbeiten, sollten Sie daher den (fundamentalen) Typ *AnsiChar* verwenden. Die Größe des Typs *Char* ist für die Zukunft nicht festgelegt.

Aufzählungstypen

Pascal kennt zwei Arten von einfachen Typen, die nicht durch ein Schlüsselwort, sondern durch eine von Ihnen gegebene Definition festgelegt werden: *Teilbereichstypen* und *Aufzählungstypen*.

Aufzählungstypen kommen bei den Properties der Standardkomponenten sehr häufig vor. Die *WindowState*-Eigenschaft der Klasse *TForm* hat beispielsweise den folgenden Aufzählungstyp:

```
type
  TWindowState: (wsNormal, wsMinimized, wsMaximized);
```

Ein Aufzählungstyp besteht also aus einer Auswahl von benannten Zuständen, die Sie der Variable oder dem Property zuweisen können. Der Compiler verwaltet die in Frage kommenden Werte intern als Konstanten mit per Voreinstellung aufsteigenden Werten: Im obigen Beispiel erhält *wsNormal* den Wert 0, *wsMaximized* den Wert 2.

Die Pascal-Compiler von Borland ab dem Baujahr 2001 (Delphi 6, Kylix) erlauben Ihnen zusätzlich, die intern für die Aufzählungswerte verwendeten Konstanten selbst festzulegen. Dies ist unter anderem wichtig, um Aufzählungstypen von C, deren einzelne Werte von der voreingestellten Nummerierung abweichen, direkt nach Object Pascal übersetzen zu können, beispielsweise wie folgt:

```
type
  TBits = (bErstesBit = $01, bZweitesBit = $02,
           bDrittesBit = $04, bViertesBit = $08);
```

Typisch für die VCL-Aufzählungstypen ist übrigens die Voranstellung eines Kürzels, wobei z.B. *ws* für *WindowState* steht. Auf diese Weise werden Konflikte mit anderen Aufzählungstypen, bei denen auch ein Zustand wie *Normal* vorkommen könnte, vermieden.

Sie können mit Aufzählungstypen auch rechnen bzw. alle Werte eines Aufzählungstyps in einer Schleife durchlaufen, wenn Sie sich das ordinale Wesen des Aufzählungstyps zunutze machen (siehe ordinale Typen, Seite 239).

Teilbereichstypen

Durch die Definition eines Teilbereichstyps teilen Sie dem Compiler mit, dass nicht der gesamte Wertebereich eines Bytes, Words oder sonstigen Integers für diese Variable erlaubt ist:

```
type
  ArrayIndex = 0..55;
  Temperatur = -200..10000;
```

Variablen des obigen Typs *ArrayIndex* würden vom Compiler wie ein Byte, die des Typs *Temperatur* wie eine Integer-Variable behandelt werden, da diese den von Ihnen gewünschten Wertebereich am platzsparendsten umfassen. Zusätzlich besteht der Compiler darauf, dass Sie diesen Variablen nur Werte zuweisen, die in den angegebenen Bereich fallen. Weisen Sie einer Variablen statt einer Konstanten einen Ausdruck mit Variablen zu, kann der Compiler natürlich nicht vorhersehen, ob das Ergebnis zur Laufzeit in den geforderten Bereich fällt. Daher bietet er Ihnen an, mit der Option *Bereichsüberprüfung* bzw. dem Compilerschalter *$R+* Code zu generieren, der die Bereichsgrenzen auch zur Programmlaufzeit überprüft und bei Überschreitung eine *ERangeError*-Exception erzeugt. Die Bereichsüberprüfung kann auch bei der Indizierung von Arrays und Strings nützlich sein. Sie ist allerdings nur für Programme zu empfehlen, die noch in der Entwicklung sind, denn sie kostet viel (Lauf-)Zeit und vergrößert den Umfang des Programms.

Boolesche Typen

Als einen vordefinierten Aufzählungstyp, für den nur zwei Werte in Frage kommen, können Sie den Typ *Boolean* ansehen. Seine beiden Werte stehen für wahr (*True*) und falsch (*False*) und entstehen bei der Auswertung von booleschen Ausdrücken wie ($x < MaxX$) and ($y < MaxY$). Für eine *Boolean*-Variable reserviert der Compiler ein Byte Speicherplatz.

Intern wird *True* als 1, *False* als 0 gespeichert. Die interne Repräsentation von *ByteBool*, *WordBool* und *LongBool* wurde unter Delphi 3 allerdings an die entsprechenden Visual-Basic-Typen angepasst, *True* wird darin als -1 gespeichert. Dadurch sind *Byte-*, *Word-* und *LongBool* auf der einen und *Boolean* auf der anderen Seite nicht mehr völlig miteinander kompatibel.

Ordinale Typen

Für alle einfachen Typen außer den Fließkommatypen und den »Riesen-Integer« *Int64* gibt es eine eigene Bezeichnung: die der *ordinalen Typen*. Ihre wichtigste Gemeinsamkeit ist, dass sie alle auf leicht einsehbare Weise in ganze Zahlen umgewandelt werden können, falls sie das nicht schon sind. Mit der Funktion *ord* erhalten Sie zu einer ordinalen Variablen diesen zugehörigen Zahlenwert. Da jede ordinale Variable intern als eine solche Zahl gespeichert wird, muss die Funktion *ord* überhaupt nichts tun und kommt einer Typenumwandlung gleich. Hinter den verschiedenen Ordinaltypen verbergen sich die folgenden Zahlen:

- Die Ordinalzahl eines Zeichens ist sein ANSI-Code bzw. sein Unicode (für *WideChars*),
- die einer booleschen Variablen ist 1 für den Wert *True*, ansonsten 0,
- und die Konstanten eines Aufzählungstyps werden aufsteigend und von 0 beginnend durchnummeriert.

Natürlich können Sie auch den umgekehrten Weg einschlagen und einen Zahlenwert in einen anderen ordinalen Typ umwandeln, mit dem Ausdruck *Boolean(1)* erhalten Sie also wieder den Wert *True*.

High and Low R138

Zwei weitere wichtige Funktionen für ordinale Typen sind *High* und *Low*. Mit *High(Typenbezeichner)* oder *High(Variablenbezeichner)* erhalten Sie den höchsten Wert, der im Wertebereich des Typs bzw. der Variablen liegt. *High(Boolean)* ergibt also den Wert *True*, *High* angewendet auf *TWindowState* fördert *wsMaximized* zu Tage. Den kleinsten Wert erhalten Sie entsprechend mit der Funktion *Low*.

Mit diesen beiden Funktionen können Sie leicht alle Werte eines Aufzählungstyps durchlaufen:

```
type
  TWerte = (Null, Eins, Zwei);
   { wenn dies z.B. geändert wird auf:
     "TWerte = (Null, Eins, Zwei, Drei, Vier);"
      ... muss der folgende Code nicht geändert werden: }
var
  Zaehler: TWerte;
begin
  for Zaehler := low(TWerte) to high(TWerte) do
    ... { Zaehler nimmt hier die Werte 'Null', 'Eins', und 'Zwei' an. }
```

2.4.2 Operatoren und Ausdrücke

Die arithmetischen Operatoren für die vier Grundrechenarten und die Operatoren für Vergleiche von Zahlen sehen in Pascal so aus, wie man es erwartet, lediglich die Operatoren für ungleich (<>), kleiner gleich (<=) und größer gleich (>=) müssen aufgrund fehlender anderer Symbole auf der Tastatur aus zwei Zeichen zusammengesetzt werden.

Die meisten anderen Operatoren sind Schlüsselwörter, die hinsichtlich ihrer Bedeutung leichter zu merken sind als Zeichenkombinationen.

Die Rangfolge der Operatoren

Nach der Rangfolge sind die Operatoren in vier Prioritätsstufen unterteilt; folgen mehrere Verknüpfungen gleicher Priorität, werden sie von links nach rechts ausgewertet. Für eine davon abweichende Verknüpfungsreihenfolge müssen Sie Klammerungen vornehmen:

Priorität	Operatoren	Beschreibung
1	()	Klammern (werden in Pascal nicht als Operatoren bezeichnet)
2	@, not	unäre Operatoren
3	*, /, div, mod, and, shl, shr	multiplizierende Operatoren
4	+, -. or, xor	addierende Operatoren
5	=, <>, <, >, <=, >=, in	relationale Operatoren

Spezielle Integer-Operatoren

Eine Besonderheit hat sich der Pascal-Erfinder Wirth bei den Divisionen von Integern einfallen lassen: Hierfür dürfen Sie nicht das übliche Symbol »/« verwenden, sondern müssen zu *div* greifen. Den Rest einer Division ermitteln Sie mit dem Operator *mod* (z.B. *7 mod 3 = 1*).

Der Adressoperator

Ebenfalls etwas aus der Reihe fällt der Adressoperator »@« . Er ist der einzige Operator, der nur mit Variablen anzuwenden ist und bei dem der Typ egal ist. Sein Ergebnis ist die Adresse der angegebenen Variablen bzw. ein Zeiger darauf (zu Zeigern siehe Kapitel 2.4.6).

Bitweise Operatoren

Für die Verknüpfung von Zahlen auf Bitebene verwenden Sie Operatoren, deren Name bereits die Art der Operation angibt:

Operator	**Verknüpfung**	**Beispiel**
not	bitweises Negieren	not $F0 = $0F
and	bitweises Und	$0F or $11 = $01
or	bitweises Oder	$40 or $04 = $44
xor	bitweises Exklusiv-Oder	$A8 xor $A0 = $08
shl	alle Bits nach links schieben	$01 shl 4 = $10
shr	alle Bits nach rechts schieben	$20 shr 2 = $08

Boolesche Operatoren

Die Operatoren *not*, *and*, *or* und *xor* können auch zur Verknüpfung von booleschen Werten verwendet werden und bedeuten auch dort so viel wie *nicht*, *und* bzw. *oder*; *xor* bedeutet so viel wie »genau eines von beiden« . So entsteht aus zwei booleschen Werten ein neuer boolescher Wert: Der Ausdruck

```
Button1.Checked and Button2.Checked
```

prüft z.B., ob zwei Markierungsschalter angewählt sind, und kann in einer der Anweisungen zur Programmablaufsteuerung verwendet oder einer booleschen Variablen zugewiesen werden.

Auswertung von booleschen Ausdrücken

Die Auswertung von booleschen Ausdrücken kann auf verschiedene Arten erfolgen. Wenn Funktionen am Ausdruck beteiligt sind – und in den folgenden Beispielen soll jeder Bezeichner für eine Funktion stehen –, kann das große Auswirkungen auf die Programmgeschwindigkeit haben ...

```
if True or RechneEineHalbeStunde then
  ...
```

... oder auf die Programmlogik:

```
if DisketteLeer and BenutzerErlaubt and DisketteFormatieren
   then Meldung('Diskette erfolgreich formatiert.');
```

Wenn der erste Operand in einer Oder-Verknüpfung wahr ist, wie im ersten Beispiel, steht bereits fest, dass der gesamte Ausdruck wahr ist, auch wenn der zweite Teil *False* ergibt. Sind im zweiten Teil intensive Berechnungen erforderlich, ist es sehr vorteilhaft, wenn das Programm die Auswertung des Ausdrucks so früh wie möglich abbricht und bei vorzeitigem *True*-Ergebnis den *then*-Teil sofort ausführt.

Auch im zweiten Beispiel hätte eine vollständige Auswertung des Ausdrucks gravierende Folgen: Selbst wenn der Benutzer in letzter Sekunde das Formatieren der Diskette verhindern wollte und die Funktion *BenutzerErlaubt* den Wert *False* zurückliefert, würde das Programm versuchen, den letzten Teil (*DisketteFormatieren*) auszuwerten, indem es die Funktion aufruft, die die Diskette formatiert.

Welche Auswertungsmethode der Compiler verwendet, hängt von der Einstellung der Option BOOLESCHE AUSDRÜCKE VOLLSTÄNDIG im Optionsdialog bzw. der Compileranweisung *$B* ab; per Voreinstellung bricht der Compiler die Auswertung so früh wie möglich ab, tut also in beiden obigen Beispielen das Sinnvolle.

Manchmal sind auch vollständige Auswertungen sinnvoll – der folgende Ausschnitt wird wegen der *$B*+-Option in jedem Fall mit vollständiger Auswertung übersetzt:

```
{$B+}
   GesamtTestOK := CDROMansprechbar and FestplatteOK;
```

In diesem Beispiel sollen CD-ROM und Festplatte getestet werden. Wenn sich beim Testen des CD-ROM-Laufwerks und seines Treibers ein Fehler ergibt, kann der Gesamttest zwar nicht mehr erfolgreich ausgehen, der Benutzer möchte aber trotzdem erfahren, ob denn wenigstens die Festplatte in Ordnung ist.

Relationale Operatoren

Die Vergleichsoperatoren haben ebenfalls boolesche Werte zum Ergebnis. Bei ihnen ist zu beachten, dass sie niedriger priorisiert sind als die oben beschriebenen booleschen Operatoren. Im folgenden Fall benötigen Sie also eine Klammerung, sonst verknüpft der Compiler zuerst *MaxX* und *Y*, um dann einen Fehler wegen des »>«-Operators zu melden:

```
if (X > MaxX) or (Y > MaxY) then
   raise EIndexOutOfRange.Create('X oder Y'+
               ' außerhalb des zulässigen Bereichs');
```

Operatoren und andere Typen

Einige der bisher besprochenen Operatoren sind gewissermaßen polymorph wie eine virtuelle Funktion. Sie verursachen bei verschiedenen Typen verschiedene Operationen. Andere Typen außer den ordinalen und den Fließkommatypen, deren Variablen ebenfalls mit Operatoren verknüpft werden können, sind Zeiger, Mengen und Strings. Mehr dazu lesen Sie im entsprechenden Abschnitt.

2.4.3 Arrays

Arrays sind Tabellen mit einer oder mehreren Dimensionen, in denen Variablen, die alle denselben Typ haben, gespeichert werden. Object Pascal kennt auch indizierte Pro-

perties, die für den Nutzer wie ein Array aussehen. Einzelne Elemente von Arrays und indizierten Properties werden auf dieselbe Weise angesprochen: Hinter dem Namen des Arrays/Properties geben Sie die Indizes in eckigen Klammern an. Die Indizes müssen nicht unbedingt Zahlen sein, Arrays können alle ordinalen Typen als Indizes haben, bei Properties sind sogar alle Typen erlaubt. Ein Beispiel für Arrays mit ordinalen Dimensionen:

```
GruenAnteil := Farbe[Gruen];
SpreadSheet[5, 6] := '4*b';
ObjektImRaum := Raum[x,y,z];
```

Wie indizierte Properties deklariert werden, lesen Sie in Kapitel 2.2.5; an dieser Stelle sei angenommen, dass es sich im obigen Beispiel um normale Arrays handelt. Diese könnten dann mit Hilfe der Schlüsselwörter *array* und *of* wie folgt deklariert worden sein:

```
type
  (*  zwei ordinale Typen  *)
  Farbanteile = (rot, gruen, blau);
  TabellenIndizes = 1..100;
var
  Farbe: array[Farbanteile] of Integer;
  SpreadSheet: array[TabellenIndizes, TabellenIndizes] of Integer;
  Raum: array[0..10, 0..10, 0..10] of TRaumObjekt;
```

Es ist also möglich, das erste Element eines Arrays mit dem Index 0, 1 oder jeder anderen Zahl im zulässigen Wertebereich zu belegen.

Dynamische Arrays

Object Pascal bietet eine besondere Unterstützung für *dynamische Arrays*, d.h. Arrays, deren Speicher erst zur Laufzeit reserviert wird (verfügbar ab Delphi 4). Im Gegensatz zu »handgemachten« dynamischen Arrays (mit Hilfe von Zeigern und Routinen zur dynamischen Speicherverwaltung) können Sie auf ein dynamisches Array im Sinne von Object Pascal wie auf ein statisches Array zugreifen, also ohne einen Zeiger dereferenzieren zu müssen. Darüber hinaus können Sie durch einfache Anweisungen eine Größenveränderung des Arrays veranlassen. Schließlich krönt Borland seine dynamischen Arrays noch mit einer Referenzzählung, doch dazu später mehr.

Ein dynamisches Array wird wie folgt deklariert:

```
var
  intArray1: array of Integer;
```

Dies mag so aussehen wie ein offenes Array, der Unterschied ist aber, dass es sich bei obiger Deklaration nicht um einen Funktionsparameter handelt. Wie bei dynamischen Objekten in Object Pascal üblich (siehe Konstruktor-Aufruf von Objekten), müssen Sie auch bei dynamischen Arrays explizit veranlassen, dass Speicher reserviert wird:

```
begin
  intArray1[1] := 100; // << Fehler, da noch nicht reserviert!
  SetLength(intArray1, 100);
  intArray1[100] := 100; // << Fehler, da Indizes von 0 bis 99
  intArray1[0] := 100; // Beschreiben des ersten Elements
```

Das Erfreuliche ist nun, dass Sie die Funktion *SetLength* auch weitere Male aufrufen können, wenn Sie die Größe des Arrays ändern wollen. Der bisherige Array-Inhalt wird dadurch nur dann »beschädigt«, wenn Sie das Array verkleinern.

Speicherverwaltung der dynamischen Arrays

Wie schon die langen Strings und Objekte mit Interfaces verfügen auch die dynamischen Arrays über eine automatische Referenzzählung, die es der Delphi-Laufzeitbibliothek erlaubt, automatisch für die Freigabe des Speichers zu sorgen, sobald das Array (bzw. der String bzw. das COM-Objekt) nicht mehr benötigt wird. Im einfachsten Fall ist das, wenn das Array in einer lokalen Variable erzeugt wird und die lokale Variable ungültig wird, weil die Ausführung der Funktion, in der sie deklariert wurde, beendet ist. Wenn Sie ein Array schon früher freigeben wollen, als dies automatisch geschehen würde, können Sie dies durch folgenden Aufruf tun:

```
Finalize(intArray1);
```

Verwaltung mehrerer Referenzen

Der Referenzzähler eines dynamischen Arrays gibt an, von wie vielen Variablen ein Array referenziert wird. Im Schritt (*2*) des folgenden Beispiels wird beispielsweise erreicht, dass das im ersten Schritt erzeugte Array gleichzeitig von zwei verschiedenen Variablen benutzt wird:

```
type
  TIntArray = array of LongInt; // Diese Typdeklaration ist für die
                                // Zuweisung in Zeile (*4*) erforderlich.
var
  GlobaleInts : TIntArray;

procedure LokalerGueltigkeitsbereich;
var
  intarray1, intarray2: TIntArray;
begin
(*1*)  SetLength(intarray1, 1000000); (* Speicherverbrauch: 4 MByte *)
(*2*)  intarray2 := intarray1;
(*3*)  intarray2[50] := 100;
(*4*)  GlobaleInts := intarray2;
(*5*) end;
```

Wichtig ist, was in Zeile 3 passiert: Die Zuweisung an die fünfzigste Zahl des zweiten Arrays verändert gleichzeitig auch *intarray1[50]*, da beide Array-Variablen auf das-

selbe Array verweisen, dessen Referenzzähler sich nun auf 2 erhöht hat. Probierten wir Ähnliches mit langen Strings, würden wir zwei verschiedene Strings mit einem Referenzzähler von 1 erhalten (zu *Copy On Write* siehe Kapitel 2.4.4, *Interner Aufbau der langen Strings*).

Zeile 4 bewirkt schließlich, dass der Referenzzähler auf 3 steigt, da nun auch die globale Variable *GlobaleInts* auf das Array verweist. Am Ende der Prozedur in Zeile 5 werden nur die beiden lokalen Variablen ungültig, der Referenzzähler des dynamischen Arrays wird um zwei kleiner, sinkt also nicht auf Null, so dass das Array auch noch nicht automatisch freigegeben wird.

Hinweis: Falls Sie sich von der Referenzzählung in der Praxis überzeugen wollen, so können Sie das im Falle der COM-Objekte anhand des Beispielprogramms aus Kapitel 8.5.5 tun.

Mehrdimensionale Arrays

Alles bisher Gesagte lässt sich direkt auf mehrdimensionale Arrays übertragen, wenn Sie nur wissen, wie ein mehrdimensionales Array deklariert wird ...

```
var
  IntMatrix: array of array of Int64;
```

... und dass die Prozedur *SetLength* mehrere Parameter haben kann:

```
SetLength(IntMatrix, 100, 100); // reserviert 100*100 Int64-Zahlen
```

Stellt man sich *array of Int64* als geklammert vor, so zeigt die Deklaration von *IntMatrix* bereits, dass sich *IntMatrix* auch als eindimensionales Array ansehen lässt, und es ist tatsächlich erlaubt, es auch so im Speicher anzulegen:

```
SetLength(IntMatrix, 100); // reserviert 100 array of Int64-Variablen
```

Für die 100 Elemente des Arrays gilt dasselbe wie für normale Variablen des Typs *array of Int64*: Jedes Element ist ein dynamisches Array, das einzeln mit *SetLength* oder durch Zuweisung initialisiert werden muss. Und da dynamische Arrays verschieden groß sein können, können Sie jedem Element ein Array einer anderen Größe zuweisen, so dass das gesamte Array *IntMatrix* nicht mehr rechteckig ist, sondern beispielsweise dreieckig oder völlig unregelmäßig gestaltet. Die Speicherverwaltung der Delphi-Laufzeitbibliothek jedenfalls lässt sich durch so etwas nicht verwirren.

2.4.4 Die verschiedenen Stringtypen

In Object Pascal gibt es drei unterschiedliche Stringtypen: *ShortString* (Strings mit internem Längenbyte, wie von Turbo Pascal bekannt), *String* (seit Delphi 2 wird für diesen Standard-Stringtyp ein neues Format verwendet) und *WideString* (ein String,

der aus *WideChars* besteht, der also mit Zeichensätzen umgehen kann, die mehr als 255 verschiedene Zeichen enthalten). Für die Interoperabilität mit C/C++-Code wichtig ist der Typ *PChar*, der ein Zeiger auf ein C-kompatibles Zeichen-Array darstellt. Er stellt in Object Pascal keinen eigenen String-Typ dar (das heißt, dass Sie *PChar*-Variablen nicht wie *String*-Variablen manipulieren können). Wenn Sie einen *PChar*-Zeiger brauchen, verwenden Sie in Object Pascal normalerweise einen *String* und gewinnen daraus durch Typumwandlung einen *PChar*, wenn erforderlich.

Der Typ *String* ist vergleichbar mit den generischen Typen *Integer* und *Char*, da sich sein interner Aufbau zwischen Delphi 1 und Delphi 2 verändert hat, seine Verwendung aber nahezu gleich geblieben ist. Für die Zukunft ist eine erneute Änderung absehbar, falls der UniCode-Zeichensatz einmal den ANSI-Zeichensatz als Standard ablösen sollte (*String* wird dann zu einem *WideString*). Wenn Sie einen Stringtyp suchen, dessen interner Aufbau gleich bleibt (vergleichbar mit den fundamentalen Integer-Datentypen), so bietet Ihnen Object Pascal diesen im Typ *AnsiString* (String aus *AnsiChar*-Zeichen, zurzeit noch mit den normalen Strings identisch).

- Grundsätzlich sind alle Strings, die Sie mit dem Bezeichner *String* deklarieren, unabhängig von der verwendeten Compiler-Version sehr einfach zu handhaben (z.B. können Sie zwei Strings mit dem Operator '+' zu einem dritten String zusammenfügen). Die Einführung des neuen String-Formats in Delphi 2 wurde aber nicht nur erforderlich, um die steinzeitliche Längenbegrenzung auf 255 Bytes aufzuheben, sondern auch, um eine höhere Interoperabilität mit C/C++-Funktionen zu erreichen: Um einen Object-Pascal-String an eine C/C++-Funktion wie etwa eine Windows-API-Funktion weiterzugeben, müssen Sie ihn lediglich in einen *PChar* umwandeln, z.B. *ApiFunktionsAufruf(PChar(LangerPascalString))*.

Die folgenden Abschnitte behandeln die drei verschiedenen Stringtypen im Einzelnen.

Alte Pascal-Strings mit Längenbyte

Intern bestehen die Pascal-Strings des alten Formats (die Standard-Strings von allen Turbo/Object-Pascal-Compilern bis Delphi 1) aus einem Zeichenarray, dessen Indizierung bei 0 beginnt. An Position 0 befindet sich ein Längenbyte, das angibt, wie viele Zeichen der String umfasst und wie viele Bytes des reservierten Speichers er somit verwendet. Die Menge des reservierten Speichers ist bei diesem Typ konstant. Wenn Sie also beispielsweise mit *string[50]* 51 Bytes (inklusive des Längenbytes) reservieren und der Variablen einen String zuweisen, der kürzer als 50 Zeichen ist, bleibt ein Teil des Speichers ungenutzt.

Die Deklaration von alten Pascal-Strings unterscheidet sich zwischen den Delphi-Versionen etwas. Die empfehlenswerte, weil eindeutige Deklarationsweise ist jene, die Länge des Strings in eckigen Klammern hinter dem Wort *String* anzugeben:

```
var
  StringFuer100Zeichen: string[100];
```

Diese Deklaration reserviert in allen Delphi-Versionen einen statischen String des alten Typs. Wenn Sie die Größenangabe in Klammern weglassen, erhalten Sie

- unter 16-Bit-Delphi einen alten String mit einer Kapazität von 255 Zeichen;
- unter 32-Bit-Delphi mit der standardmäßig eingestellten Option für lange Strings (*Huge Strings* in den Compileroptionen bzw. *{$H+}* im Quelltext) einen neuen dynamischen String mit bis zu 2 GB Kapazität;
- unter 32-Bit-Delphi mit abgeschalteten langen Strings (*{$H-}*) einen alten String mit 255 Zeichen Kapazität.

Normalerweise können Sie es jedoch immer bei der Einstellung *{$H+}* belassen, denn einen alten String mit 255 Zeichen erhalten Sie in jeder Delphi-Version mit *String[255]*. In der aktuellen Object-Pascal-Version gibt es außerdem die Typenbezeichner *ShortString* (für einen String im alten Format) und *AnsiString* (für einen langen String).

Die Verwendung dieses Stringtyps funktioniert so wie die Verwendung der langen Strings, sofern Sie nur die Routinen der Delphi-Laufzeitbibliothek auf diese Strings anwenden.

Die langen 32-Bit-Strings

Einen langen String erhalten Sie auf zwei verschiedene Arten:

- in der Voreinstellung *{$H+}* (*Huge Strings* in den Compileroptionen) einfach über den Typenbezeichner *String* oder über *AnsiString*;
- in der Einstellung *{$H-}* nur über den Typenbezeichner *AnsiString*.

Zeichenketten, die durch Hochkommas eingeschlossenen sind, sind mit dem neuen Stringtyp (und auch mit dem alten kurzen String) kompatibel. Sie können eine solche Zeichenkette also direkt einer Stringvariablen zuweisen und als Parameter für Funktionen verwenden, die einen String erwarten (sofern sie nicht zwingend eine String*variable* verlangen).

```
var
  s: string;
begin
  s := 'Zuweisung eines (kurzen bzw. langen) Strings';
  MessageDlg('Ausgabe eines Strings im Meldungsfenster',
             mtInformation, [mbOK], 0);
```

Um die Länge eines Strings in Erfahrung zu bringen, sollten Sie immer die Funktion *Length* verwenden:

```
Len := Length(s);  { funktioniert immer }
Len := Byte(s[0]); { Direkte Abfrage des Längenbytes: funktioniert
                     nur bei den alten, kurzen Strings }
```

Um Strings miteinander zu verbinden oder ihre alphabetische Reihenfolge zu vergleichen, verwenden Sie den '+'-Operator bzw. die relationalen Operatoren '<' und '>'. Mit '=' ist ein schneller Stringvergleich möglich. All diese Operatoren unterscheiden zwischen Groß- und Kleinschreibung. (Für einen Stringvergleich ohne Beachtung der Groß- und Kleinschreibung ist die Funktion *CompareStr* vorgesehen.)

Zur Manipulation der Strings stehen Ihnen verschiedene Funktionen zur Verfügung, zu denen Sie in Kapitel 2.8.4 eine Übersicht finden. Um beispielsweise von einem String, der einen Punkt enthält, den Teil rechts des Punktes in einen neuen String zu kopieren, können Sie den Punkt mit der Funktion *Pos* suchen und dann den rechten Stringteil mit *Copy* kopieren:

```
PunktPosition := Pos(Eingabe, '.');
if PunktPosition > 0 then
  Ausgabe := Copy(Eingabe, { String, von dem kopiert werden soll }
                  PunktPosition+1, { Erstes Zeichen zum Kopieren }
                  Length(Eingabe)-PunktPosition) { Zahl der kopierten Z. }
else Ausgabe := 'Die Eingabe enthält keinen Punkt!';
```

Stringmanipulation auf Zeichenbasis

Auf die einzelnen Zeichen eines Strings können Sie wie bei einem ganz normalen Array zugreifen:

```
ZehntesZeichen := s[10];
s[9] := ZehntesZeichen;
```

Und Sie können sogar Zeichen beschreiben, die hinter dem bisherigen String-Ende liegen, wenn Sie den String vorher explizit verlängern. Zu Beginn hat beispielsweise jeder mit *String* deklarierte lange String die Länge 0 (ein mit *String[255]* außerhalb einer Klasse deklarierter String des alten Formats hat im uninitialisierten Zustand eine nicht vorhersagbare Länge von maximal 255 Zeichen, da das Längenbyte einen zufälligen Wert enthält). Wenn Sie nun im langen String das fünfte Zeichen setzen wollen, können Sie das wie folgt tun:

```
var
  s: AnsiString;
begin
  // s ist jetzt 0 Zeichen lang
  SetLength(s, 5);
  // s ist jetzt 5 Zeichen lang
  s[5] := NeuesZeichen;
```

Die Funktion *SetLength* funktioniert sowohl mit kurzen als auch mit langen Strings. In Turbo Pascal und Delphi 1 hätten Sie die Länge eines Strings jedoch direkt mit *s[0] := Char(5)* setzen müssen. Wichtig ist bei den langen Strings, dass Sie die Länge *vor* dem Zugriff auf das Zeichen schon groß genug einstellen, da Sie sonst eine *EAccessViolation*-Exception erhalten.

Die Zeichen an den Positionen eins bis vier im obigen Beispiel haben übrigens nach Ausführung der beiden obigen Anweisungen noch einen zufälligen Inhalt.

Interner Aufbau der langen Strings

Variablen für lange Strings sind selbst immer nur vier Bytes groß und stellen einen Zeiger auf einen dynamisch verwalteten Speicherbereich dar, der den eigentlichen String-Inhalt enthält. Die Verwaltung dieses Speicherbereichs (dessen Größe bei einer Längenänderung des Strings angepasst wird) geschieht völlig automatisch, ebenso wie die Dereferenzierung des Zeigers.

Der dynamisch verwaltete Speicherblock des Strings ist wie folgt aufgebaut: Er beginnt mit zwei 32-Bit-Werten, von denen einer die Länge des Strings angibt und der zweite einen *Referenzzähler* enthält. Daraufhin folgt der eigentliche String-Inhalt, abgeschlossen durch ein Nullzeichen.

Der Referenzzähler ermöglicht eine große Einsparung von Speicher, denn er erlaubt, dass sich mehrere Stringvariablen einen String teilen. Er zählt, wie viele Stringvariablen ein und denselben dynamischen String-Datenblock verwenden. Im folgenden Programmausschnitt wird nur einmal Speicher für einen String reserviert. *A* und *B* verwenden danach denselben Speicherbereich, der Referenzzähler dieses Bereichs enthält danach den Wert 2:

```
var
  a, b: AnsiString;
begin
  a := 'Zuweisung eines langen Strings';
  b := a;
```

Diese doppelte Verwendung desselben Speicherbereichs hindert Sie jedoch nicht daran, einen der beiden Strings zu verändern, denn die Laufzeitbibliothek verfährt in diesem Fall nach der so genannten *Copy-On-Write*-Methode. Wird die folgende Zeile ausgeführt ...

```
b := b+' mit anschließender Verlängerung';
```

... kopiert die Laufzeitbibliothek den noch unveränderten (von *a* und *b* genutzten) String in einen neuen Speicherbereich, setzt danach die Referenzzähler der beiden jeweils nur noch einmal verwendeten Speicherbereiche auf eins und hängt an den Speicherblock, der jetzt nur noch für die Variable *b* vorgesehen ist, die oben angege-

bene Zeichenkette an. (Dieses Verhalten steht im Unterschied zu den dynamischen Arrays, bei denen es keinen *Copy-On-Write*-Mechanismus gibt, siehe Kapitel 2.4.3)

Das Nullzeichen nach dem eigentlichen String-Inhalt ist zwar nicht mehr nötig, um das String-Ende festzustellen, da es hierfür ja schon die Größenangabe am Anfang des Speicherblocks gibt; aber durch das Nullzeichen lässt sich der String einfach in einen nullterminierten String umwandeln, beispielsweise, um ihn an eine C-Funktion zu übergeben:

```
MessageBox(handle, PChar(TextImLangenString), PChar(Titel), 0);
```

Dieser Aufruf erhöht allerdings *nicht* den Referenzzähler der beiden umgewandelten Strings, daher bergen solche Umwandlungen die Gefahr in sich, dass ein String zu früh automatisch freigegeben wird. Und auch wenn Sie einen so umgewandelten String verändern wollen, ist Vorsicht geboten: Indem Sie einen *String* als *PChar* ansprechen, umgehen Sie ebenfalls die automatische Verwaltung des Referenzzählers. Wenn Sie Strings dennoch auf diese Weise verändern wollen, finden Sie in der Online-Hilfe zu den langen Strings weitere Informationen dazu (so gibt es z.B. eine spezielle Prozedur *UniqueString*).

Umgekehrt können Sie einen nullterminierten String in einen *String* umwandeln, indem Sie beispielsweise *String(NullTermString)* schreiben. Oft wird eine solche Umformung jedoch schon automatisch durchgeführt, z.B. wenn Sie einem *String* einen nullterminierten String einfach zuweisen.

Hinweis: Für einen langen String mit einer Länge von null Zeichen wird überhaupt kein dynamischer Speicher reserviert. Lediglich der Zeiger, aus dem die eigentliche Stringvariable besteht, nimmt weiterhin vier Bytes in Anspruch; er enthält dann den Wert *nil*. Da Sie die Länge eines langen Strings von der Funktion *Length* erfahren können, brauchen Sie sich auch um dieses interne Detail normalerweise nicht zu kümmern.

Nullterminierte C-Strings

Nullterminierte Strings waren zu 16-Bit-Zeiten die einzige Möglichkeit, Strings, die länger als 255 Zeichen sind, zu verwalten. Inzwischen sind sie als eigenständige Strings kaum mehr nötig, denn jeder String des aktuellen Formats *enthält* quasi einen nullterminierten String, ergänzt durch eine zusätzliche Längenangabe, die die Verwaltung des Strings vereinfacht, und weiterhin ergänzt durch ein vollkommen automatisches Speichermanagement.

Ein nullterminierter String besitzt kein Längenbyte, sondern definiert sein Ende durch ein Terminator-Nullzeichen (#0). Durch das Fehlen des Längenbytes gestalten sich einige Operationen einfacher. So können Sie beispielsweise einen Teilstring aus einem

nullterminierten String erhalten, indem Sie einfach einen Zeiger auf das Zeichen setzen, mit dem der Teilstring beginnen soll. Der Zeiger weist zwar nur auf einen Teilstring, dieser ist aber jederzeit als eigenständiger String verwendbar. Dies ist mit den Object-Pascal-Strings nicht möglich: Weder aus kurzen, noch aus langen Pascal-Strings lassen sich so einfach Teilstrings herausziehen, die wieder ein Pascal-String sind.

Natürlich gibt es auch Nachteile; vor allem ist die Frage, wie lang ein nullterminierter String ist, nur sehr zeitaufwändig zu beantworten. So muss die Funktion *StrLen* (das Gegenstück zur Pascal-String-Funktion *Length*) die Länge eines Strings ermitteln, indem sie alle Zeichen des Strings bis zum abschließenden Nullzeichen zählt. Außerdem wird die Verwendung von nullterminierten Strings weitaus weniger durch den Compiler unterstützt als die Verwendung von Pascal-Strings. So können Sie die nullterminierten Strings nicht mit Operatoren manipulieren, sondern müssen die Funktionen der Unit *SysUtils* (siehe auch Kapitel 2.8.4) verwenden. Sogar für die Zuweisung eines in Hochkommas angegebenen Strings an ein nullterminiertes Zeichenarray benötigen Sie einen Funktionsaufruf (*StrCopy*).

Doch zuerst zur Deklaration eines nullterminierten Strings. Sie deklarieren ihn wie ein normales Zeichenarray, dessen erster Index 0 ist:

```
var
  NullStr: array[0..100] of Char;
```

Nun können Sie die zahlreichen Funktionen der Unit *SysUtils* auf diesen String anwenden. Um beispielsweise einen Pascal-String in diesen zu kopieren, verwenden Sie die Funktion *StrPCopy*:

```
var
  Str: string;
begin
  StrPCopy(NullStr, Str);
```

PChar

Schneller geht es, wenn Sie Konstanten vom Typ *PChar* verwenden, der einen Zeiger auf einen nullterminierten String darstellt. Ihm können Sie Zeichenketten direkt zuweisen:

```
const
  StringZeiger: PChar = 'Dieser String wird nicht kopiert.';
```

Dies veranlasst den Compiler dazu, den angegebenen String als nullterminierten String zu speichern und den Zeiger *StringZeiger* auf dessen Anfangsadresse zu setzen. Eine *PChar*-Variable können Sie überall dort einsetzen, wo Sie auch einen nullterminierten String verwenden können. Zu bedenken ist aber, dass Sie die *PChar*-Variable vorher auf einen reservierten Speicherbereich setzen müssen.

Das folgende Beispiel demonstriert den oben schon erwähnten Vorteil der nullterminierten Strings und durchläuft einen String vom ersten Zeichen bis zum abschließenden Nullzeichen. Wichtig ist dabei, dass *Zeiger* zu jeder Zeit auf einen gültigen nullterminierten String weist, den Sie wie jeden anderen nullterminierten String verwenden können:

```
procedure DurchlaufeStr(Str: PChar);
var
  Zeiger: PChar;
begin
  Zeiger := Str;
  while Zeiger^ <> #0 do begin
    inc(Zeiger); { um eins erhöhen }
    ...
```

Sie können also mit einem *PChar*-Zeiger rechnen, indem Sie Zahlen hinzuaddieren (oder Funktionen wie *inc* und *dec* verwenden), wodurch sich seine Position um die entsprechende Zeichenzahl nach rechts oder links bewegt. Wenn zwei *PChar*-Variablen in den gleichen String zeigen, können Sie durch die Vergleichsoperatoren feststellen, welche der beiden Positionen vor der anderen liegt.

WideStrings

Ein Unicode-String ist ein String, dessen einzelne Zeichen jeweils zwei Bytes groß sind. Für diese Strings gibt es den Typ *WideChar*, der ein zwei Byte großes Zeichen speichert, und den Typ *PWideChar*, der dem Typ *PChar* für 1-Byte große Zeichen entspricht, mit dem Sie also ebenfalls »rechnen« können und den Sie ebenfalls als *array[0..x] of WideChar* anlegen können.

Passend zum *WideChar* gibt es seit Delphi 3 den Typ *WideString*, der wie ein *AnsiString* automatisch verwaltet wird, allerdings ohne die Vorteile einer Referenzzählung. Das folgende Beispiel zeigt, dass Sie mit *WideStrings* wie mit einem *AnsiString* umgehen können, dass Sie jedoch für die Konvertierung von *WideString* in *AnsiString* eine Hilfsfunktion aufrufen müssen (Gleiches gilt für den umgekehrten Weg):

```
var
  W1, W2: WideString;
  NormalString: String;
begin
  W1:='Test der langen und weiten Strings. ';
  W2:=W1+W1;
  NormalString:=WideCharToString(@w1[1]);
```

WideStrings werden bisher noch eher selten eingesetzt. Nur ein einziges Beispielprogramm dieses Buchs, der Shell-Explorer aus Kapitel 8.5.3, rechnet damit, dass es von den neuesten Windows-Versionen in einem Fall bereits Unicode-Strings zurückgeliefert bekommt, die es dann aber sofort mit *WideCharToString* in das herkömmliche Format umwandelt.

2.4.5 Strukturierte Typen

Die Records der herkömmlichen Pascal-Programmierung sind die Grundlage für die Objekttypen und können auch jetzt noch bei einfachen Strukturen, die keine Methoden benötigen, als reine Datentypen verwendet werden. Eine Record-Deklaration besteht aus dem Schlüsselwort *record*, einer Liste von Datenelementen und einem abschließenden *end*, dem noch ein Semikolon folgt:

```
type
  TDatei = record
    Name: string[12];
    Pfad: string[120];
    Groesse: LongInt;
  end;
```

Ein Record ist ein eigener Gültigkeitsbereich. Seine Elemente dürfen grundsätzlich alle als Bezeichner erlaubten Namen haben, auch wenn diese in einem äußeren Gültigkeitsbereich schon definiert sind, denn um die Elemente anzusprechen, müssen Sie sowieso zuerst den Namen des Records angeben, wodurch Verwechslungen ausgeschlossen sind:

```
var
  Name: string;
  Datei: TDatei;
begin
  Datei.Name := Name;
```

Varianten

Eine kompliziertere Form der Records sind die Records mit Abschnitten, die auf verschiedene Weise gedeutet werden können, den *Varianten*. In einer Syntax, die der *case*-Anweisung ähnelt, geben Sie hier verschiedene Möglichkeiten an, wie der mehrdeutige Bereich interpretiert werden kann. Der mehrdeutige Bereich muss sich immer am Schluss des Records befinden (weshalb er nicht durch ein eigenes *end* beendet wird) und kann z.B. so aussehen:

```
type
  VariantRecord = record
     VariableImmerDa: Integer;
     case GrosseTeile: Boolean of
       True: (a, b, c: LongInt);
```

```
        False: (d, e, f, g: Word;
                i: LongInt)
    end;
```

Semikolons sind hier erforderlich, um die einzelnen Fälle und die Deklarationen innerhalb der Fälle zu trennen.

Mengen

Wir kommen nun zu einem Typ, der in der VCL zahlreiche Verwendung findet: dem Mengentyp. Grundsätzlich gilt die folgende Syntax für die Deklaration eines Mengentyps und die Angabe von Mengen, die darin gespeichert sind:

```
var
  CharSet: set of Char;
begin
  CharSet := ['a'..'z'];
```

Der Basistyp einer Menge (der Typ, der hinter dem *of* angegeben wird) muss ein ordinaler Typ sein, der nicht mehr als 256 verschiedene Werte annehmen kann, denn für jeden möglichen Wert braucht eine Mengenvariable ein Bit, um zu speichern, ob der Wert in der Menge enthalten ist oder nicht. Das heißt, dass eine Zeichenmenge wie die obige mit 32 Byte so viel Platz wie 8 *LongInts* in Anspruch nimmt.

Eine Menge ist natürlich nur dann interessant, wenn schnell Elemente hinzugefügt oder herausgenommen werden können und wenn man überprüfen kann, ob ein Element enthalten ist oder nicht. Zwei Standardfunktionen vereinfachen das Hinzufügen und Wegnehmen einzelner Elemente:

```
{ Hinzufügen eines Elements: }
  Include(CharSet, 'd');
{ Löschen eines Elements: }
  Exclude(CharSet, 'x');
```

Diese Funktionen sind allerdings nicht auf Properties anwendbar, da ihre Parameter Variablenparameter sind. Ein Mengen-Property wird beispielsweise wie folgt durch ein neues Element erweitert:

```
BorderIcons := BorderIcons+[biMaximize];
```

Neben dem '+'-Operator können Sie noch viele weitere Operatoren auf Mengen anwenden:

Operator	Funktion	Ergebnis	Beispiel
+	Vereinigung	Menge	[1..3,5,9]+[4..6]=[1..6, 9]
-	Differenz	Menge	['c'..'h']-['a'..'z']=[]
*	Schnittmenge	Menge	[1,3,5,6]*[1,2,3]=[1,3]

Operator	Funktion	Ergebnis	Beispiel
=	gleich	Boolean	([blau]=[blau])=True
<>	ungleich	Boolean	
<=	Teilmenge	Boolean	([5]<=[1,5,10])=True
>=	Obermenge	Boolean	
<	echte Teilmenge	Boolean	([rot]<[rot])=False
>	echte Obermenge	Boolean	
in	Element enthalten	Boolean	(Gruen in [Rot, Gruen])=True

2.4.6 Zeigertypen

Die Bedeutung der Zeigertypen (Zeiger = *Pointer*) ist in Object Pascal zwar dadurch etwas zurückgegangen, dass der Compiler Objekte und Strings von sich aus dynamisch verwaltet und diesen Vorgang nahezu völlig transparent für den Entwickler gestaltet. In anderen Bereichen sind Zeiger jedoch weiterhin sehr hilfreich: Die Verwaltung von nullterminierten Strings und dynamischen Speicherblöcken, deren Größe sich womöglich zur Laufzeit noch ändern kann, ist mit Zeigern besonders vorteilhaft, ebenso wie der Aufbau von dynamischen Datenstrukturen wie Listen und Bäumen.

Während bei allen anderen Variablen Speicher für irgendwelche Daten reserviert wird, enthalten Zeigervariablen lediglich die Adresse eines solchen Speicherbereichs. Für diese werden bisher seit den Anfangstagen von Turbo Pascal 4 Bytes Speicher benötigt, bei zukünftigen 64-Bit-Prozessorarchitekturen kann dieser Bedarf aber auch auf 64 Bit anwachsen.

Bei der Deklaration einer Zeigervariablen wird zwischen typisierten und untypisierten Zeigern unterschieden. Der Inhalt beider ist völlig identisch, jedoch kann der Compiler den Umgang mit typisierten Zeigern besser überwachen. Einen typisierten Zeiger, der auf die Variable eines Basistyps zeigt, erhalten Sie mit der Schreibweise »^*BasisTyp*«. Meistens erhalten Zeigertypen eigene Bezeichner und werden erst dann einer Variablen zugewiesen (im folgenden Beispiel werden die Typen *PByteFeld* und *PInteger* auf diese Weise deklariert):

```
type
  TByteFeld = array[0..100] of Byte;
  TEinfachRecord = record
    a, b: Byte;
  end;
  PByteFeld = ^TByteFeld;
  PInteger = ^Integer;

var
  ZeigerAufInteger: PInteger;
```

```
  ByteFeldPtr: PByteFeld;
  RecordPtr1, RecordPtr2: ^TEinfachRecord;
```

Zuweisungen an Zeigervariablen

Da Zeigervariablen Adressen enthalten, können Sie ihnen auch nur Adressen zuweisen. Diese können Sie grob gesagt auf drei Arten erhalten:

- Sie nehmen die Adresse eines anderen Zeigers,
- reservieren neuen Speicher und verwenden die Adresse, die von einer Speicherreservierungs-Funktion wie *New* oder *GetMem* zurückgeliefert wird,
- oder Sie holen sich mit dem Adressoperator »@« oder der Funktion *Addr* die Adresse einer Variablen:

```
var
  ReserviertesByteFeld: TByteFeld;
  ReservierteZahl: Integer;
{ außerdem gelten die Bezeichner des letzten Beispiels }
begin
  RecordPtr1 := RecordPtr2;
  ByteFeldPtr := addr(ReserviertesByteFeld); { oder: }
  { Reservierung von 101 Bytes für ein TByteFeld }
  GetMem(ByteFeldPtr, sizeof(TByteFeld));

  ZeigerAufInteger := new(PInteger); { oder: }
  ZeigerAufInteger := addr(ReservierteZahl);
```

Das obige Beispiel setzt voraus, dass *RecordPtr2* vorher auf eine der letzten beiden Arten auf einen gültigen Speicherbereich gesetzt wurde, denn beim Programmstart zeigen Zeigervariablen (die nicht Teil eines automatisch initialisierten Objekts sind) in zufällige Bereiche, was zu schweren Fehlern führen kann, sobald Sie in diese Bereiche hineinschreiben.

nil-Pointer

Um Zeiger zu kennzeichnen, die auf kein gültiges Objekt zeigen, können Sie ihnen den Wert *nil* zuweisen. Ein Schreiben an die Adresse *nil* führt zwar immer noch zu einem Fehler, jedoch können Sie die Adresse *nil* vor dem Schreiben eindeutig als fehlerhaft erkennen. Andererseits können Sie davon ausgehen, dass Zeiger, die nicht *nil* sind, gültig sind, wenn Sie am Anfang des Programms alle ungültigen Zeiger mit *nil* initialisiert haben und jeden Zeiger wieder auf *nil* zurücksetzen, wenn Sie den ihm zugeordneten Speicher freigeben. Zur Überprüfung auf *nil* können Sie einen direkten Vergleich anstellen oder die Standardprozedur *Assigned* verwenden:

```
RecordPtr2 := nil;
if Assigned(RecordPtr2) then
```

```
  { ... Zugriff }
else { kein Zugriff }
```

Dereferenzierung

Sehen wir uns nun an, wie diese kritischen Zugriffe überhaupt erfolgen können. Um von der Adresse, auf die ein Zeiger zeigt, lesen zu können oder auf sie zu schreiben, müssen Sie den Zeiger *dereferenzieren*. Dafür ist wieder das Zeichen '^' zuständig, allerdings wird es diesmal hinter die Variable geschrieben. *RecordPtr2^* gibt also nicht mehr den Zeiger an, sondern den Speicherbereich, auf den er zeigt. Mit diesem können Sie so arbeiten wie mit einer normalen Variable des Zeiger-Basistyps. Wenn es sich um einen Record handelt, können Sie dessen Elemente einzeln wie im folgenden Beispiel ansprechen:

```
RecordPtr2^.a := 1;
ByteFeldPtr^[0] := 0;
ZeigerAufInteger^ := 1000;
GetMem(ByteFeldPtr, sizeof(ByteFeldPtr^));
```

Hinweis: Die Dereferenzierung ist auch beim Debuggen wichtig. Wenn Sie etwa im Datenbereich des CPU-Fensters den Speicherbereich sehen wollen, auf den ein Zeiger zeigt, müssen Sie unter ZU ADRESSE GEHEN den Ausdruck *ZeigerVar^* eingeben. Wenn Sie nur *ZeigerVar* eingeben, zeigt Delphi den Speicherbereich der eigentlichen Zeigervariablen, also den Bereich, in dem die Adresse liegt.

Untypisierte Zeiger

Wenn Sie Zeigeroperationen frei von den Zwängen des Compilers durchführen wollen, können Sie zu den untypisierten Zeigern greifen (meist ergibt sich deren Benutzung jedoch nicht aus Freiheitsstreben, sondern aus pragmatischen Gründen). Untypisierte Zeiger werden mit dem Wort *Pointer* deklariert:

```
var
  NatuerlicherZeiger: Pointer;
  KultivierterZeiger: PByteArray;
begin
  NatuerlicherZeiger := KultivierterZeiger; { ohne Typenumwandlung }
  { mit Typenumwandlung: }
  KultivierterZeiger := PByteArray(NatuerlicherZeiger);
```

Das Beispiel zeigt, dass untypisierte Zeiger überall dort verwendet werden können, wo auch andere Zeiger stehen. Um einen untypisierten Zeiger jedoch einem typisierten zuweisen zu können, müssen Sie dem Compiler über eine Typenumwandlung wie *PByteArray(NatuerlicherZeiger)* garantieren, dass der untypisierte Zeiger tatsächlich auf einen Speicherbereich zeigt, der mit dem Typ des typisierten Zeigers verträglich ist (siehe Abschnitt 2.4.7).

2.4.7 Typenkompatibilität und Typenumwandlungen

Mit den in Abschnitt 2.4.6 beschriebenen Record-Varianten können Sie bereits auf verschiedene Weise auf ein und denselben Speicherbereich zugreifen. Zu diesem und noch zu mehr in der Lage ist die Typenumwandlung (auf die Speicherbereiche kommen wir Ende des Abschnitts noch einmal zurück).

Kompatibilitätsregeln

Werfen wir zunächst einen Blick auf die Typenkompatibilitätsregeln von Object Pascal. Dass zwei Variablen, die denselben Typenbezeichner als Typ besitzen, sich gut vertragen, versteht sich von selbst. In vielen Fällen wäre es jedoch sehr unbequem, wenn nur identische Typen miteinander kooperieren könnten. Daher gibt es in Pascal zahlreiche Regeln zur *Typenkompatibilität*, die z.B. erlauben, dass in einem Ausdruck verschieden große Integervariablen (z.B. *Byte* und *Integer*) und Variablen von verschiedenen Fließkommatypen (z.B. *Real* und *Double*) miteinander »verrechnet« werden können. Andere Regeln betreffen z.B. die Kompatibilität verschiedener Prozedur- und Recordtypen, werden aber so selten benötigt, dass an dieser Stelle auf eine Object-Pascal-Referenz verwiesen sei.

Ein etwas strengeres Kriterium ist die *Zuweisungskompatibilität*, die z.B. nicht gegeben ist, wenn Sie eine *LongInt*-Variable in ein *Byte* zwängen wollen, denn das würde den Zahlenwert verstümmeln, sofern er größer als 255 oder kleiner als 0 ist. So achtet der Compiler bei Zuweisungen immer darauf, dass die rechte Seite der Zuweisung in die linke hineinpasst (dabei wandelt er bei Bedarf Integerwerte automatisch in Fließkommawerte um).

Umwandlung von Werten *R137*

Die Zuweisung einer größeren Variablen an eine kleine ist ein gutes Beispiel für Typenumwandlung. Wenn Sie beispielsweise wissen, dass eine Ihrer *Word*-Variablen nicht größer als 1000 werden kann, und Sie sie durch 10 teilen, können Sie danach beruhigt eine Typenumwandlung vornehmen. Dieses und weitere Beispiele im folgenden Listing:

```
ByteVar := Byte(WordVar div 10);
Zeichen := Char(10); { Umwandlung von Zahl in Char (Char(10) = #10) }
Falsch := Boolean(0); { Umwandlung von Zahl in Boolean }
```

Umwandlung von Variablen

Die Typenumwandlungen aus diesen Beispielen sind nur auf der rechten Seite der Zuweisung möglich, da sie einen Wert zum Ergebnis haben. Die andere Art der Typenumwandlung betrifft die Interpretation von Variablen, ähnlich den Varianten in den Records (besprochen in Kapitel 2.4.5).

```
type
  TCasedLongInt = record
     case Boolean of
       True: (LoWord, HiWord: Word;);
       False: (AsLongInt: LongInt;)
  end;
  TLongIntSplit = record
    Low, High: Word;
  end;

var
  L1: LongInt;
  L2: TCasedLongInt;
```

Um eine der *LongInt*-Variablen nun als *LongInt* und zugleich als zwei *Words* ansprechen zu können, haben Sie bei obigen Deklarationen zwei Möglichkeiten:

```
begin
{ Möglichkeit 1: }
  L2.AsLongInt := ... { als LongInt ansprechen }
  L2.LoWord := ... { eine Hälfte als Word ansprechen }
{ Möglichkeit 2: }
  L1 := ...  { als LongInt }
  TLongIntSplit(L1).Low := ...  { als Word }
```

Eine Typenumwandlung wie die in Möglichkeit 2 wandelt das auf der linken Seite der Zuweisung stehende Objekt in einen anderen Typ um. Dies ist nur erlaubt, wenn der neue Typ exakt die gleiche Größe hat wie der alte, so dass seine Struktur die Struktur der umzuinterpretierenden Variablen genau überdeckt.

Hinweis: Diese Einschränkung haben Sie bei Varianten in Records nicht, denn diese dürfen auch unterschiedlich lang sein.

2.4.8 Initialisierte Konstanten strukturierter Typen

Während Sie einfache Konstanten typisieren oder als echte Konstanten deklarieren können, müssen Sie bei strukturierten und Array-Konstanten immer den Typ angeben. Das folgende Beispiel zeigt die Syntax dieser Deklarationen:

```
const
  ObjectConst: TDatei = (
    Name: 'DCC32.EXE';
    Pfad: 'D:\BORLAND\DELPHI5\BIN'
  );
  ArrayConst: array[0..10] of Byte =
  ( 0, 0, 0, 100, 1, 100, 0, 0, 0, 1, 0);
```

Bei initialisierten Records müssen Sie nicht alle Elemente initialisieren, die initialisierten müssen jedoch in der richtigen Reihenfolge angegeben werden. Wichtig ist auch,

dass hinter der letzten Angabe kein Semikolon stehen darf. Bei initialisierten Arrays müssen Sie alle Elemente angeben und als Trennzeichen ein Komma verwenden.

Besonders bei den Arrays ist diese Art der Initialisierung bequemer, als wenn Sie das Array im Programm mit einzelnen Zuweisungen füllen würden. Darüber hinaus ist sie viel eleganter, denn zur Programmlaufzeit wird keine Zeit mit dem Kopieren einzelner Elemente verschwendet, da diese sich schon wohlgeordnet in Record- bzw. Array-Struktur in der EXE-Datei befinden. Im Falle einer Initialisierung per einzelner Zuweisung würden die einzelnen Werte zur Programmlaufzeit unter Umständen sogar doppelt vorliegen: Beim Programmstart wären sie nur einmal als einzelne Konstanten zu finden, bei der Zuweisung würden sie dann kopiert werden, was eine Verdopplung bedeutete.

2.5 Anweisungen und Funktionen

Das Thema dieses Kapitels sind reine Prozeduren und Funktionen, die auch ohne OOP verwendet werden können. Da die Funktions- und Prozedursyntax auch zur Deklaration von Methoden in Objektklassen Verwendung findet, sind diese Grundlagen jedoch keineswegs veraltet. Object Pascal weist außerdem gegenüber älteren Turbo-Pascal-Versionen Neuerungen wie offene Strings, offene Arrays und Prozedurtypen auf.

Zuerst fasst dieses Kapitel jedoch einen wichtigen Bestandteil für den Code der Prozeduren und Funktionen – die zur Sprache gehörenden Anweisungen – zusammen, die Object Pascal von Borland Pascal komplett erbt. Neue Kontrollstrukturen für Ausnahmebehandlung (Exceptions) werden in Kapitel 2.6 erläutert.

2.5.1 Pascal-Anweisungen

Die meisten Anweisungen der Sprache Pascal dienen zur Programmablaufsteuerung, sind also Schleifen, Bedingungsabfragen und Sprünge – alles Aufgaben, die mit Prozeduren und Funktionen alleine nicht zu bewältigen sind. Pascal verfügt mit der *with*-Anweisung über eine weitere, hauptsächlich der Bequemlichkeit dienende Anweisung, die nichts mit dem Programmablauf zu tun hat.

Die for-Schleife

Die einfachste Kontrollstruktur ist die *for*-Schleife, die eine oder mehrere Anweisungen mehrmals ausführt. Die Zahl der Wiederholungen steht vor der ersten Wiederholung fest:

```
x := a + b;
for i := a to b do
   WiederholteAnweisung;
```

Zu Beginn der Schleife setzt das Programm die Variable *i* auf den Wert *a*, erhöht sie am Ende jeder Schleife und führt diese das letzte Mal aus, wenn *i* den Wert hat, den *b* vor Beginn der Schleife besaß. Änderungen von *a* und *b* in der Schleife haben also keine Auswirkungen.

Blöcke

Alle Kontrollstrukturen können nicht nur einzelne Anweisungen, sondern auch Anweisungsblöcke ausführen. Außer bei der *repeat...until*-Schleife müssen diese in *begin* und *end* eingefasst werden, was im Beispiel der *for*-Schleife wie folgt aussieht:

```
for i := 1 to 100 do begin
   Anweisung1;
   Anweisung2;
   ...
end;
```

Bedingungen

Alle anderen Anweisungen zur Programmablaufsteuerung arbeiten mit expliziten Bedingungen. Bei der Auswertung einer Bedingung entsteht in Pascal immer entweder einer der Werte *True* und *False*, der besagt, ob die Bedingung erfüllt (*True*) ist oder nicht (*False*). Die Formulierung der Bedingungen ist einfach, da meist bekannte Vergleichsoperatoren der Mathematik Verwendung finden.

Die if-Anweisung

Die *if*-Anweisung besteht aus einem zu testenden Ausdruck und maximal zwei alternativen Anweisung(sblöck)en. Sie wertet den Ausdruck aus und führt, falls sie auf das Ergebnis *True* gekommen ist, die erste Alternative aus, andernfalls die zweite, hinter dem Schlüsselwort *else* angegebene, die jedoch auch weggelassen werden kann:

```
if Ausdruck
   then EsIstWahr
else EsIstFalsch;
```

Um nicht von den Unterschieden der *if*-Anweisungen in den verschiedenen Sprachen (z.B. im Vergleich zu C++) verwirrt zu werden, sollte man sich merken, dass in Pascal die gesamte *if*-Anweisung mitsamt dem *else*-Teil als eine einzelne Anweisung zählt und dass ein Semikolon nie innerhalb einer Anweisung stehen darf, also auch nicht vor dem *else* wie in C++. Auch die Interpretation der Schachtelung basiert auf diesem Grundsatz:

```
if Alternative1 then
   if Alternative2 then
     Anweisung1
   else Anweisung2;
```

Hier stellt sich bei jeder Sprache die Frage, zu welcher Abfrage *Anweisung2* zählt. In Pascal erwartet der Compiler nach dem ersten *then* eine komplette Anweisung, und die ist nach *Anweisung1* noch nicht zu Ende, da kein Semikolon dahinter steht (und wenn eins da stünde, wäre es ein Fehler, da es die äußere *if*-Anweisung unterbrechen würde). Das heißt also, dass *Anweisung2* in Object Pascal zur zweiten *if*-Anweisung gezählt wird. Das obige Beispiel könnte durch ein weiteres *else* ergänzt werden, das dann zur ersten *if*-Anweisung gehören würde (dann müsste das Semikolon hinter *Anweisung2* allerdings wieder gelöscht werden).

All diese Fragen stellen sich natürlich nicht, wenn Sie die zusammengehörenden Anweisungen durch *begin* und *end* kennzeichnen, wodurch Sie auch den obigen *else*-Teil einer anderen *if*-Ebene zuordnen können:

```
if Alternative1 then begin
  if Alternative2 then
     Anweisung1;
end
else Anweisung2;
```

Die repeat-Schleife

Kommen wir zur ersten Endlosschleifen-Kandidatin, der *repeat...until*-Schleife. Sie führt die zwischen den beiden Schlüsselwörtern stehenden Anweisungen so lange durch, bis der hinter *until* angegebene Ausdruck *True* ergibt. Das Folgende ist eine Endlosschleife, die jedoch in ihrem endlosen Tun keinen Sinn erkennen lässt:

```
repeat
   a := a+1;
   if a > 1 then a := 1;
until a > 9;
```

Die while-Schleife

Es ist nicht auf eine empfehlenswerte Weise möglich, den Rumpf der *repeat...until*-Schleife zu überspringen, da die Bedingung erst am Ende überprüft wird. Diese Tatsache macht diese Schleife zu einem Spezialfall der wirklich allgemeinsten Pascal-Schleife, der *while*-Schleife. Diese prüft schon zu Beginn, ob der Rumpf überhaupt ausgeführt werden soll:

```
{ Leeren eines Stacks: }
while not Stack.Empty do
   Stack.Pop;
```

Auch hier gelten für mehrere Anweisungen und für die Bedingung hinter dem *while* die bekannten Regeln.

Die case-Anweisung

Die *case*-Anweisung hilft, komplizierte *if*-Abfragen zu vermeiden, indem sie praktisch eine Tabelle mit möglichen Werten und den dazugehörigen Funktionen enthält:

```
case Key of
   VK_F1: Anweisung1;
   VK_F2: begin
     Anweisungen;
   end;
   VK_, VK_...:
   else ...
end;
```

Die Variable hinter dem *case* muss einen ordinalen Typ haben, der maximal die Größe eines *Word* haben darf. Sie können mehrere Bedingungen wie im Beispiel durch Kommas trennen und gemeinsam behandeln. Alle übrig bleibenden Bedingungen können optional im *else*-Teil mit einer speziellen Aktion behandelt werden.

Ist eine Bedingung gefunden, so wird der *case*-Block nach den dahinter stehenden Anweisungen vollständig beendet, es ist dazu also nicht wie etwa in C extra ein *break* erforderlich.

Sprünge

In Object Pascal gibt es vier Arten von Sprüngen:

- Mit *Exit*, das ausnahmsweise kein Schlüsselwort ist, verlassen Sie eine Prozedur oder Funktion sofort, was gleichbedeutend ist mit einem Sprung zum *end* der Funktion. Mit dieser Anweisung lassen sich unter Umständen eine oder mehrere Ebenen von *if*-Anweisungen umgehen. Eine jüngere Sprungart macht die *exit*-Anweisung jedoch unnötig:
- Exceptions sind die modernste Art von Sprüngen, unter Hinzunahme der anderen Kontrollstrukturen lassen sich sowohl *exit*- als auch *goto*-Sprünge leicht vermeiden. Ihnen gehört ein eigenes Kapitel dieses Buchs (Kapitel 2.6).
- Sprünge in Schleifen: Mit *continue* springen Sie innerhalb einer Schleife zurück an den Schleifenanfang, wo die Schleife bei der nächsten Wiederholung fortgesetzt wird. Mit *break* verlassen Sie eine Schleife sofort.
- Für besonders waghalsige Sprünge stehen weiterhin die *goto*-Anweisungen zur Verfügung, die zu einer Zahlenmarke springen, die vorher in einem *label*-Block wenigstens scheinbar zivilisiert vorangekündigt werden muss. Aus technischen

Gründen (Stack) ist es außerdem nicht möglich, zwischen verschiedenen Funktionen herumzuspringen. Wir verzichten hier auf ein Beispiel.

Aufgrund besonderer Grobheit verwandt mit *break*, *continue* und *exit* ist außerdem die Anweisung *Halt*. Sie beendet das Programm, ist aber intern erheblich komplexer aufgebaut als ein normaler Sprung, da beispielsweise die *finalization*-Abschnitte aller Units ausgeführt werden müssen.

Die with-Anweisung

Die einzige Kontrollstruktur von Object Pascal, zu der es kein C++-Äquivalent gibt, ist die *with*-Anweisung. In ihr geben Sie einen Gültigkeitsbereich an, in dem der Compiler zuerst nach den Bezeichnern suchen soll – die er dann natürlich auch verwendet. Mit der *with*-Anweisung können Sie statt

```
DefaultFontStruct.Name := 'Phantom';
DefaultFontStruct.Height := 20;
DefaultFontStruct.UseRaytracing := True;
```

einfach schreiben:

```
with DefaultFontStruct do begin
   Name := 'Phantom';
   Height := 20;
   UseRaytracing := True;
end;
```

Durch die *with*-Anweisung werden also eventuell Bezeichner im aktuellen Gültigkeitsbereich verdeckt. Sie können hinter dem *with* sogar mehrere durch Kommas getrennte Gültigkeitsbereiche angeben und *with*-Blöcke schachteln. Diese Möglichkeiten sind in der Praxis jedoch selten sinnvoll anwendbar.

2.5.2 Prozeduren und Funktionen

Die Routinen und Methoden werden in Object Pascal nach Prozeduren und Funktionen unterschieden – wohl eher aus traditionellen als aus logischen Gründen, denn auch Funktionen, die mehrere Ergebnisse nicht in einer Struktur zurückgeben wollen, werden häufig in eine Prozedur umgewandelt.

Theoretisch unterscheidet sich die Funktion durch die Rückgabe eines Ergebnisses von der Prozedur. Dieses geben Sie hinter einem Doppelpunkt nach der Parameterliste an:

```
function BerechneTangens(f: real): real;
begin
  Result := sin(f) / cos(f);
end;
```

```
procedure FuehreGeheimauftragDurch(ae34, bx45, cd77: LongInt;
                                   returntime: Word);
begin
  ...
end;
```

Die Regeln für die Parameterliste sind bei Prozeduren und Funktionen identisch: Die Syntax ist dieselbe wie innerhalb eines *var*-Blocks, allerdings hat das Schlüsselwort *var* hier eine andere Bedeutung, wie der nächste Abschnitt erläutern wird.

Zum Aufrufen einer Prozedur schreiben Sie deren Namen, gefolgt von der Liste der Parameter, wie z.B.:

```
FuehreGeheimauftragDurch(12, 9978, 9990000, 1010);
```

Bei Aufrufen von Prozeduren ohne Parameter dürfen dem Prozedurnamen keine leeren Klammern (wie in C) folgen. Für Funktionen gilt dasselbe mit dem Zusatz, dass das Funktionsergebnis für irgendeinen Zweck verwendet werden sollte, wie oben in der Beispielfunktion mit dem Ergebnis von *cos* geschehen. Funktionsergebnisse können auch ignoriert werden, so dass Sie die Funktionen wie Prozeduren aufrufen können (dazu muss allerdings die Compileroption ERWEITERTE SYNTAX / *$X* eingeschaltet sein, was standardmäßig der Fall ist).

Funktionsergebnisse

Eine der vielen kleinen, aber sehr nützlichen Erweiterungen von Object Pascal ist die Variable *Result*, die automatisch in jeder Funktion definiert ist. Sie ist eine neue Bezeichnung für etwas, das es in Pascal schon immer gab: eine Speicherstelle für das Funktionsergebnis. Früher hatte diese Speicherstelle immer den Namen der Funktion, nun können Sie sie alternativ unter dem Namen *Result* ansprechen. Dies macht es erheblich einfacher, die Funktion umzubenennen (oder zwecks geringfügiger Veränderung zu kopieren, was trotz OOP noch ab und zu vorkommen soll ...).

Vorwärtsdeklarationen

Wenn sich zwei Funktionen gegenseitig benutzen, müssen Sie eine davon in einer Vorwärtsdeklaration erwähnen:

```
procedure ProcBbenutztA; forward; { Vorwärtsdeklaration }
procedure ProcAbenutztB;
begin
  ProcBbenutztA;
end;
procedure ProcBbenutztA;
  { Definition der vorwärts deklarierten Prozedur }
begin
  ProcAbenutztB;
...
```

Die erste Funktion könnte die zweite nicht aufrufen, wenn der Compiler diese nicht schon durch die Vorwärtsdeklaration kennen würde.

Solche Vorwärtsdeklarationen sind bei Methoden natürlich nicht erforderlich, da die Klassendeklaration bereits quasi eine einzige große Vorwärtsdeklaration für alle Methoden ist.

Aufrufkonventionen

Hinter einer Funktionsdeklaration können Sie eine Direktive angeben, die festlegt, auf welche Art die Funktion aufgerufen wird:

- *Register* ist die in 32-Bit-Delphi voreingestellte Methode, bei der der Compiler versucht, möglichst viele Parameter zu übergeben, ohne dafür den Stack in Anspruch zu nehmen. Einige Parameter werden so direkt in den Registern des Prozessors an die Funktion übergeben, was natürlich erheblich schneller geht als die Inanspruchnahme des im Hauptspeicher befindlichen Stacks.
- *Pascal* ist eine etwas antiquiert anmutende Methode, die seit Turbo Pascal und bis zur ersten Delphi-Version als Standardmethode verwendet wurde. Die Parameter werden hier in der Reihenfolge auf den Stack gelegt, in der sie deklariert werden. Auch C-Funktionen in DLLs arbeiten manchmal nach dieser Konvention.
- *Cdecl* ist nach der Sprache C benannt, in der die Funktionen ihre Parameter traditionell in umgekehrter Reihenfolge erwarten und zusätzlich voraussetzen, dass der Aufrufer der Funktion einige Stack-Aufräumarbeiten durchführt.
- *StdCall* ist eine Art Mischung zwischen *pascal* (für die Reihenfolge) und *cdecl* (für die Stack-Verwaltung). *StdCall* wird von Windows-API-Funktionen verwendet.
- *SafeCall* entspricht *StdCall* mit einer zusätzlichen Fehler- und Ausnahmebehandlung. Mit dieser Aufrufkonvention wird in der COM-Automation sichergestellt, dass der vom Compiler erzeugte Code sich in Fehler- und Ausnahmebehandlung an die COM-Richtlinien hält.

2.5.3 Parametertypen

Zuerst zwei Begriffserklärungen: Die Parameter, die Sie in der Deklaration einer Funktion angeben, heißen *formale Parameter*, während die Parameter, mit denen die Funktion zur Laufzeit arbeitet, als *aktuelle Parameter* und in der deutschen Übersetzung der Compiler-Fehlermeldungen auch als »*wirkliche Parameter*« bezeichnet werden.

Wertparameter

Für Parameter gibt es in Pascal, wie in den meisten anderen Sprachen auch, zwei Möglichkeiten der Übergabe: Entweder handelt es sich bei den aktuellen Parametern um

die Originalvariablen, die Sie an die Funktion übergeben haben, oder um Kopien davon. Im letzteren Fall kann die Funktion den Wert der Parameter ändern, ohne dass sich das auf die aufrufende Funktion auswirkt, solche Parameter heißen *Wertparameter*:

```
procedure PaintCells(von, bis: Integer);
begin
  for von := von to bis do begin
     ...
  end;
end;

var
  Start, Ende: Integer;
begin
  Start := 0;
  Ende := 10;
  PaintCells(Start, Ende);
```

In diesem Beispiel wird die Variable *Start* nicht verändert, da *PaintCells* eine Kopie des Wertes in einer neuen Variablen *von* erhält. Diese Variable liegt auf dem Stack und wird ungültig, sobald *PaintCells* beendet wird.

Variablenparameter

Damit die Prozedur aus dem letzten Beispiel den Originalparameter *Start* erhält, müsste sie wie folgt deklariert werden (was natürlich in diesem Fall nicht sinnvoll wäre):

```
procedure PaintCells(var von: Integer; bis: Integer);
```

Das Schlüsselwort *var* teilt dem Compiler mit, dass er der Funktion einen Zeiger auf den Originalparameter übergeben soll. Der Name *von* würde sich dann auf denselben Speicherplatz beziehen wie die Variable *Start*. Es versteht sich von selbst, dass Sie bei Variablenparametern keine Werte übergeben können, die quasi obdachlos sind, weil sie keine Adresse im Speicher haben (z.B. direkt angegebene Zahlen).

Nützlich sind Variablenparameter nicht nur zum Verändern der Werte, sondern auch zum Sparen von Stack-Speicher und zu einem schnelleren Aufruf der Prozedur/Funktion:

```
type
  TBigArray: array[0..100] of LongInt;

procedure TakeArray(var A: TBigArray);
```

Ohne das Schlüsselwort *var* würde das Programm bei jedem Aufruf von *TakeArray* die 400 Bytes des Array-Parameters auf den Stack kopieren. Mit *var* wird lediglich ein Zeiger auf das Array übergeben, der 4 Bytes benötigt. Allerdings kann die Prozedur das

Original-Array nun verändern. Um diesen Nachteil von *var* auszugleichen, ist das *const*-Schlüsselwort auch für Parameter anwendbar.

Const-Parameter

Object Pascal bietet die Möglichkeit, einen Parameter als *const* zu deklarieren. Der Compiler erlaubt dann nicht, dass dieser Parameter innerhalb der Prozedur verändert wird. Wenn es sich bei dem Parameter sowieso nur um eine Kopie handelte, wäre das sicher nicht sinnvoll, daher wird ein *const*-Parameter so übergeben wie ein *var*-Parameter. Dies ist bei Variablen, die viel Speicher in Anspruch nehmen, die effektivere Lösung, da anstatt einer kompletten Kopie nur ein Zeiger übergeben werden muss.

Untypisierte Parameter

Des Weiteren gibt es die Möglichkeit, Parameter ohne einen festgelegten Typ zu deklarieren. In diesem Fall muss es sich entweder um einen *var* oder um einen *const*-Parameter handeln:

```
procedure InterpretiereDatenBlock(const Datenblock);
```

Der Parameter *Datenblock* hat keinen Typ, was den Vorteil hat, dass er in jeden beliebigen Typ umgewandelt werden kann, und den Nachteil, dass der Compiler nicht überprüfen kann, ob diese Umwandlung zulässig ist. Die Prozedur *InterpretiereDatenBlock* könnte den Datenblock beispielsweise in einen privaten Recordtypen umwandeln, der mit der Struktur der Daten übereinstimmt (siehe Typenumwandlung in Kapitel 2.4.7).

Offene Arrays

Im Zusammenhang mit Arrays gibt es als letzte Parametervariante die des *offenen Arrays* und speziell für Strings die *offenen Strings* – sofern Sie die voreingestellte Compileroption *$P+* (*Offene Arraygrenzen*) nicht ausgeschaltet haben. Diese Parameter heißen offen, da im formalen Parameter (also in der Deklaration) nicht festgelegt wird, wie viele Elemente das Array oder wie viele Zeichen der String hat. Das heißt, dass Sie Strings und Arrays beliebiger Größe als (aktuelle) Parameter übergeben können. Sie müssen beim offenen Array lediglich den Typ der Elemente angeben und haben darüber hinaus die Wahl zwischen normaler, *const*- und *var*-Parameterübergabe:

```
function SummiereArrayKopie(a: array of Integer): Integer;
function SummiereArrayDirekt(const a: array of Integer): Integer;
procedure SortiereArray(var a: array of Integer);
```

Die Unterschiede zwischen *var*-, *const*- und normalem Aufruf stimmen mit dem von normalen Variablen überein, die Prozedur *SummiereArrayKopie* ist also zur Laufzeit deutlich weniger effektiv als *SummiereArrayDirekt*, da sie eventuell große Kopieraktionen notwendig macht, die aufgrund der Offenheit des Arrays nicht vorhersehbar sind.

Einen offenen formalen Array-Parameter können Sie auf zwei verschiedene Weisen mit einem aktuellen Array-Parameter füllen:

```
var
  ParamArray: array[0..10000] of Integer;
begin
  SortiereArray(ParamArray); { durch Angabe einer Array-Variablen }
  SummiereArrayKopie([322, 223, 443, 988]); { mit einem temporären Array }
```

SortiereArray nimmt beliebige Integer-Arrays an, daher können Sie die Variable *ParamArray* ohne Bedenken übergeben. In der zweiten Anweisung wurde keine Array-Variable verwendet, sondern das im nächsten Abschnitt beschriebene Konstrukt.

Der Open-Array-Konstruktor

Die zweite Anweisung im letzten Beispiel verwendet den neuen *Open-Array-Konstruktor* von Object Pascal, der unscheinbarerweise einfach aus eckigen Klammern besteht. In diesen eckigen Klammern geben Sie eine Liste von Werten an, die zu einem temporären Array zusammengesetzt werden, das dann anschließend an die Funktion weitergegeben wird. Bei der Funktion *SummiereArray* im Beispiel kommt das konstruierte Array so an, als hätten Sie es wie folgt deklariert:

```
const
  TempArray: array[0..3] of Integer =
   (322, 223, 443, 988);
```

Es ist logisch, dass ein so konstruiertes Array als Variablenparameter nicht sinnvoll ist, weil die veränderten Werte sofort nach dem Aufruf wieder verloren gehen – schließlich handelt es sich um ein temporäres Array.

Arbeiten mit offenen Arrays

Die sinnvolle Verwendung von offenen Arrays ist nur möglich, wenn Sie innerhalb der Prozedur feststellen können, wie groß das Array tatsächlich ist. Innerhalb der Prozedur können Sie daher die folgenden Funktionen auf das Array anwenden:

```
StartIndex := Low(a); { Index des ersten Elements (ist immer 0) }
EndIndex := High(a); { Index des letzten Array-Elements }
GesamtArrayGroesse := SizeOf(a);
```

Da das erste Element den Index 0 hat, ist die Gesamtzahl der Elemente *EndIndex+1*. Wenn Sie die Bereichsüberprüfung *$R* eingeschaltet haben, generiert der Compiler zusätzlichen Code, der jeden Zugriff auf das offene Array auf die Einhaltung der tatsächlichen Array-Grenzen überprüft (siehe Kapitel 2.6.5).

Hinweis: Den offenen Arrays sehr ähnlich sind die offenen Strings (Typ *OpenString* bzw. Compileroption *$P*), die jedoch aufgrund der langen Strings seit Delphi 2 nicht mehr benötigt werden und hier nur der Vollständigkeit halber erwähnt werden sollen.

Variable Parameterlisten

Der Open-Array-Konstruktor nimmt bereits eine beliebige Anzahl von Parametern auf, aus denen er ein temporäres Array macht, das Sie einer Prozedur mit offenem Array-Parameter übergeben können. Auf diese Weise könnte man schon fast variable Parameterlisten in Object Pascal nachbilden. An der Simulation einer wirklich variablen Parameterliste hindert nur noch die Beschränkung, dass Array-Parameter nur Werte eines einzigen Typs übernehmen können.

Als ersten Schritt zur Lösung bietet die Unit *System* den Typ *TVarRec* an, der je nach Delphi-Version Varianten für 9 bis 16 verschiedene Parametertypen – inklusive Objektinstanzen und Klassenreferenzen – aufweist:

```
type
  TVarRec = record
    case Byte of
      vtInteger: (VInteger: LongInt { ab Delphi 2: Integer}; VType: Byte);
      vtBoolean: (VBoolean: Boolean);
      vtChar:    (VChar: Char);
      vtExtended:(VExtended: PExtended);
      vtString:  (VString: PString { ab Delphi 2: PShortString } );
      vtPointer: (VPointer: Pointer);
      vtPChar:   (VPChar: PChar);
      vtObject:  (VObject: TObject);
      vtClass:   (VClass: TClass);
     { neu in Delphi 2: }
      vtWideChar:   (VWideChar: WideChar);
      vtPWideChar:  (VPWideChar: PWideChar);
      vtAnsiString: (VAnsiString: Pointer);
      vtCurrency:   (VCurrency: PCurrency);
      vtVariant:    (VVariant: PVariant);
     { neu in Delphi 3: }
      vtInterface:  (VInterface: Pointer);
      vtWideString: (VWideString: Pointer);
     { neu in Delphi 4: }
      vtInt64:      (VInt64: PInt64);
     { keine Neuigkeit in Delphi 5 und Delphi 6 }
    end;
```

Für alle Typen, die größer als 4 Bytes sind, speichert *TVarRec* einen Zeiger, der auf das Element weist, so dass in jedem Fall 5 Bytes zur Speicherung eines *TVarRec* genügen (inklusive dem Byte *VType*, das in jedem Fall den Typ des Elements angibt, auch, wenn es nur in der Alternative *vtInteger* deklariert ist). Wenn Sie ein Parameter als ein offenes

Array von *TVarRec*-Elementen deklarieren, können Sie bereits Parameter beliebiger Anzahl und beliebigen Typs entgegennehmen. Es kommen jedoch noch zwei kleine Erweiterungen von Object Pascal hinzu, die den Vorgang weiter vereinfachen:

- Der offene Array-Konstruktor ist nun auch in der Lage, *TVarRec*-Arrays automatisch zu konstruieren. Umschließen Sie einfach die zu übergebenden Parameter in eckigen Klammern, und der Array-Konstruktor macht daraus ein *TVarRec*-Array, wobei er auch das Feld *TVarRec.Type* selbstständig ausfüllt.
- Schließlich gibt es für derartige offene Arrays eine eigene Schreibweise: *array of const*.

Eigene Funktionen mit variablen Parameterlisten *R139*

Das folgende Beispiel fasst die Verwendung des offenen Arrays mit variablen Elementtypen zusammen und zeigt, wie die Funktion *WriteLn* mit einem offenen Array-Parameter nachgebildet werden kann: (*Writeln* schreibt eine Zeile in eine Textdatei, wobei die Zeile aus einer beliebigen Folge von Strings und verschiedenen anderen Variablen bestehen kann.)

```
function NewWriteln(var f: Text; const Args: array of const): string;
var
  i: Integer;
begin
  for i := 0 to High(Args) do
    case Args[i].Type of
      tkInteger: ...
end;

var
  Str: string;
  Int: Integer;
  Ext: Extended;
begin
  { Vergleich des Aufrufs einer System-Funktion mit variabler
    Parameterliste ... }
  System.Writeln(Zieldatei, 'Ausgabe dreier Variablen: ',
                 Str, ' ', Int, ' ', Ext);
  { ... mit dem Aufruf einer selbst geschriebenen Funktion mit
    einem offenen TVarRec-Array (array of const): }
  NewWriteln(Zieldatei, ['Ausgabe dreier Variablen: ',
                         Str, ' ', Int, ' ', Ext]);
```

Formal hat *NewWriteln* zwar immer nur zwei Parameter, durch die einfache Konstruktion des offenen Arrays bleibt aber kaum noch ein Unterschied zur völlig freien Parameterangabe beim Original-*Writeln*.

Laufzeitbibliothek und VCL enthalten bereits mehrere Funktionen, die offene Parameter dieser Art erhalten, z.B. die Funktion *Format*, die Strings ähnlich wie die C-Funktion *sprintf* formatiert, die Menü-Konstruktionsfunktionen (Kapitel 4.6) und einige Methoden von *T(Client)DataSet* (Kapitel 7.4). Zwar ist die Konstruktion dieser temporären Parameter nicht gerade zeitsparend, aber solange Sie eine solche Aktion nicht gleich tausend Mal hintereinander ausführen, dürfte kaum Gefahr für die Programmgeschwindigkeit bestehen.

2.5.4 Überladen von Funktionen, Standardparameter

Haben Sie sich schon gefragt, warum es in der Laufzeitbibliothek von Delphi Routinen gibt, die auf eine viel flexiblere Art aufgerufen werden können als Ihre selbst geschriebenen Routinen? Zwei simple Beispiele dafür sind die Prozeduren *inc* und *write*. Bei *inc* können Sie einen optionalen Parameter angeben:

```
inc(Zaehler);    // erhöht den Zaehler um den Standardwert 1
inc(Zaehler, 5); // erhöht den Zaehler um 5
```

Mit *write* können Sie verschiedene Datentypen beispielsweise in eine Textdatei schreiben:

```
var
  Zahl: Integer;
begin
  write(Textdatei, Zahl); // schreibt eine Variable des Typs Integer
  write(Textdatei, '. Element'); // schreibt einen String
```

Seit Delphi 4 erlaubt es auch Object Pascal jedem Entwickler, Prozeduren, Funktionen und Methoden zu schreiben, die in dieser variablen Art aufgerufen werden können. Dies funktioniert im Prinzip übrigens genauso wie die entsprechenden Techniken von C++.

Standardparameter

Wenn Sie eine Funktion haben, bei der einige Parameter meistens mit einem Standardwert belegt werden, so können Sie es dem Aufrufer der Funktion ersparen, diesen Standardwert jedes Mal aufschreiben zu müssen, indem Sie den Standardwert schon in der Funktionsdeklaration angeben:

```
// Eine Liste von Strings wird in einem einzigen String aneinander
// gereiht, optional kann eine maximale Länge der Einzelstrings und ein
// Trennzeichen angegeben werden:
function MakeString(Einzelwerte: TStringList; Trennzeichen: char = ';';
           MaxLen: Integer = 0) // 0 steht für "keine Längenbegrenzung"
         : String;
```

Wichtig ist für die Deklaration, dass alle Parameter, die einen Standardwert haben, am Schluss der Parameterliste stehen müssen. Dies erlaubt dem Compiler beim Aufruf der Methode, alleine an der Zahl der angegebenen Parameter festzustellen, welche Parameter mit Standardwerten zu belegen sind. Die Beispiel-Funktion kann auf drei verschiedene Arten aufgerufen werden:

```
Str := MakeString(Liste);   // Erzeugung eines Strings nach Voreinstellung
Str := MakeString(Liste, ','); // statt ';' ein ',' zur Trennung verwenden
Str := MakeString(Liste, ',', 30); // zusätzlich die einzelnen Strings
                                   // nach 30 Zeichen abschneiden
```

Für den Aufruf der Funktion gilt eine ähnliche Regel wie für die Deklaration: Parameter dürfen nicht an beliebiger Stelle weggelassen werden, sondern nur am Schluss. Wenn Sie also den dritten Parameter angeben wollen, dürfen Sie den zweiten nicht weglassen, selbst wenn dieser über eine Standardeinstellung verfügt. Im Beispiel müsste also der voreingestellte Wert für den zweiten Parameter wiederholt werden, damit die Voreinstellung für den dritten Wert überschrieben werden kann:

```
Str := MakeString(Liste, ';', 30);
```

Für den Rumpf der Funktion ändert sich nichts, dieser greift genauso auf die einzelnen Parameter zu, als wenn keine Standardwerte existieren würden.

Hinweis: Standardparameter eignen sich auch gut für den Fall, dass Sie eine bereits häufig verwendete Methode um neue Parameter erweitern wollen, die Sie aber nicht in allen schon bestehenden Methodenaufrufen dazuschreiben wollen. Wenn Sie die neuen Parameter an den Schluss der Parameterliste anhängen und mit Standardwerten versehen, mit denen sich die schon bestehenden Aufrufe vertragen, müssen Sie die bestehenden Aufrufe nicht ändern.

Überladen von Funktionen

Auch Routinen, die sich wie die vordefinierte *write*-Prozedur mit verschiedenen Parametertypen aufrufen lassen, können Sie nun selbst definieren. Für die *write*-Aufrufe aus dem einleitenden Beispiel hätten Sie z.B. die beiden folgenden Prozeduren deklarieren können:

```
procedure write(var f: file; i: Integer); overload;
procedure write(var f: file; s: String); overload;
```

Da es ja schon Prozeduren gibt, die Integers und Strings in eine Datei schreiben, wäre es sicher sinnvoller, eine Prozedur zu schreiben, die einen eigenen Datentypen in eine Datei schreibt:

```
procedure write(var f: file; Rec: TMeineDatenstruktur); overload;
```

Wichtig ist nur, dass Sie die folgenden Dinge beachten:

- Die *overload*-Direktive muss angegeben werden, damit der Compiler die neue Funktion wirklich nur als Alternative zur bisher bestehenden Funktion ansieht und nicht als *Ersatz* (wobei ein solcher Ersatz nur dann erlaubt wäre, wenn die beiden Funktionen in verschiedenen Gültigkeitsbereichen deklariert worden wären).
- Alle überladenen Funktionen mit dem gleichen Namen müssen anhand der Typen ihrer Parameter oder an ihrem Rückgabetyp eindeutig unterscheidbar sein.
- Standardparameter können die geforderte Unterscheidbarkeit erschweren, da eine Funktion mit einem Standardparameter bereits mit zwei verschiedenen Arten von Parameterlisten aufgerufen werden kann.
- Und schließlich: Es müssen *alle* Funktionen, die als Alternativen gelten sollen, mit *overload* deklariert sein. Da die vordefinierte *write*-Prozedur *nicht* mit *overload* deklariert ist, ist es nicht möglich, sie für den Datentyp *TMeineDatenstruktur* zu überschreiben. Die obige Deklaration von *write(var f: file; Rec: TMeineDatenstruktur)* würde also die vordefinierte *write*-Prozedur *ersetzen* und nicht überladen. Um die normalen Datentypen schreiben zu können, müssten Sie die vordefinierte Prozedur dann mit *System.write* aufrufen. Entsprechendes gilt auch für Methoden und für geerbte Methoden: Auch diese werden nach wie vor von gleichnamigen Methoden verdeckt, sofern nicht beide Methoden mit *overload* deklariert sind.

Wenn schon nicht die *write*-Prozedur, so gibt es doch zumindest ein paar vordefinierte Routinen, die schon als *overload* deklariert sind, so z.B. in der *SysUtils*-Unit:

```
function Min(A,B: Integer): Integer; overload;
function Min(A,B: Int64): Int64; overload;
function Min(A,B: Double): Double; overload;
```

2.5.5 Prozedurtypen

Die auf Methodenzeiger erweiterten Prozedurtypen spielen in der VCL eine Schlüsselrolle bei der Verteilung der Nachrichten: Wenn Sie im Objektinspektor dem Ereignis *OnClick* eine Methode zuweisen, setzen Sie in Wirklichkeit das Property namens *OnClick* auf einen Zeiger zu dieser Methode. Alle Properties auf der Ereignisseite sind derartige Zeiger auf Methoden.

Beginnen wir bei den normalen Prozedurtypen: Einen Zeigertyp auf eine Funktion, die eine Fließkommazahl als Parameter erwartet und eine ebensolche als Ergebnis zurückliefert, deklarieren Sie z.B. wie folgt:

```
type
  TRealFunction = function(number: real): real;
```

Der Unterschied zu einer normalen Funktionsdeklaration ist also nur, dass statt eines Funktionsbezeichners *hinter* »*function*« ein Typenname und ein Gleichheitszeichen *vor* »*function*« steht. Zur Anwendung dieses Typs weisen Sie ihn einfach einer Variablen zu und setzen diese Variable auf eine Funktion, deren Parameter und Rückgabewerte mit denen der obigen Deklaration übereinstimmen. Im folgenden Beispiel wird der Prozedurparameter *Funktion* mit dem Typ *TRealFunction* deklariert:

```
procedure ZeichneKurve(Funktion: TRealFunction);
function Funktion1(x: real): real;
function Funktion2(x: real): real;
...
ZeichneKurve(Funktion1);
ZeichneKurve(Funktion2);
```

Die beiden Funktionen müssen so deklariert sein wie oben gezeigt, nur der Name des Parameters darf abweichen.

Methodenzeiger

Die kleine Erweiterung, die die schon etwas älteren Prozedurtypen für das OOP erfahren haben, besteht aus den beiden Schlüsselwörtern *of* und *object*. Der Typ für Methoden, die das *OnClick*-Ereignis bearbeiten können, ist beispielsweise wie folgt deklariert:

```
type
  TNotifyEvent = procedure(Sender: TObject) of object;
```

Methodenzeiger sind nicht mit Zeigern auf normale Funktionen kompatibel, da Methoden zusätzlich einen *self*-Zeiger auf das Objekt als unsichtbaren Parameter benötigen (zu *self* siehe Kapitel 2.2.4) und da Methodenzeiger neben der Methode noch das Objekt speichern, von dem sie stammen, und das später zum *self*-Parameter wird (dies ist ein wesentlicher Vorteil gegenüber den Methodenzeigern von C++).

2.6 Fehlerbehandlung mit Exceptions

Seit der Einführung von Delphi können auch Pascal-Programmierer das tun, was vorher verschiedenen Präsidenten, Regierungen und C++-Programmierern vorbehalten war: Ausnahmezustände verhängen. Die allgemeine Bezeichnung als *Ausnahmen* kann Sie daran erinnern, dass Fehlersituationen nicht das einzige Einsatzgebiet der *Exceptions* sind, sondern dass damit beispielsweise auch besonders freudige Ereignisse behandelt werden können (beispielsweise, wenn sich die Größe der Festplatte völlig unerwartet verzehnfacht).

Die besondere Natur der Exceptions

Exceptions sind völlig anders als alle bisher behandelten Kontrollstrukturen: Sie führen, wenn tatsächlich eine Ausnahmebedingung auftritt, zu einem komplexen Programmablauf, der ohne Exceptions nur durch zusätzliche Funktionsparameter und die Kombination vieler anderer Kontrollstrukturen oder *goto*-Anweisungen durchführbar wäre. Innerhalb dieses komplexen Systems werden nun Objekte befördert, deren Aufbau viel einfacher ist als beispielsweise der Aufbau der Komponenten. Trotz des einfachen Aufbaus handelt es sich um echte Objekte, deren Klassen in einer Klassenhierarchie angeordnet sind, die fast komplizierter ist als die Objekte selber. Tatsächlich dienen die Objekte oft nur dazu, eine bestimmte Klasse zu haben, aber keinen wichtigen Inhalt. Mit diesen Merkwürdigkeiten müssen Sie rechnen, wenn Sie das Folgende lesen.

2.6.1 Verhängung des Ausnahmezustandes

Wir beginnen damit, den Ursprung einer Exception zu betrachten, und versetzen uns in die Lage einer Funktion, die versucht, eine Datei zum Lesen zu öffnen, die jedoch nicht existiert. Angenommen, sie ruft eine Funktion *CallDosInterrupt* auf, die frei erfunden ist und nur als Beispiel dient:

```
(* Datei vom Betriebssystem öffnen lassen *)
ErrorCode := CallDosInterrupt(xyz, Dateiname);
(* Fehlerstatus überprüfen: *)
if ErrorCode <> 0 then
  raise EFileNotFound.Create('Datei konnte nicht gefunden werden.',
                             ErrorCode);
```

Die letzte Zeile ist der Ursprung einer Exception der Klasse *EFileNotFound*. Die eigentliche Erzeugung des Exception-Objekts läuft so ab wie bei jedem anderen Objekt auch: über den *Create*-Konstruktor der zugehörigen Klassen. Um den besonderen Programmablauf der Ausnahmebehandlung zu starten, ist zusätzlich das Schlüsselwort *raise* notwendig. Mit der Bedeutung von »Erheben« zeigt dieses bereits gut an, was geschieht: Die aktuelle Funktion wird verlassen und die Exception eine Ebene höher an die aufrufende Funktion übergeben. Wie diese darauf reagieren kann, zeigt das Kapitel 2.6.4.

2.6.2 Exception-Klassen

Statt für jeden Fehler einen Fehlercode zur Verfügung zu stellen, zu dem es dann womöglich noch eine Fehlermeldung gibt, enthalten die Units der VCL für jeden Fehlertyp eine eigene Exception-Klasse. Jede davon ist direkt oder indirekt von der Klasse *Exception* abgeleitet, die in der Unit *SysUtils* deklariert wird.

Daten der Exceptions

Die Klasse *Exception* enthält das, was alle Exceptions übereinstimmend benötigen: einen String für die Fehlermeldung, den Sie über das Property *Exception.Message* ansprechen können, und eine Hilfekontextnummer, die den Benutzer vom Hilfeschalter des Fehlermeldungsfensters in die Hilfedatei führen kann.

In der Vererbungshierarchie kann nun jede Exception-Klasse diese Meldung mit weiteren Informationen ergänzen. Von den in Delphi vordefinierten Exception-Klassen machen jedoch nur wenige von dieser Möglichkeit Gebrauch, so etwa die Klassen *EInOutError* und *EOsError*, die das zusätzliche Datenelement *ErrorCode* für den Fehlercode des Betriebssystems enthalten. Die einzige Information, die Sie ansonsten neben der Fehlermeldung und dem Hilfekontext haben, ist die Klasse, zu der die Exception gehört. So kann beispielsweise bei Divisionen die Exception *EZeroDivide* auftreten, die zur Klasse der *EMathError*-Exceptions gehört.

Eigene Exceptions definieren

Alle vordefinierten Exceptions können Sie natürlich auch in Ihrem Code erzeugen. Wenn in Ihrem Programm ein neuer Fehlertyp auftreten kann, den Sie ebenfalls in die Ausnahmebehandlung mit aufnehmen möchten, können Sie auch eigene Exception-Klassen deklarieren, die Sie wahrscheinlich in die schon existierende Hierarchie einfügen möchten, z.B.:

```
type
  ETableOverflow = class (EOutOfMemory)
    public
      TableSize: LongInt;
  end;
```

Diese Exception könnten Sie beispielsweise dann »erheben«, wenn in einer internen Tabelle des Programms kein Speicher mehr vorhanden ist, sei es, weil die Tabelle statisch ist oder weil kein zusätzlicher dynamischer Speicher mehr vorhanden ist, durch den die Tabelle erweitert werden könnte. Sie könnten auch direkt die Klasse *EOutOfMemory* verwenden, wenn Sie das oben deklarierte Datenelement *TableSize* nicht benötigten, aber dann könnten Sie nicht zwischen Ihrer und den vordefinierten Exceptions unterscheiden (zur Abfrage der Exception-Klasse siehe Kapitel 2.6.4).

Dieses Beispiel zeigt auch, warum Exception-Klassen so wenige Daten enthalten: *TableSize* wird höchstwahrscheinlich ein unnötiges Element sein, weil das Programm auf andere Weise feststellen kann, wie groß seine eigene Tabelle ist.

2.6.3 Schadensbegrenzung mit finally

Wir betrachten hier allgemein alle Funktionen, die die Exceptions nicht abschließend behandeln, sondern an die übergeordnete Funktion weitergeben (dazu gehört meistens auch die Funktion, die die Exception erzeugt hat). In vielen Fällen genügt es nämlich nicht, dass die Funktion einfach abgebrochen wird: Wenn diese in ihrem bisherigen Verlauf schon einige Ressourcen reserviert hat, sollte sie diese auch dann freigeben, wenn eine Exception auftritt. Ohne Exceptions hätte der Code dann beispielsweise so aussehen können:

```
Speicher := GetMem(10000); { Belegen der Ressource }
...
if not OpenFile(...) then begin { Fehlerquelle }
   FreeMem(Speicher);  { Notfreigabe der Ressource }
   exit;
end;
{ weitere Anweisungen }
FreeMem(Speicher, 10000);  { normale Freigabe }
```

Das Beispiel setzt voraus, dass *OpenFile* einen Fehlschlag beim Öffnen durch die Rückgabe des Wertes *False* anzeigt. Nachdem der obige Code festgestellt hat, dass ein Fehler aufgetreten ist, kann er selbst entscheiden, ob er trotzdem weiterarbeitet oder ob die Funktion mit *exit* verlassen werden soll. Im letzteren Fall muss natürlich vorher der Speicher wieder freigegeben werden.

Würde *OpenFile* das Scheitern ihrer Bemühungen jedoch nicht über das Funktionsergebnis, sondern mit einer Exception bekannt geben, müsste das Beispiel wie folgt umgeschrieben werden, um die Freigabe des Speichers weiterhin sicherzustellen:

```
Speicher := GetMem(10000);
...
try
  OpenFile()...
  { weitere Anweisungen }
finally
  FreeMem(Speicher, 10000)
end;
```

Die Anweisungen, die nach *OpenFile* und vor *finally* noch folgen, werden bei einer Exception immer übersprungen, hier hat das Programm keine Wahlmöglichkeit mehr wie im ersten Beispiel ohne Exceptions.

Der Ablauf einer *try...finally...end*-Konstruktion ist wie folgt: Tritt eine Exception innerhalb des von *try* und *finally* umgebenen Blocks auf, wird der Block sofort verlassen und die Anweisungen zwischen *finally* und *end* werden ausgeführt. Wenn keine Exception auftritt, läuft das Programm so ab, als wären die Wörter *try* und *finally* gar nicht anwesend: Zuerst wird der gesamte *try*-Block ausgeführt, dann der *finally*-Block.

Geschützte Ressourcen

Dadurch, dass die Ressourcen, die vor dem *try* reserviert wurden, auf jeden Fall freigegeben werden (falls Sie die entsprechenden Anweisungen in den *finally*-Teil geschrieben haben), erhält der *try*-Block die Bezeichnung *geschützter Block*. Wenn Sie innerhalb dieses Blocks weitere Ressourcen reservieren, die ebenso geschützt werden müssen, können Sie diese Kontrollstruktur auch schachteln:

```
AllocateRes1 { 1. Ressource }
try { Schutzblock für die 1. Ressource }
  ...  { Position (*) }
  AllocateRes2; { 2. Ressource }
  try { Schutzblock für 2. Ressource }
    ...
  finally
    FreeRes2; { Freigabe für Ressource 2 }
  end;
finally
  FreeRes1; { Freigabe der ersten Ressource }
end;
```

Tritt in diesem Fall im ersten, aber noch vor dem zweiten geschützten Block eine Exception auf (an der mit (*) markierten Position), so wird nur der *finally*-Block der äußeren Ebene ausgeführt, der innere *finally*-Block ist auch nicht nötig, weil die zweite Ressource noch gar nicht reserviert ist. *Finally*-Blöcke werden vom Programm also erst beim Eintritt in den zugehörigen *try*-Block für die unbedingte Ausführung vorgemerkt und dann auf jeden Fall ausgeführt – übrigens auch dann, wenn Sie den *try*-Block mit *exit* verlassen.

2.6.4 Behandeln der Exceptions

Mit den im letzten Abschnitt beschriebenen Aktionen ist der Ausnahmezustand noch nicht aufgehoben. Nach der Ausführung des *finally*-Blocks wird das Exception-Objekt so lange eine Aufrufebene höher gereicht, bis eine Funktion erreicht wird, die einen *except*-Block zur Verfügung stellt. Alle Funktionen, die das nicht tun, werden entweder ganz oder bis zu ihrem *finally*-Teil übersprungen.

Der *except*-Block (Exception-Handler) dient dazu, den Fehler so zu behandeln, dass das Programm danach wieder normal weiterlaufen kann. Wie schon bei *finally* müssen Sie zuerst mit dem *try* einen Programmabschnitt einleiten, der auf die Exceptions reagieren soll.

Im folgenden Beispiel wird die Exception praktisch an den Benutzer weitergegeben:

```
begin
  try
    OpenFile();
```

```
    ...
  except
    on EFileNotFound do
      MessageBox("Die Datei konnte nicht geöffnet werden.");
  end;
end;
```

Abfragen der Exception-Klasse

Wie das Beispiel zeigt, sind *except*-Blöcke differenzierter als *finally*-Blöcke, denn sie richten sich nach der Klasse der Exception und funktionieren ähnlich wie eine *case*-Anweisung. Hinter dem Schlüsselwort *on* geben Sie die Exception-Klasse an, die Sie behandeln möchten. Wenn die Exception diese oder eine davon abgeleitete Klasse hat, werden die Anweisungen nach dem *do* ausgeführt. Anschließend wird der *except*-Block beendet, auch wenn sich darin noch andere *do*-Klauseln befinden, die mit der Exception übereinstimmen. Das folgende Beispiel zeigt eine *do*-Klausel, die niemals ausgeführt wird, da sie eine Unterklasse einer Klasse abfragt, die schon vorher erledigt wurde:

```
except
  on EDatabaseError do ...
  on EDBEngineError do ...
end;
```

Die beiden *on*-Behandlungen müssen also vertauscht werden, um sinnvoll zu werden. Die zweite *on*-Bedingung trifft dann nur noch die *EDatabaseError*-Exceptions, die keine *EDBEngineError*-Exceptions sind.

Weitere Ähnlichkeiten mit der *case*-Anweisung sind: Mehrere Anweisungen werden in einem *begin...end*-Block zusammengefasst, Sie können mehrere Klassen hinter einem *on* angeben und einen *else*-Zweig für den Fall bereitstellen, dass eine andere Ausnahme auftritt. Letzteres ist jedoch meistens nicht zu empfehlen, da auch Exceptions, die Ihnen vielleicht nicht bekannt sind, auf diese Weise aufgehoben werden. Dadurch ist es einem äußeren Block nicht mehr möglich, die Exception auf die angemessene Weise zu behandeln.

Abfangen des Exception-Objekts

In den obigen Beispielen wurde nur die Klasse der Exception abgefragt. Sie können auch das Objekt selbst, das in der *raise*-Anweisung über einen Konstruktor initialisiert wurde, abfragen. Dazu müssen Sie ihm lediglich einen Namen geben, den Sie hinter *on* angeben, wie bei einer Variablendeklaration:

```
  except
    on E: ETableOverflow do begin
      { E.TableSize ist ansprechbar }
```

```
    end;
  end;
```

So können Sie auf die Datenelemente des Exception-Objekts zugreifen, z.B. *Message* für alle Exceptions und *TableSize* für den in einem früheren Beispiel deklarierten Typ *ETableOverflow*.

Es gibt eine weitere Möglichkeit, dieses Objekt zu erhalten, beispielsweise im *else*-Zweig des *except*-Blocks: Die Funktion *ExceptObject* (Unit *SysUtils*) liefert Ihnen das jeweils aktuelle Exception-Objekt als *TObject*-Referenz zurück (oder *nil*, wenn keine Exception vorhanden ist).

Erneuern und Schachteln von Exceptions

Wenn Sie in einem Exception-Handler erneut von der *raise*-Anweisung Gebrauch machen, kann das zwei verschiedene Wirkungen haben:

```
try
 ...
except
  on Exception do begin
    { Alle Ressourcen freigeben, die *nur bei einer Exception*
      nicht mehr gebraucht werden, die aber bei einer fehlerfreien
      Ausführung weiter belegt bleiben sollen. }
     ...
    raise; { Weitere Behandlung vom Aufrufer verlangen. }
  end;
end;
```

Falls Sie *raise* wie in diesem Beispiel ohne Parameter verwenden, erneuern Sie lediglich die Exception, die Sie gerade behandeln. Wenn Sie den Fehlerzustand in Ihrer Funktion nur teilweise beheben können, haben Sie so die Möglichkeit, diesen Teil im *except*-Block durchzuführen und dann die Exception noch einmal weiterzugeben. Das obige Beispiel wirkt also wie ein *finally*-Block, der nur im Fall einer Exception aufgerufen wird.

Ein selten vorkommender Spezialfall bleibt noch: Wenn Sie beim Aufruf von *raise* innerhalb eines *except*-Blocks ein neues Exception-Objekt konstruieren und mit *raise* starten, so ersetzt dieses das bisherige Exception-Objekt, falls Sie es nicht noch innerhalb desselben *except*-Blocks selbst behandeln.

Das Ende einer Exception

Eine Exception gilt als behandelt, sobald ein Exception-Handler gefunden wurde, der die Exception nicht erneuert. Ein solcher Exception-Handler kann drei verschiedene Erscheinungsformen haben:

- Eine *on*-Abfrage, die mit der aktuellen Exception übereinstimmt, beendet den Ausnahmezustand wie beschrieben.
- Der *else*-Teil einer *on*-Abfrage behandelt alle übrig bleibenden Exceptions.
- Ein *except*-Abschnitt kann statt einer *on*-Verzweigungsliste auch nur eine Reihe von Anweisungen enthalten. Durch diese wird der Ausnahmezustand in jedem Fall aufgehoben, so als würde der *except*-Abschnitt statt irgendwelcher *on*-Abfragen nur einen *else*-Teil enthalten.

Funktionen, die keine Exception-Behandlung durchführen, dienen den Exceptions, ohne es zu merken, als Durchgangsstation auf ihrem Weg nach oben. Falls sich keine Behandlung für eine Exception finden lässt, gelangt diese schließlich zur Laufzeitbibliothek zurück, die eine Meldung über die Exception anzeigt, die Bearbeitung der Nachricht, bei der diese Exception aufgetreten ist, abbricht und schließlich die Programmbearbeitung fortsetzt.

Selbst wenn Sie eine Exception nicht bearbeiten, kann Ihre Anwendung auf diese Weise meistens problemlos fortgesetzt werden, wenn Sie alle kritischen Programmteile mit *try..finally* behandelt haben. Wenn der Benutzer, z.B. über einen Schalter, das Ereignis ausgelöst hat, das zur Exception geführt hat, sollte ein sicheres Programm nach der Exception wieder in einem Zustand sein, als hätte der Benutzer den Schalter nie gedrückt. Da die VCL eine Meldung über die Exception anzeigt, kann der Benutzer sich etwas einfallen lassen, wie er den Fehler umgehen kann.

2.6.5 Optionen für Exceptions

Sowohl in den Projektoptionen als auch in den Umgebungseinstellungen finden Sie Optionen, die die Exceptions betreffen. Für das Programm wichtig sind die Compilerschalter der Projektoptionen (Abbildung 1.14). Mit diesen können Sie bestimmen, ob bestimmte Routineüberprüfungen im Fehlerfall in einer Exception resultieren oder nicht. Wenn die folgenden Optionen ausgeschaltet sind, werden die entsprechenden Fehler in Ihrem Programm nicht automatisch entdeckt, wodurch eventuell Folgefehler bis zum Programmabsturz entstehen können. Bei angeschalteter Option erzeugt der Compiler zusätzlichen Code, der bestimmte Bedingungen überprüft und eine Exception erzeugt, falls ein Fehler festgestellt wird. Sie können daher die Operationen, die zu Fehlern führen können, in *try*-Blöcken unterbringen:

- BEREICHSÜBERPRÜFUNG (*$R*): Überprüft, ob die Bereiche von Arrays, Strings und Unterbereichstypen eingehalten werden mit Ausnahme der Prozeduren *Inc* und *Dec*, die durch eine Bereichsprüfung um ein Vielfaches langsamer werden würden. Die Exception-Klasse ist *ERangeError*, abgeleitet von *EIntError*.

- ÜBERLAUFPRÜFUNG (*$Q*): Fängt Überläufe bei Integer-Operationen ab (wenn Sie beispielsweise mit den *Byte*-Variablen *a*, *b* und *c* die Anweisung *a:=b*c* ausführen und *b* und *c* den Wert 100 haben, so ist das Ergebnis zu groß für eine *Byte*-Variable), nicht überprüft werden die Prozeduren *Inc* und *Dec*. Auch hierfür gibt es eine von *EIntError* abgeleitete Exception-Klasse: *EIntOverflow*.

- I/O-PRÜFUNG (*$I*): Testet verschiedene I/O-Operationen auf Fehler, beispielsweise die Standardfunktionen für Dateioperationen wie *Reset*, *Rewrite*, *BlockWrite* etc. Für I/O-Fehler definiert die Laufzeitbibliothek die Klasse *EInOutError*, deren öffentliche *ErrorCode*-Variable weitere Informationen liefert.

  Auch wenn Sie diese Option abschalten, können Sie I/O-Fehler überprüfen, indem Sie die herkömmliche Funktion *IoResult* verwenden, für die hier auf die Online-Hilfe verwiesen sei.

Sie können die genannten Optionen auch für bestimmte Teile des Programms an- und abschalten, indem Sie die Compilerschalter im Programmcode setzen (siehe Kapitel 2.1.3). Auch die Überprüfung auf einen Stack-Überlauf lässt sich an- und abschalten, allerdings führt ein Überlauf bei angeschalteter Option zu einem sofortigen Programmabbruch ohne Exceptions.

Bereichs-, Überlauf- und Stack-Prüfung sollten übrigens nur in der Testphase eines Programms eingesetzt werden, da sie das Programm verlangsamen (und auch die ausführbare Datei vergrößern).

Debugger-Optionen

Unter den Umgebungsoptionen (TOOLS | DEBUGGER-OPTIONEN... | SPRACH-EXCEPTIONS) finden Sie auch die Option BEI DELPHI-EXCEPTIONS STOPPEN, die bewirkt, dass der Debugger das Programm stoppt, sobald eine Exception auftritt. Außerdem können Sie eine Liste von »Ausnahme-Exceptions« angeben, bei denen das Programm auch bei eingeschalteter Stop-Option *nicht* angehalten werden soll (ab Delphi 4).

Mit den Exceptions hängen auch die Haltepunkt-Aktionen zusammen, die das Verhalten des Debuggers bei Eintreten einer Exception automatisch und während des Programmlaufs beeinflussen können (siehe Kapitel 1.7.3).

2.6.6 Exceptions im Beispielprogramm

Als abschließendes Beispiel kommt dieser Abschnitt noch einmal auf die Exception-Behandlung des Beispielprogramms aus Kapitel 1.9 zurück. Eine potenzielle Fehlerquelle ist dort die Bibliotheksfunktion *StrToTime*, die es mit einer *EConvertError*-Exception ahndet, wenn eine Alarmzeit ungültige Zeichen enthält oder nicht den erlaubten Formaten (00:00:00 oder 00:00) entspricht.

Wir untersuchen dies anhand einer Programmversion, die die Alarmzeiten als String speichert und bei jedem Timer-Ereignis diesen String in der Funktion *IsTimeOver* mit *StrToTime* in einen Zeitwert umwandelt, um diesen dann mit der aktuellen Zeit zu vergleichen:

```
function TAlarmForm.IsTimeOver(Alarm : string) : boolean;
var
  AlarmTime : TDateTime;
begin
  Result:=False;
  if Alarm<>'' then begin
    AlarmTime:=StrToTime(Alarm);
    Result:=Time>AlarmTime;
  end;
end;
```

Für die Exception-Behandlung kommen nun zwei Stellen in Frage: in der gezeigten Funktion *IsTimeOver* und eine Aufrufebene höher in der Funktion *TimerTimer*, an die die Exception weitergereicht wird, falls *IsTimeOver* sie nicht behandelt:

```
procedure TAlarmForm.TimerTimer(Sender: TObject);
var
  FoundAlarm : string;
begin
  TimeText.Caption:=TimeToStr(Time);
  if AlarmAktiviert then
    if IsTimeOver(Alarmzeit) then
      // Ausgabe einer Meldung etc.
      ...
end;
```

Dazu muss zuerst überlegt werden, wie die Exception überhaupt behandelt werden soll. Eine Möglichkeit wäre, den Eingabefehler einfach zu ignorieren. Dazu müsste der Aufruf von *StrToTime* lediglich in einen *try*-Block eingeklammert werden, dessen *except*-Teil leer bleibt und alle Exceptions aufhebt. Dies würde in der Funktion *IsTimeOver* stattfinden:

```
function TAlarmForm.IsTimeOver(Alarm : string) : boolean;
var
  AlarmTime : TDateTime;
begin
  Result:=False;
  if Alarm<>'' then begin
    try
      AlarmTime := StrToTime(Alarm);
    except
      // die folgende Zeile kann auch weggelassen werden,
      // dann werden alle Exceptions ignoriert:
      on E: EConvertError do ; // nur EConvertError ignorieren
    end;
```

```
      Result:=Time>AlarmTime;
    end;
  end;
```

Statt dessen soll das Programm aber eine Fehlermeldung ausgeben und die Alarmfunktion abschalten, damit der Benutzer den Fehler korrigieren kann. Diese Tätigkeit liegt nicht im Zuständigkeitsbereich der Methode *IsTimeOver*, denn diese soll lediglich entscheiden, ob die Weckzeit vorüber ist, und keine weiteren Nebeneffekte haben.

Daher behandelt *IsTimeOver* die Exception nicht; die Exception wird also eine Ebene höher zu *TimerTimer* gereicht, die sie mit dem in Kapitel 1.8.5 gezeigten Exception-Block behandelt:

```
procedure TAlarmForm.TimerTimer(Sender: TObject);
begin
  TimeText.Caption := TimeToStr(Time);
  try
    ...
     if IsTimeOver(AlarmEdit.Text) then
    ...
  except
    on E: EConvertError do begin
      AlarmActive.Checked := False;
      MessageDlg('Ungültige Eingabe, bitte korrigieren:'+sLineBreak
                 + E.Message, mtError, [mbOK], 0);
    end;
  end;
end;
```

Ein *finally*-Block ist hier nicht notwendig, da *TimerTimer* keine Ressourcen anfordert, insbesondere nicht vor dem Aufruf der Funktion, aus der eine Exception »herauskommen kann«, der Funktion *IsTimeOver*.

2.7 Interfaces

Dieses Kapitel widmet sich den Klassen-*Schnittstellen*, die kurz als *Interfaces* (Schnittstellen) bezeichnet werden, nicht zu verwechseln mit den Unit-Schnittstellen, die in Object Pascal ebenfalls durch das Schlüsselwort *interface* eingeleitet werden. In diesem Kapitel bezieht sich die Bezeichnung *Interface* immer auf ein Klasseninterface.

Für den Einstieg können Sie sich ein solches Interface wie eine abstrakte Klasse vorstellen, also eine Klasse, die nur abstrakte Methoden und keine Datenelemente enthält. Die aus diesen Methoden bestehende Schnittstelle kann nun an beliebige (normale) Klassen vererbt werden – zusätzlich zur einfachen Vererbung von Object Pascal.

Historische Anmerkung

Obwohl sich Interfaces hervorragend für allgemeine Zwecke verwenden lassen und eigentlich jeder objektorientierten Programmiersprache gut zu Gesicht stünden (Java gibt ein weiteres Beispiel dafür), diente die Einführung dieser Interfaces in die Sprache Object Pascal ursprünglich nicht in erster Linie dazu, ein besseres Software-Engineering zu ermöglichen. Anlass war vielmehr das Ziel von Borland, in Delphi 3 eine vollständige Unterstützung des Component Object Models (COM) von Microsoft Windows zu bieten. Hierbei ging es darum, durch die Interfaces die Kommunikation zwischen verschiedenen Anwendungen zu ermöglichen. Die Verwendung von Interfaces für die Kommunikation *zwischen* Anwendungen wird in Kapitel 8.5 näher erläutert werden. Ein kleines praktisches Beispiel zur Verwendung eines Interfaces finden Sie im TreeDesigner (siehe Kapitel 5.7.5).

2.7.1 Abstrakte Basisklassen versus Interfaces

Um den Unterschied der Interfaces zur bisherigen Art der objektorientierten Programmierung in Object Pascal zu verdeutlichen, kommen wir noch einmal auf das Beispiel aus Kapitel 2.3.3 zurück, in dem der Benutzer aus mehreren Klassen, die bestimmte gemeinsame Fähigkeiten haben, eine Klasse auswählen soll, von der das Programm dann zur Laufzeit ein neues Objekt erzeugt.

Schnittstellen-Vererbung ohne Interfaces

Der Einfachheit halber verzichten wir diesmal wieder auf die in Kapitel 2.3.4 zu Hilfe genommenen Klassenreferenzen und erzeugen das Objekt mit einem normalen Aufruf des Konstruktors der gewählten Klasse:

```
var
  Objekt: AbstrakteKlasse;
begin
  case Benutzerauswahl of
    { "Benutzerauswahl" kann z.B. sein: (Sender as TButton).Tag }
     Button1: Objekt := Klasse1.Create;
     Button2: Objekt := Klasse2.Create;
  ...
     Button10: Objekt := Klasse10.Create;
```

Die gemeinsamen Fähigkeiten der verschiedenen Klassen waren in diesem Beispiel auf eine virtuelle Methode *Arbeite* beschränkt, die in einer abstrakten Basisklasse deklariert wurde:

```
type
  AbstrakteKlasse = class
  { Konstruktor Create wird von TObject geerbt }
    procedure Arbeite; virtual;
  end;
```

Nach der Initialisierung von *Objekt* in der obigen *case*-Anweisung konnten wir dieses *Objekt* mit *Objekt.Arbeite* dazu bringen, seine ganz spezielle Version der Methode *Arbeite* auszuführen, wobei zum Zeitpunkt der Kompilierung nicht feststand, ob dies nun die Methode *Klasse1.Arbeite*, *Klasse2.Arbeite* oder noch eine andere Methode sein würde.

In diesem Beispiel haben alle Klassen eine Schnittstelle gemeinsam, die Schnittstelle der Klasse *AbstrakteKlasse*, die aus der Methode *Arbeite* besteht (und diese »Schnittstelle« ist noch kein Interface im Sinne von Delphi!). Voraussetzung für das Funktionieren dieses Beispiels war allerdings, dass alle Klassen, die dem *Objekt* zugewiesen werden konnten, von einer gemeinsamen Basisklasse abgeleitet worden sind, in diesem Beispiel der Klasse *AbstrakteKlasse*.

Was ist ein Interface?

Ein Interface ist genau das, was wir im letzten Beispiel als Schnittstelle der Klasse *AbstrakteKlasse* bezeichnet haben: eine wohldefinierte Menge von Methoden. Um die Schnittstelle von *AbstrakteKlasse* nun als echtes Delphi-Interface zu deklarieren, können wir die Deklaration wie folgt umschreiben (wobei wir dem Interface einen anderen Namen geben, der mit dem für Interfaces üblichen *I* beginnt):

```
type
  IArbeite = interface
    procedure Arbeite;
    // hier können weitere Methoden folgen
  end;
```

Allerdings besteht ein Interface *nur* aus solchen Methodendeklarationen. Damit ist ein Interface »weniger« als eine Klasse, enthält es doch

- keine Variablen und Properties
- keinen Code. Die Methodendeklarationen sind damit sozusagen alle »abstrakt«, nur dass Sie hier nicht das Schlüsselwort *abstract* verwenden müssen wie bei Klassendeklarationen.

Mit den Interfaces von Object Pascal ergeben sich nun die folgenden neuen Möglichkeiten:

- Klassen mit einer gemeinsamen Schnittstelle müssen nicht mehr von einer gemeinsamen Basisklasse abstammen, damit sie ein und demselben (*polymorphen*) Objekt zugewiesen werden können (abgesehen von der Tatsache, dass in Object Pascal alle Klassen von *TObject* abstammen).
- Klassen können mehrere völlig verschiedene Interfaces unterstützen.

Um das obige Beispiel zu vervollständigen: Mit der Schnittstelle *IArbeite* ist es nun möglich, einer Variablen den Typ *IArbeite* zu geben (anstatt *AbstrakteKlasse*):

```
var
  ArbeitsfaehigesObjekt: IArbeite;
begin
  case Benutzerauswahl of
    { "Benutzerauswahl" kann z.B. sein: (Sender as TButton).Tag }
     Button1: ArbeitsfaehigesObjekt := Klasse1.Create;
  ...
     Button100: ArbeitsfaehigesObjekt := Klasse100.Create;
```

Der Variablen *ArbeitsfaehigesObjekt* können Sie beliebige Objekte mit einer Klasse, die die *IArbeite*-Schnittstelle unterstützt, zuweisen. Von welchen Klassen diese Klassen abgeleitet sind, spielt zumindest theoretisch keine Rolle. *Klasse1* kann zum Beispiel eine Erweiterung der vordefinierten Klasse *TStringList* sein und *Klasse100* könnte ein externer COM-Server sein, der sich in einer ganz anderen Anwendung befindet, die zufällig auch die *IArbeite*-Schnittstelle unterstützt.

In Kapitel 2.7.4 werden wir zu der Frage kommen, wie ein Interface im Unterschied zu einer Klasse überhaupt implementiert wird. Vorher werden wir mit einem kleinen Beispielprogramm beginnen (Kapitel 2.7.2) und ein paar wichtige allgemeine Dinge klären (Kapitel 2.7.3).

2.7.2 Verwendung eines Interfaces

Ein zweites Beispiel für ein Interface zeigt das folgende Listing, es stammt aus dem Beispielprogramm *InterfaceDemo*, das Sie auf der CD finden:

```
type
  IContainer = interface
    procedure AddElement(const ElementName: string);
    procedure DeleteElement(const ElementName: string);
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;
```

Dieses Interface beschreibt eine Schnittstelle, über die Sie einem Containerobjekt Elemente in Form von Strings hinzufügen können (*AddElement*), die Sie später auch wieder löschen können *(DeleteElement*). Mit den letzten beiden Methoden können Sie die Zahl der Elemente sowie den Namen des ersten Elements abfragen. Diese Schnittstelle sagt noch nichts über den Container aus, außer dass er Teile besitzt, die wie in der gezeigten Schnittstelle als Strings angesprochen werden können.

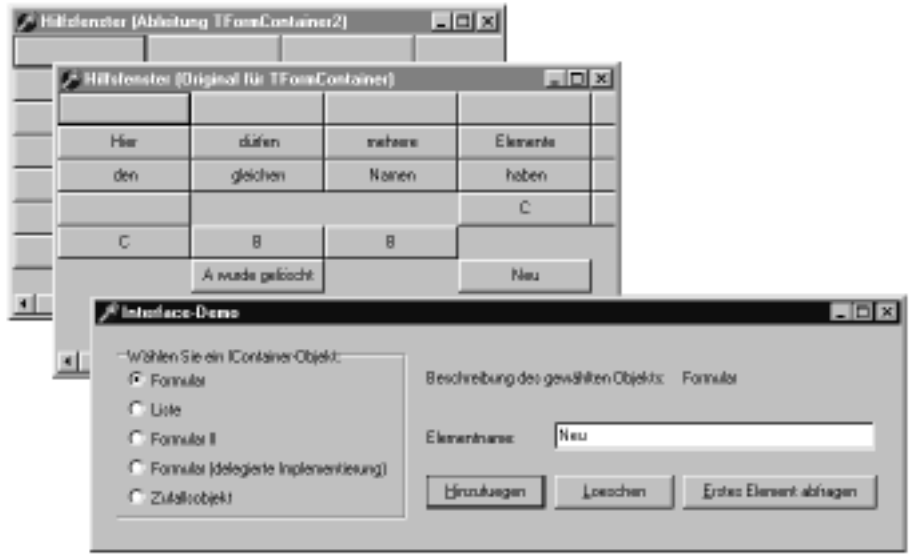


Abbildung 2.1: Die Programmoberfläche für die Verwendung mehrerer völlig verschiedener Objekte, die dasselbe Interface unterstützen (Beispielprogramm InterfaceDemo).

Abbildung 2.1 zeigt die Oberfläche des Beispielprogramms. Zur Laufzeit wählen Sie über die Radioschaltergruppe des Hauptfensters einen von fünf Containern aus, zu dem Sie dann über zwei Buttons das im Editierfeld angegebene Element hinzufügen oder löschen können:

- Die Elemente des Formular-Containers sind Schalter. Jedes »Element« , das mit *AddElement* hinzugefügt wird, wird in einem zweiten Formular als Schalter dargestellt. Entsprechend wird ein Schalter gelöscht, wenn die *DeleteElement*-Methode aufgerufen wird. Dieses an sich sinnlose Beispiel wurde gewählt, um einen möglichst großen Kontrast zum zweiten Containerobjekt zu geben. Zur Implementation dieses Containers kommen wir in Kapitel 2.7.4.

- Dieses zweite Containerobjekt, die »Liste« , merkt sich die mit *AddElement* hinzugefügten Objekte einfach nur, ohne sie irgendwo am Bildschirm anzuzeigen. Als »Beweis« , dass sich die Liste richtig verhält, können Sie ja die einzelnen Elemente mit *GetFirstElementName* abfragen und wieder löschen. Die Interna dieses Objekts werden Sie in Kapitel 2.7.5 kennen lernen.

- Das Containerobjekt *Formular II* verhält sich nach außen hin genauso wie das erste Objekt, ist intern ähnlich wie der Listen-Container realisiert (dazu mehr in Kapitel 2.7.5).

- Das Objekt »Formular (delegierte Implementierung)« verhält sich nach außen hin genauso wie *Formular II*, ist aber intern anders realisiert. Es verwendet die Delegationstechnik, die ebenfalls in Kapitel 2.7.5 erläutert wird.
- Für die ersten vier Containerobjekte definiert das Programm jeweils eine eigene Klasse. Mit *Zufallsobjekt* wählen Sie ein fünftes Objekt aus, das zu Programmbeginn von einer zufällig aus diesen vier Klassen ausgewählten Klasse erzeugt wird.

Nun zum Programm: Die Hauptformular-Unit greift je nach Selektion der Radioschaltergruppe auf eines der folgenden Array-Elemente zu:

```
var
  ContainerArray: array[0..4] of IContainer;
```

Dieses Array ist in einer zweiten Unit namens *Containers* deklariert, in der sich auch die eigentlichen Containerobjekte befinden. Worum es sich bei diesen Objekten handelt, lassen wir weiter offen, denn das ist eine Privatsache der Unit *Containers* (bzw. ihres Implementationsteils). Die Methoden des Formulars kennen nur die anfangs abgedruckte Schnittstellendefinition und dieses vierelementige Array, denn das genügt bereits, um die Methoden für das Hauptformular zu implementieren. Die Methoden für die beiden Schalter sehen beispielsweise wie folgt aus:

```
// ContainerTyp: TRadioGroup; (zwei Radioschalter, siehe Abbildung)
// ElementName: TEdit; (Editierfeld, siehe Abbildung)
// ContainerArray: siehe oben

procedure TForm1.HinzufuegenClick(Sender: TObject);
begin
  ContainerArray[ContainerTyp.ItemIndex].AddElement(ElementName.Text);
end;

procedure TForm1.LoeschenClick(Sender: TObject);
begin
  ContainerArray[ContainerTyp.ItemIndex].DeleteElement(ElementName.Text);
end;
```

Zur Zeit der Kompilierung steht also noch nicht fest, welche Art von Element in diesen Zeilen angesprochen wird. *ContainerArray[0..4]* sind *polymorphe Objekte*, wie wir sie auch bei Verwendung von Klassen schon erhalten haben. Ohne die Verwendung von Interfaces hätte *ContainerArray* auch wie folgt deklariert sein können:

```
var
  ContainerArray: array[0..4] of TContainer;
```

wobei *TContainer* eine Klasse sein könnte. Bei der zu Beginn gezeigten Interface-Deklaration hätten nur das Wort *interface* durch *class* ersetzt und die Methodendeklarationen durch ein *abstract* ergänzt werden müssen, und *ContainerArray* hätte genauso verwendet werden können, wie oben gezeigt.

Der Unterschied zwischen Klasse und Interface ergibt sich hier erst bei der Implementierung, für die wir uns im übernächsten Abschnitt dem Implementationsteil der Unit *Containers* zuwenden werden.

2.7.3 IUnknown, Co-Klassen und andere Begriffe

Während die Verwendung eines existierenden Interfaces so einfach ist, wie im letzten Abschnitt gezeigt, müssen Sie bei der Implementierung von Interface-Klassen und -Objekten einige weitere Begriffe kennen. Da die Interfaces in Object Pascal dazu gedacht sind, Microsofts Component Object Model (COM), den Standard für objektorientierte Windows-Schnittstellen, in Delphi zu repräsentieren, werden im Folgenden auch die entsprechenden COM-Begriffe erwähnt:

- *Interface-Vererbung*: Wie eine Klasse die Elemente einer anderen Klasse erben kann, kann ein Interface die Methoden eines anderen Interfaces erben. Die Syntax für die Vererbung stimmt mit der Vererbungssyntax bei Klassen überein, das Erb-Interface wird also in Klammern hinter dem Schlüsselwort *interface* angegeben.
- *IInterface*: So wie *TObject* der obligatorische Vorfahre aller Klassen ist, gibt es auch für die Interfaces ein Interface, das immer geerbt wird, auch wenn Sie kein Erb-Interface explizit angeben. Dieses Interface heißt *IInterface*.
- *IUnknown* ist ein Alias-Name für *IInterface*. Bis Delphi 5 gab es unter Windows sogar nur den Namen *IUnknown*, da *IInterface* genau dem grundlegenden *IUnknown*-Interface aus dem COM von Microsoft entspricht. Diese Namensgleichheit hat immer wieder zu dem Fehlurteil geführt, dass Delphis Interfaces vom COM abhängig seien. Kylix beweist das Gegenteil, da seine Interfaces unter Linux auch ohne COM sehr gut funktionieren. Die Namensänderung von *IUnknown* in *IInterface* unterstreicht diese Unabhängigkeit.
- Eine *implementierende Klasse* zu einem Interface ist eine Klasse, die die im Interface deklarierten abstrakten Methoden *implementiert*. Object-Pascal-Klassen können gleichzeitig die Methoden von mehreren Interfaces implementieren. Im COM werden solche implementierenden Klassen auch als *Co-Klassen* bezeichnet.
- Wenn Sie von einer Klasse, welche eines oder mehrere Interfaces implementiert, ein Objekt erzeugen (in Abschnitt 2.7.4 werden wir dies tun), dann erhalten Sie ein Objekt, welches Interfaces unterstützt. Unter Windows wird dieses Objekt dann ein *COM-Objekt* genannt (mit anderen Worten: Ein COM-Objekt ist die Instanz einer Co-Klasse). Die über Interfaces verfügenden Objekte einer Kylix-Anwendung unter Linux arbeiten genauso wie Delphis COM-Objekte unter Windows, nur dass sie ihre Funktionalität nicht nach außen, zu anderen Anwendungen oder anderen Rechnern exportieren können, denn hierfür ist ja unter Windows Microsofts (D)COM zuständig.

- Eine *Interface-Variable* (oben z.B. jedes Element von *ContainerArray*) entspricht in Object Pascal einer Zugriffsmöglichkeit für die in einem bestimmten Interface definierten Methoden eines bestimmten (COM-)Objekts. Mit der Interface-Variablen *ContainerArray[0]* können Sie beispielsweise alle *IContainer*-Methoden des ersten Objekts aufrufen, mit *ContainerArray[1]* die *IContainer*-Methoden des zweiten Objekts. Um Methoden aufzurufen, die zum selben Objekt gehören, aber in einem anderen Interface deklariert sind, benötigen Sie eine weitere Interface-Variable.

Es stellt sich nun die Frage, wie Sie an eine solche weitere Interface-Variable für ein anderes Interface desselben COM-Objekts kommen. Die Antwort dazu finden wir bei einer genaueren Untersuchung des *IInterface*-Interfaces in einer von dessen Methoden.

IInterface und IUnknown

IInterface besitzt drei Methoden, die aufgrund der automatischen Vererbung von *IInterface* in *jeder* Interface-Variablen zur Verfügung stehen. Da ist zunächst die oben bereits angekündigte Methode, mit der sich andere Interfaces desselben Objekts ermitteln lassen:

- *QueryInterface* liefert eine Interface-Variable für eine andere Schnittstelle desselben Objekts zurück. Wenn Sie beispielsweise in *Container: IContainer* die Schnittstelle eines Containerobjekts gespeichert haben und dieses Objekt auch noch eine zweite Schnittstelle unterstützt (beispielsweise *IClassDescription*), können Sie diese Schnittstelle mit *QueryInterface* anfordern: *Window := Container.QueryInterface(IClassDescription)*. Besserer Object-Pascal-Stil ist es jedoch, statt *QueryInterface* den Object-Pascal-Operator *as* zu verwenden, der intern auf *QueryInterface* zurückgreift: *Window:=Container as IClassDescription*.

Voraussetzung für die Funktion von *QueryInterface* ist, dass es für das gesuchte Interface eine Kennzahl (Interface Identifier, IID) gibt. Die Angabe einer IID findet am Anfang der Deklaration eines Interfaces in eckigen Klammern wie im folgenden Beispiel statt:

```
type
  IClassDescription = interface
    ['{22910AA3-C028-11D0-9E2F-444553540000}']
    function GetClassDescription: string;
  end;
```

Fragt sich, wie man sich ein solches Ungetüm von Kennzahl ausdenken soll. Glücklicherweise kann man hier das Denken einmal an den Rechner delegieren und drückt im Editor der Delphi-IDE einfach Shift+Strg+G (für *GUID*). Der Editor fügt dann einen komplett neuen GUID mitsamt innerer Klammerung, Hochkommata und äußerer Klammerung ein. Der GUID wird nach den Regeln des COM generiert, wonach die aktuelle Zeit und technische Daten des gerade verwendeten Computers in die Zahl einfließen. (Dies soll lediglich sicherstellen, dass der GUID weltweit eindeutig ist, es dient nicht etwa dazu, Daten des verwendeten Computers für Spionagezwecke in den GUIDs zu verschlüsseln).

Referenzzählung

Die anderen beiden Methoden von *IInterface* werden in Object Pascal immer automatisch aufgerufen. Für die interne Funktionsweise der Interfaces sind sie jedoch so wichtig, dass Sie sie kennen sollten:

- *_AddRef* benachrichtigt ein (COM-)Objekt darüber, dass eines seiner Interfaces verwendet wird. Das Objekt erhöht daraufhin einen internen Referenzzähler, der angibt, wie oft das gesamte Objekt zurzeit verwendet wird.
- *_Release* ist das Gegenstück von *_AddRef*, es sagt dem Objekt, dass es seinen Referenzzähler um eins verringern kann. Ist der Zähler bei 0 angekommen, kann sich das Objekt selbst freigeben. Eine Anwendung ruft *_Release* auf, wenn sie ein Objekt nicht mehr benötigt, beispielsweise am Ende einer Prozedur, in der das Objekt als lokale Variable verwendet wurde.

Die Referenzzählung ist im COM deshalb so wichtig, weil COM-Objekte mehrfach in völlig verschiedenen Anwendungen verwendet werden können. Die einzelnen Anwendungen dürfen nicht alleine dafür verantwortlich sein, einmal reservierte COM-Objekte freizugeben, denn sie könnten sonst aus Versehen ein Objekt freigeben, das in einer anderen Anwendung noch verwendet wird, wodurch diese andere Anwendung möglicherweise abstürzen würde. Statt dessen ist jede Anwendung lediglich dafür verantwortlich, den Referenzzähler seiner Objekte richtig »zu bedienen« , so dass diese sich von selbst freigeben können, falls notwendig.

Wenn Sie nun Objekte mit Interfaces lediglich innerhalb einer Delphi-Anwendung einsetzen, so können Sie dort dieselben Vorteile nutzen. Wie die verschiedenen COM-Anwendungen können auch verschiedene Objekte innerhalb derselben Anwendung ein Objekt mit Interfaces verwenden, ohne sich um die Freigabe kümmern zu müssen. Der vom Delphi-Compiler erzeugte Code wird automatisch dafür sorgen, dass das Objekt freigegeben wird, sobald der letzte Nutzer des Objekts *_Release* aufruft. Wobei ja wie gesagt auch dieser Aufruf von *_Release* automatisch stattfindet (und zwar am Ende einer Methode, wenn das Objekt eine lokale Variable der Methode ist, oder bei der Freigabe eines Objekts, wenn es sich um die Variable eines Objekts handelt).

Wie unschwer zu erkennen ist, ist eine irrtümliche Freigabe eines COM-Objekts auch mit *_AddRef* und *_Release* nicht ausgeschlossen, denn niemand hält eine Anwendung davon ab, die *_Release*-Methode öfter aufzurufen, als ihr zusteht. Auch innerhalb einer Delphi-Anwendung können Sie ja *_Release* noch manuell aufrufen und dadurch Fehler verursachen. Daher ist der vollkommen automatische Aufruf von *_AddRef* und *_Release* in Object Pascal ein sehr großer Vorteil. Zu einer Demonstration des Referenzzählers und dessen automatischer Verwaltung siehe auch das Beispielprogramm mit dem selbst definierten Interface in Kapitel 8.5.3.

2.7.4 Implementierung eines Interfaces

Nach dieser umfangreichen Theorie soll nun endlich die Frage beantwortet werden, wie Sie in Object Pascal eine Interface-Variable erzeugen. Das folgende Listing zeigt zum Vergleich noch einmal, wie einfach es ist, von einer Klasse zu einem Objekt zu kommen:

```
type TKlasse = class ... end;
var Objekt: TKlasse;
begin
  Objekt := TKlasse.Create;
```

Bei einem Interface ist das nicht so einfach, *IContainer.Create* gibt es also nicht. Zwischen dem Interface und der Interface-Variablen stehen als weitere Stationen noch die implementierende Klasse (Co-Klasse) und das Erzeugen einer Instanz dieser Klasse.

TInterfacedObject als Basis für die implementierende Klasse *R131*

Das in Kapitel 2.7.2 bereits eingeführte Beispielprogramm aus Abbildung 2.1 benötigt für jedes der verschiedenen *IContainer*-Objekte eine eigene Co-Klasse, die die *IContainer*-Schnittstelle implementiert. Das folgende Listing zeigt eine der beiden Klassen – die Klasse für das *IContainer*-Formular:

```
type
  TFormContainer = class(TInterfacedObject, IContainer)
    // FormContainer erzeugt das Formular selbst, sobald es benötigt wird.
    // Welche Formularklasse es verwenden soll, wird im Konstruktor
    // angegeben.
    HostFormClass: TFormClass; // Klasse des Formulars (siehe Konstruktor)
    HostForm: TForm; // Das dynamisch erzeugte Formular, in das die
                     // "Elemente" eingefügt werden.
    AddCount: Integer; { zählt die Zahl der Aufrufe von AddElement }
  public
    constructor Create(AFormClass: TFormClass);
    procedure AddElement(const ElementName: string);
    procedure DeleteElement(const ElementName: string);
```

```
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;
```

Die erste Zeile dieser Deklaration ist bereits die wichtigste: *IContainer* in den Klammern hinter dem *class*-Schlüsselwort bedeutet, dass diese Klasse die *IContainer*-Schnittstelle implementieren wird. Die Angabe von *TInterfacedObject* bewirkt, dass *TFormContainer* von der in Delphi vordefinierten Klasse *TInterfacedObject* abgeleitet wird. Sinn von *TInterfacedObject* ist es, eine Implementation der drei obligatorischen *IInterface*-Methoden bereitzustellen. Wenn Sie Ihre Implementationsklasse also von *TInterfacedObject* ableiten, brauchen Sie keinen Referenzzähler zu deklarieren und keine eigenen *QueryInterface*-, *_AddRef*- und *_Release*-Methoden zu schreiben.

TInterfacedObject ist nur eine von mehreren alternativen Klassen, von denen Co-Klassen abgeleitet werden können. Den anderen Klassen – *TComObject*, *TTypedComObject*, *TAutoObject* und *TActiveXControl* – werden wir in den Kapiteln 6.8, 8.6 und 8.7 noch begegnen. Außerdem wird Kapitel 2.7.5 zeigen, dass Sie auch Co-Klassen erzeugen können, die weder von *TInterfacedObject* noch von einer anderen der vorgegebenen Klassen abgeleitet sind.

Kommen wir nun zum »Inhalt« der Klasse aus dem obigen Listing. Außer dem eigenen Konstruktor, der nur dazu dient, den von außen angelieferten Parameter in der Objekt-internen Variablen festzuhalten, sind in der Klasse *TFormClass* nur die Methoden des *IContainer*-Interfaces deklariert. Zur Veranschaulichung sei hier die Methode *AddElement* gezeigt:

```
procedure TFormContainer.AddElement(const ElementName: string);
var
  B: TButton;
begin
  if not Assigned(HostForm) then HostForm := HostFormClass.Create(nil);
  B := TButton.Create(HostForm);
  B.Parent := HostForm;
  // Die Schalter so positionieren, dass kein Schalter verdeckt wird
  // (Lücken, die in DeleteElement entstehen, werden nicht gefüllt):
  B.Width := 100;
  B.Top := (AddCount div 5) * B.Height;
  B.Left := (AddCount mod 5) * B.Width;
  B.Caption := ElementName;
  inc(AddCount);
end;
```

Wie der Code dieser Methode hat auch der Code der anderen Methoden von *TFormClass* nichts mit unserem Thema der Interfaces zu tun, daher können wir die Implementierung der Klasse *TFormClass* hier als gegeben betrachten.

Hinweis: Seit Delphi 6 hat die Code-Vervollständigungsfunktion auch innerhalb von Klassendeklarationen eine sinnvolle Anwendung gefunden: Wenn Sie in einer Klasse Interfaces implementieren wollen, so können Sie in der noch leeren Klassendeklaration Strg + Leer drücken und daraufhin aus der aufgeklappten Liste die Interface-Methoden auswählen, die implementiert werden sollen. Delphi fügt dann die Methodendeklarationen so in die Klasse ein, wie sie vom Interface vorgegeben wurden. Als kleiner Hinweis, dass eine Klasse immer *alle* Interface-Methoden implementieren muss, sind diese Methoden in der Liste auch noch rot hervorgehoben.

Erzeugung von Objekten mit Interfaces

Wie schon erwähnt ist ein COM-Objekt eine Instanz einer Co-Klasse. Um nun ein Objekt der Klasse *TFormContainer* zu erzeugen, müssen wir nur auf das Wissen über Klassen in Object Pascal zurückgreifen:

```
var
  FormContainer: TFormContainer;
begin
  FormContainer := TFormContainer.Create(THelperForm);
```

Wenn Sie sich an Kapitel 2.7.2 erinnern, haben wir dort jedoch nicht mit einer Variablen vom Typ *TFormContainer* gearbeitet, sondern mit einer Variablen des Typs *IContainer*, welche auch das Ziel des aktuellen Abschnitts sein soll. Das COM-Objekt *FormContainer* ist nur ein Zwischenschritt auf dem Weg zu diesem Ziel.

Die Interface-Variablen

Die folgenden Programmzeilen deklarieren eine Interface-Variable *ContainerInterface* und weisen ihr das oben erzeugte Objekt zu:

```
var
  ContainerInterface: IContainer;
begin
  ContainerInterface := FormContainer;
```

Da Sie über die Interface-Variable nicht mehr auf alle theoretisch möglichen Methoden von *FormContainer* zugreifen können, entspricht diese Zuweisung einer Typumwandlung von einer abgeleiteten in eine höhere Klasse und *ContainerInterface* entspricht einem polymorphen Objekt, dessen wahrer Typ nicht bekannt ist.

Diese Polymorphie wird in der Beispiel-Unit *Containers* deutlich, in der das schon in Kapitel 2.7.2 verwendete Array wie folgt initialisiert wird, wobei auch die in den folgenden Abschnitten besprochenen implementierenden Klassen zum Einsatz kommen:

```
var
  Zufall: byte;
initialization
  ContainerArray[0] := TFormContainer.Create(THelperForm);
  ContainerArray[1] := TListContainer.Create;
  ContainerArray[2] := TFormContainer2.Create(Screen);
  ContainerArray[3] := TFormContainerWrapper.Create(Screen);
  Randomize; Zufall := Random(4); // Zufällige Auswahl aus 4 Elementen
  case Zufall of
    0: ContainerArray[4] := TFormContainer.Create(THelperForm);
    1: ContainerArray[4] := TListContainer.Create;
    2: ContainerArray[4] := TFormContainerWrapper.Create(Screen);
    3: ContainerArray[4] := TFormContainer2.Create(Screen);
  end;
end.
```

Für den Programmcode in Kapitel 2.7.2 ist es völlig unerheblich, welche Klassen auf der rechten Seite der Zuweisung genannt werden. Der Compiler achtet lediglich darauf, dass Sie eine Klasse angeben, die die *IContainer*-Schnittstelle implementiert. Das Programm funktioniert also auch noch, wenn Sie die Objekte vertauschen oder alle vier Objekte von zufällig ausgewählten Klassen erzeugen lassen.

Delegierte Implementierung

Eine besondere Technik, mit der Interfaces implementiert werden können, ist die Delegation (ab Delphi 4). Eine Klasse muss die unterstützten Interfaces nicht mehr durch eigene Methoden implementieren, sondern sie darf die Implementation an ein anderes Objekt bzw. andere Objekte delegieren. Im Beispielprogramm wird dies von der folgenden Klasse demonstriert:

```
TFormContainerWrapper = class(TObject, IContainer)
private
  FContainer: TFormContainer2;
public
  property Container: TFormContainer2
                        read FContainer implements IContainer;
  constructor Create; override;
end;
```

Auch *TFormContainerWrapper* soll das *IContainer*-Interface unterstützen. Sie soll aber dessen Methoden nicht selbst implementieren, sondern dies an eine andere *IContainer*-Klasse delegieren, und zwar an die Beispielklasse *TFormContainer2* aus dem nächsten Abschnitt (natürlich könnte ebenso gut auch die bereits bekannte *TListContainer*-Klasse verwendet werden). Ein Objekt dieser Klasse wird in *FContainer* gespeicher. *FContainer* wird im *Create*-Konstruktor von *TFormContainerWrapper* konstruiert:

```
constructor TFormContainerWrapper.Create;
begin
```

```
    FContainer := TFormContainer2.Create(Application);
  end;
```

Da *FContainer* alle *IContainer*-Methoden unterstützt, kann *TFormContainerWrapper* seine *IContainer*-Verpflichtungen an *FContainer* delegieren. Das Property *Container* dient lediglich dazu, diese Delegation dem Compiler mitzuteilen. Entscheidend ist hierbei das Ende der Property-Deklaration: *implements IContainer* besagt, dass das in diesem Property gespeicherte Objekt die *IContainer*-Schnittstelle implementiert.

Das Property braucht hierfür lediglich lesbar zu sein, theoretisch wäre es auch möglich, dass das zurückgelieferte Objekt (hier also *FContainer*) mehrere Interfaces implementiert, die dann als kommaseparierte Liste hinter *implements* angegeben werden. Eine letzte Variation bestünde darin, statt an ein Objekt wie *FContainer* direkt an eine *IContainer*-Variable zu delegieren.

Das obige Beispiel ist sicher aufgrund seiner geringen Komplexität noch nicht geeignet, die Vorteile der Delegation besonders herauszuheben. Die Beispiel-Klasse *TFormContainerWrapper* kommt ja aufgrund der Delegation mit nur einer einzigen Methode aus – dem oben gezeigten Konstruktor – und wird dadurch zu einer leeren Hülle. Hätten wir es anstatt mit *TFormContainerWrapper* mit einer komplexeren Klasse zu tun, die auch noch andere Aufgaben zu erfüllen hat, so könnte die Delegation eine nützliche Entlastung darstellen.

Ein weiterer Vorteil der Delegation, auf den hier nur am Rande hingewiesen werden soll, besteht darin, dass Sie die Implementierung eines Interfaces zur Laufzeit auswechseln können. Wenn Sie im obigen Beispiel etwa zur Laufzeit ein anderes Objekt an *FContainer* zuweisen, vielleicht sogar eines mit einer ganz anderen Klasse (gefordert ist ja lediglich, dass diese auch die *IContainer*-Schnittstelle unterstützt), dann wechseln Sie quasi einen Teil der Methoden von *TFormContainerWrapper* zur Laufzeit aus.

Hinweis: Würde *TFormContainerWrapper* noch weitere Interfaces unterstützen, so könnten diese zwar theoretisch sehr schön an verschiedene Objekte delegiert werden, allerdings versteckt sich hinter dieser Idee ein größerer Nachteil: Jedes dieser Objekte würde über die Vererbung von Interfaces auch die *IInterface*-Schnittstelle unterstützen und damit eine *QueryInterface*-Methode besitzen. Diese Methode würde aber standardmäßig immer nur die anderen Interfaces desselben implementierenden Objekts zurückliefern, aber kein Interface der anderen Objekte.

2.7.5 Interface-Mix-In

Wir kommen nun zum zweiten Container-Objekt des Beispielprogramms: Dieses soll eine Liste darstellen, welche die *IContainer*-Schnittstelle implementiert, indem es etwa die mit *AddElement* hinzugefügten Strings in dieser Liste speichert. Dieses Listen-Con-

tainer-Objekt wird im Beispielprogramm durch die Klasse *TListContainer* implementiert. Der Unterschied zur im letzten Abschnitt behandelten Klasse *TFormContainer* ist folgender:

- *TFormContainer* implementiert die gesamte »Container-Funktionalität«, indem es ein anderes Objekt (ein *TForm*-Objekt) *verwendet*, um etwa bei *AddElement* einen Schalter zum Formular hinzuzufügen. Die Klasse *TForm* wurde dadurch nicht verändert. *TFormContainer* erbte lediglich die elementare Funktionalität der Klasse *TInterfacedObject*.
- *TListContainer* soll zeigen, wie Sie die Unterstützung eines Interfaces zu einer bestehenden Klasse, die schon fast die gesamte für das Objekt gewünschte Funktionalität enthält, *hinzufügen*. Für die Verwaltung einer Stringliste bietet sich die von Delphi vordefinierte Klasse *TStringList* an, von der wir *TListContainer* auch ableiten werden.

In dieser Variante zeigt sich ein wesentlicher Vorteil der Interface-Technik im Allgemeinen: Verschiedene bestehende Klassen, die überhaupt nichts miteinander zu tun haben, können für eine Unterstützung eines neuen Interfaces erweitert werden (in diesem Beispiel wird die Klasse *TStringList* mit dem Interface erweitert). Wären die Methoden des Interfaces nicht in einem Interface, sondern in einer Basisklasse deklariert, müssten die Klassen alle von dieser Basisklasse abgeleitet werden. Da aber etwa *TStringList* nicht nachträglich von einer anderen Klasse abgeleitet werden kann, wäre auf diese Weise eine Erweiterung von *TStringList* nicht möglich.

Die Klasse TListContainer *R132*

TListContainer soll nun also unsere zweite Klasse werden, die das *IContainer*-Interface unterstützt, und sie soll von *TStringList* abgeleitet werden. Dies führt zu einem kleinen Problem, und zwar kann die Klasse dann nicht mehr von *TInterfacedObject* abgeleitet werden (dies wäre ja Mehrfachvererbung). Da wir uns entschlossen haben, sie von *TStringList* abzuleiten, müssen wir die Funktionalität von *TInterfacedObject* in *TListContainer* neu implementieren.

Das folgende Listing zeigt die Deklaration der Klasse *TListContainer*. Die Klassenelemente setzen sich aus den schon von *IContainer* bekannten Methoden sowie aus den Methoden zusammen, die von der *IInterface*-Schnittstelle gefordert werden, aber diesmal nicht von *TInterfacedObject* übernommen werden können:

```
type
  TListContainer = class(TStringList, IContainer)
  private
    FRefCount: Integer;
  protected
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
```

```
    function _Release: Integer; stdcall;
  public
    procedure IContainer.AddElement = Append;
    procedure DeleteElement(const ElementName: string);
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;
```

Die in der *QueryInterface*-Deklaration verwendeten Typen *TGUID* und *HResult* werden sehr häufig in der COM-Programmierung verwendet und bergen keine großen Geheimnisse: Für den Parameter *IID* können Sie einfach den Namen eines Interfaces angeben, das über die schon erwähnte Kennzahl (IID) verfügt (siehe auch Kapitel 8.5.5), der Compiler setzt dann statt des Namens diese Kennzahl ein. Auch die innere Struktur des Typs *HResult* ist unwichtig, da ein *HResult*-Rückgabetyp üblicherweise durch »0« oder eine vordefinierte Konstante ausgedrückt wird (im Folgenden bspw. die Konstante *E_NOINTERFACE*).

IInterface-Methoden

Um die *IInterface*-Methoden neu zu implementieren, wurden im Beispielprogramm einfach die von der vordefinierten Klasse *TInterfacedObject* verwendeten Anweisungen übernommen:

```
function TListContainer.QueryInterface(const IID: TGUID; out Obj):HResult;
begin // Speichert einen Verweis auf ein anderes Interface in der
      // Interface-Variablen Obj.
  if GetInterface(IID, Obj) then Result := 0 else Result := E_NOINTERFACE;
end;

function TListContainer._AddRef: Integer;
begin // Erhöht den Referenzzähler des Objekts um eins.
  Result := InterlockedIncrement(FRefCount);
end;

function TListContainer._Release: Integer;
begin // Verringert den Referenzzähler des Objekts um eins.
  Result := InterlockedDecrement(FRefCount);
  if Result = 0 then
    Destroy; // Falls Zähler bei 0, Objekt freigeben.
end;
```

Dabei ist *GetInterface* eine der in Kapitel 2.3.6 aufgelisteten Methoden von *TObject*, also quasi eine Grundmethode von Object Pascal. *InterlockedIncrement/Decrement* sind API-Funktionen, welche die einfachen Operationen *Inc* und *Dec* mit einer Abfrage des Wertes verbinden und gleichzeitig Konflikte zwischen mehreren Threads vermeiden. Da das Beispielprogramm kein Multithreading verwendet, hätte es statt dessen auch *Inc*

und *Dec* verwenden können, allerdings wäre der Code dann nicht so schön kurz geworden (siehe auskommentierte längere Code-Version im vollständigen Quelltext auf der CD).

IContainer-Methoden

Bei der Implementation der *IContainer*-Methoden haben wir nun den Vorteil, dass *TListContainer* bereits selbst eine StringList *ist*. Wir müssen also kein *TStringList*-Objekt erzeugen oder freigeben und können alle *TStringList*-Methoden direkt aufrufen. Auch hier soll wieder eine Beispiel-Methode genügen:

```
procedure TListContainer.DeleteElement(const ElementName: string);
begin
  if IndexOf(ElementName)>-1 then
    Delete(IndexOf(ElementName));
end;
```

Zuordnen zwischen Interface- und Klassenmethoden

Für die Methode *AddElement* sieht die Situation besonders günstig aus. Sie müsste eigentlich nur die *Append*-Methode der Stringliste aufrufen, um den String zur Liste hinzuzufügen. Da *Append* aber dasselbe Aufrufformat hat wie die *AddElement*-Methode in *IContainer*, brauchen wir überhaupt keine neue Methode zu schreiben, sondern dem Compiler nur mitzuteilen, dass die geerbte *Append*-Methode als Implementation der *AddElement*-Methode verwendet werden soll. Dies wird in der obigen Deklaration durch die folgende Zeile erreicht:

```
procedure IContainer.AddElement = Append;
```

Diese Möglichkeit der Verknüpfung zwischen unterschiedlich benannten Interface- und Klassenmethoden erlaubt es Ihnen, in Ihren implementierenden Klassen andere Methodennamen zu verwenden als im Interface vorgegeben.

Formularklasse und IContainer-Schnittstelle

Auch der Formularcontainer aus Kapitel 2.7.4 hätte auf diese Weise implementiert werden können, und zwar als Ableitung von einer Formularklasse, die alle Methoden von *IContainer* (inklusive *IUnknown*) neu implementiert:

```
TFormContainer2 = class(THelperForm, IContainer)
private
  FRefCount: Integer; // für die IUnknown-Funktionalität
protected
  // IUnknown-Methoden:
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  ...
  (Mehr dazu im vollständigen Quelltext auf der CD.)
```

Allerdings soll hier nicht ein weiterer Nachteil dieser Vorgehensweise verschwiegen werden: Die Klasse *TFormContainer* ist zur Laufzeit flexibler als die Klasse *TFormContainer2*, da sie mit Formularen unterschiedlicher Klassen arbeiten kann (die zu verwendende Klasse wird ja dem Konstruktor als Parameter übergeben). *TFormContainer2* ist dagegen auf eine bestimmte Formularklasse festgelegt, und zwar auf die Klasse, von der sie abgeleitet ist – in diesem Beispiel also *THelperForm*.

Ob es besser ist, eine Interfaces implementierende Klasse von *TInterfacedObject* abzuleiten und durch Verwendung von Objekten anderer Klassen zu implementieren wie in Kapitel 2.7.4 oder die Interfaces zu einer bestehenden Klasse hinzuzufügen wie in diesem Kapitel, hängt also von der Situation ab: Soll die Klasse möglichst flexibel sein wie *TFormContainer* oder soll sie effektiver implementiert werden können, wie z.B. die Klasse *TListContainer*, bei der wir eine der Interface-Methoden gar nicht zu schreiben brauchten, da sie schon von *TStringList* zur Verfügung gestellt wurde?

Hinweis: Versuchen Sie nicht, eine Klasse direkt von *TForm* abzuleiten, denn *TForm* versucht eine Formulardatei aus der fertig kompilierten Anwendung zu laden, und eine solche Formulardatei erhalten Sie nur, wenn Sie ein neues Formular und damit eine neue Formularklasse im Formulardesigner erzeugen.

Vergleich mit der Mehrfachvererbung

Im Zusammenhang mit Object Pascal wird oft die Frage gestellt, ob mit den Interfaces nun die Mehrfachvererbung unterstützt wird. Wenn man unter Mehrfachvererbung die Tatsache versteht, dass dort, wo man in früheren Turbo-Pascal-Versionen nur eine Elternklasse angeben konnte, jetzt mehrere Bezeichner stehen dürfen, dann wäre die Frage mit »ja« zu beantworten, ansonsten wäre die logische Antwort »nein«, denn das letzte Beispiel hat gezeigt, dass eine Klasse auch heute nur von *einer* anderen Klasse abgeleitet werden kann und nur die Variablen und den Code von dieser einen Klasse erben kann. So muss das Beispielprogramm *InterfaceDemo* in zwei Fällen die *IInterface*-Methoden neu implementieren, weil es eine Klasse nicht gleichzeitig von THelperForm *und* von TInterfacedObject ableiten kann.

Auch für Interfaces gilt, dass sie nur von jeweils einem anderen Interface erben können. Wir haben in Object Pascal also die Einfachvererbung von Klassen und die Einfachvererbung von Interfaces vorliegen. Klassen können allerdings *mehrere* Interfaces implementieren, aber dabei *erben* sie von diesen Interfaces überhaupt nichts, sie müssen sich mit ihren Methoden lediglich an die vom Interface vorgegebenen Regeln halten. Erben können sie diese Methoden allerdings von einer Basisklasse (wie z.B. von *TInterfacedObject*).

2.8 Funktionsbereiche der Laufzeitbibliothek

Da sich die in den Kapiteln 2.1 bis 2.7 besprochenen Spracheigenschaften alleine noch kaum sinnvoll einsetzen lassen, ist die Laufzeitbibliothek in Object Pascal, wie in anderen Sprachen auch, besonders wichtig. Daher ist ihr der Abschluss des zweiten Kapitels gewidmet, auch wenn sie nicht zur eigentlichen Sprache Object Pascal gehört.

Da die Bildschirmausgabe und die Eingaben des Benutzers bereits von der VCL abgedeckt werden, erwarten wir von der Laufzeitbibliothek hauptsächlich Dinge wie Funktionen zum Lesen, Schreiben und Verwalten von Dateien, Informationen über die Uhrzeit, Funktionen, die das Arbeiten mit Strings und anderen Datentypen vereinfachen, und mathematische Funktionen.

Natürlich ist es in diesem Kapitel nicht möglich, alle Funktionen der Laufzeitbibliothek auch nur knapp zu beschreiben, daher gibt dieses Kapitel nur einen Überblick über die verschiedenen Funktionsbereiche und erklärt die meisten der in diesem Buch verwendeten Funktionen.

2.8.1 Dateiverwaltung

Zu der Dateiverwaltung gehören unter anderem Funktionen, mit denen Sie

- Verzeichnisse erstellen, löschen und wechseln können (*MkDir*, *RmDir* und *ChDir*)
- Informationen über Dateien erhalten, beispielsweise über Existenz, Größe, Datum und Alter (*FileExists*, *FileSize*, *FileGetDate* und *FileAge*)
- Attribute oder den Namen einer Datei ändern können (z.B. *RenameFile*, *FileSetAttr* und *FileSetDate*)
- Strings, die Dateinamensangaben enthalten, manipulieren können: Beispielsweise können Sie die Pfadangabe, den Namen oder die Namenserweiterung extrahieren (mit *ExtractFilePath* und *Application.ExeName* wird beispielsweise in Kapitel 4.6.2 der Pfad ermittelt, in der sich die EXE-Datei der Anwendung befindet)
- wie im Beispielprogramm aus Kapitel 7.3.6 Verzeichnisse rekursiv auflisten.

Dateiauflistungsfunktionen *RIII*

Zum Finden aller Dateien, die bestimmte Kriterien erfüllen, gibt es in Delphi die beiden Routinen *FindFirst* und *FindNext*. Zunächst geben Sie die Kriterien, bestehend aus Dateimaske und Dateiattributen, an die Funktion *FindFirst* und übergeben als dritten Parameter eine Struktur des Typs *TSearchRec*, die das Ergebnis aufnimmt, zu dem der Dateiname und einige wichtige Attribute der Datei gehören:

```
FindFirst('*.*', faAnyFile, SearchRec);
```

Die Maske *.* und das Attribut *faAnyFile* sorgen dafür, dass die Funktion die erste aller Dateien zurückliefert. Statt *faAnyFile* können Sie auch *faDirectory* angeben, falls Sie nur an Verzeichnissen interessiert sind. Die anderen *fa...*-Konstanten, die in der Unit *SysUtils* zu finden sind, sind nur unter Windows wirksam, also nicht portabel nutzbar, da sie auf Eigenschaften des DOS-Dateisystems abzielen (simpel schreibgeschützte Dateien, versteckte Dateien, Systemdateien etc). *FindFirst* liefert ein Ergebnis ungleich Null, wenn sie keine Datei finden konnte.

Alle weiteren Dateien suchen Sie mit *FindNext*. Sie brauchen und können die Dateiauswahl-Kriterien nicht noch einmal angeben, *FindNext* schreibt die nächste passende Datei in den *TSearchRec*-Parameter und liefert ein Ergebnis wie *FindFirst*. Nach dem letzten Aufruf von *FindNext* sollten Sie mit *FindClose* dafür sorgen, dass die intern zur Dateisuche verwendeten Ressourcen wieder freigegeben werden.

Die Auflistung aller Dateien eines Verzeichnisses kann wie folgt stattfinden:

```
var
  F: TSearchRec;
  Error: Integer;
begin
  Error := FindFirst('*.*', faAnyFile, F);
  while (Error = 0) do begin
    ... Bearbeitung der Datei "F.Name" ...
    Error := FindNext(F);
  end;
  FindClose(F);
```

Ein Beispiel, das auch etwas mit den aufgelisteten Dateien anstellt, finden Sie wie schon erwähnt in Kapitel 7.3.6.

Hinweis: Die genannten Funktionen heißen zwar so, als wären sie zum Suchen da, werden aber durch Konkurrenz einer anderen Funktion auf ihr Spezialgebiet der Dateiauflistung zurückverwiesen. Diese Funktion heißt *FileSearch* und kann die Dateien sogar in einer Liste von Verzeichnissen suchen.

2.8.2 Dateieingabe und -ausgabe

In Delphi stehen – abgesehen von den Datenbankfunktionen – vorwiegend zwei Systeme zum Speichern von Daten auf Datenträgern zur Auswahl: die in Kapitel 4.3 beschriebenen objektorientierten Streams und die nicht objektorientierten Funktionen der Unit *System*, die noch aus alten Pascal-Zeiten stammen und die an dieser Stelle kurz angesprochen werden sollen.

Die *System*-Funktionen eignen sich für alle Zwecke gut, bei denen die gespeicherten Daten nicht polymorph sind, also nicht aus verschiedenen nicht vorhersagbaren Datentypen einer Klassenhierarchie bestehen.

Besondere Vorteile gegenüber den Streams weisen die Textdateien auf. Sie sind die einzigen Dateien, die die Datei nicht binär, sondern wie eine normale Textdatei, beispielsweise einen Pascal-Quelltext, ansehen und in denen Sie mit *writeln* und *readln* komfortabel auf einzelne Textzeilen zugreifen können.

Selbst wenn Sie für normale Dateioperationen mit Datenträgern nur Streams verwenden, erweisen sich die Textdateien beim Drucken von Text mit Hilfe des Objekts *Printer* als sehr nützlich (siehe Kapitel 5.6.4).

Dateitypen im Einsatz *R106*

Object Pascal kennt die folgenden drei Dateitypen: Textdatei, typisierte Datei (entspricht einem größenmäßig nahezu unbegrenzten Datenarray) und binäre/untypisierte Datei (in der die Daten von Struktur und Inhalt her völlig beliebig sein können).

Das folgende Beispiel zeigt die Unterschiede zwischen den drei Typen anhand des Einlesens einer kompletten Datei (der Datei-Inhalt wird nicht komplett im Hauptspeicher abgelegt, sondern nur kurz in Puffern gespeichert, die immer wieder überschrieben werden):

```
var
  ZeilenPuffer: string;
  Data: LongInt;
  Result: Word;
  LesePuffer: array[0..1023] of Byte;

{ Textdatei: }
  while not eof(Datei) do begin
    { jeweils eine Zeile lesen }
    readln(Datei, ZeilenPuffer);

{ typisierte Datei, bestehend aus LongInts: }
  while not eof(Datei) do begin
    { jeweils einen LongInt lesen }
    read(Datei, X);

{ untypisierte Datei, die mit reset(Datei, 1) geöffnet wurde: }
  repeat
    { jeweils ein kByte lesen }
    blockread(Datei, LesePuffer,
                              1024, { 1024 Datensätze zu je 1 Byte }
                              Result);
  until Result < 1024;
```

Bevor Sie die Dateien wie gezeigt verwenden können, müssen Sie die Datei öffnen und natürlich auch eine Variable deklarieren: Die Dateivariablen werden nach dem folgenden Muster deklariert:

```
var
  StartDatei: System.Text;  { Textdatei }
  DateiVoller32Bit: file of LongInt; { typisierte Datei }
  BinaerDatei: file; { untypisierte Datei }
```

Öffnen und Schließen von Dateien

Die grundsätzlichen Schritte zum Öffnen und Schließen einer Datei sind bei jedem der drei Dateitypen gleich: Zuerst müssen Sie die Dateivariable mit einer Datei verbinden (*AssignFile*), dann die Datei öffnen (*Reset*, *Rewrite* oder *Append*) und, nachdem die Ein- und Ausgabe abgeschlossen ist, dieselbe wieder mit *Close* schließen:

```
Assign(StartDatei, 'recording.102');
Reset(StartDatei); { je nach Dateityp }
...
Close(StartDatei);
```

Das Öffnen selbst unterscheidet sich wieder von Typ zu Typ:

- Textdateien können mit *Reset* zum ausschließlichen Lesen und mit *Rewrite* zum ausschließlichen Schreiben geöffnet werden. *Rewrite* löscht den bisherigen Inhalt der Datei und erwartet, wie der Name schon sagt, dass Sie den gesamten Inhalt aktualisieren, indem Sie ihn neu schreiben. Um eine Textdatei zu beschreiben, ohne sie zu löschen, verwenden Sie die Prozedur *Append*. Eine mit dieser Prozedur geöffnete Datei ist ebenfalls nur beschreibbar, jedoch werden die neuen Daten an die bestehende Datei angehängt.
- Bei typisierten Dateien entfällt die Prozedur *Append*, da sie nicht mehr notwendig ist: Mit *Reset* geöffnete typisierte Dateien können nämlich auch beschrieben werden. Der einzige Unterschied zwischen *Reset* und *Rewrite* besteht bei dieser Dateiart darin, dass *Rewrite* den bisherigen Dateiinhalt löscht (um Daten an eine mit *Reset* geöffnete Datei anzuhängen, müssen Sie den Positionszeiger der Datei nach dem Öffnen zuerst auf das Dateiende setzen: *Seek(FileSize(F))*).
- Für untypisierte Dateien gilt dasselbe wie für typisierte Dateien, allerdings können Sie den Funktionen *Rewrite* und *Reset* einen zweiten Parameter übergeben. Während bei typisierten Dateien nämlich die Größe eines Datensatzes, der mit *Read* gelesen werden kann (die Blockgröße), schon in der Deklaration der Dateivariablen durch den Basistyp angegeben ist, werden untypisierte Dateien per Voreinstellung in Datensätzen zu je 128 Bytes ausgelesen / beschrieben. Um beim Aufruf der Funktionen *blockwrite* und *blockread* die genaue Byte-Zahl angeben zu können, die

gelesen bzw. geschrieben werden soll, müssen Sie den zweiten Parameter von *Reset* auf 1 setzen (Datensatzgröße=1 Byte). Wie im oben gezeigten Beispiel für *blockread* können Sie dann immer noch auch große Datenblöcke am Stück auslesen.

Lesen und Schreiben

Für weitere Informationen zum Schreiben und Lesen der Dateien muss an dieser Stelle auf die Online-Hilfe verwiesen werden (neben den im obigen Beispiel verwendeten Routinen *read, readln* und *blockread* benötigen Sie die Routinen *write, writeln* und *blockwrite*).

Fehlerbehandlung

Das Öffnen von Dateien ist eine sehr fehleranfällige Operation: Dateien können z.B. schreibgeschützt oder nicht vorhanden, die Pfadangabe kann ungültig sein. Es gibt in Delphi zwei Methoden, diese Fehler abzufangen: über Exceptions und über die Funktion *IoResult* (gegenüber Borland Pascal wurde die strenge Laufzeitfehler-Methode also durch die behandelbaren Exceptions ersetzt). Welche der beiden Methoden verwendet wird, hängt vom Compilerschalter *$I* ab (siehe Kapitel 2.6.5).

2.8.3 Zeitformat und Zeitfunktionen

Delphi speichert Datums- und Zeitangaben in Variablen des Typs *TDateTime*, der intern lediglich eine Fließkommazahl ist, die eine Anzahl von Tagen angibt. So meint beispielsweise die Zahl 1,75 die Zeitspanne von einem Tag und 18 Stunden.

Wenn ein *TDateTime*-Wert ein festes Datum angibt, enthält er die Anzahl der Tage, die seit dem Datum 30.12.1899 vergangen sind. Enthält der Wert eine Zeit, so enthält der Teil nach dem Komma die Uhrzeit als Bruchteil von 24 Stunden.

Hinweis: Um OLE-Kompatibilität herzustellen, wurde das ursprüngliche Format von Delphi 1, bei dem vom Jahre 1 an mit der Zählung begonnen wurde, auf dieses neue Format umgestellt. *TDateTime*-Werte von 16-Bit-Delphi müssen daher um 693594 Tage verringert werden, um in der aktuellen Delphi-Version dasselbe Datum zu erhalten.

TDateTime-Rechnungen

Datum und Zeit können beliebig miteinander kombiniert und sogar verrechnet werden. Ergibt beispielsweise der Ausdruck *EndZeit-StartZeit* den Wert 14, so lagen zwei Zeitmessungen um zwei Wochen auseinander, wobei die Tageszeit nicht berücksichtigt wurde (oder zufällig in beiden Messungen genau übereinstimmte). Ein Ergebnis von 0.000116 gibt einen Zeitraum von ca. 10 Sekunden an.

Zeitroutinen

Im einem Beispielprogramm aus Kapitel 1 wurde bereits die Funktion *Time* verwendet, die die aktuelle Uhrzeit liefert. Entsprechend liefert *Date* das aktuelle Datum und *Now* die Summe aus beiden. Mit den Funktionen *StrToTime* und *TimeToStr* wurde eine Zeitangabe ebenfalls bereits in Kapitel 1 aus einem String gebildet und in einen String zurücküberführt (Kapitel 1.5.4, 1.5.5, 1.8.3, 1.8.5).

Weitere Funktionen wie *EncodeTime, DecodeTime, EncodeDate* und *DecodeDate* stellen sicher, dass Sie, ohne mit dem Fließkommawert rechnen zu müssen, auf einzelne Bestandteile der Zeitangabe (Monat, Sekunde etc.) zugreifen können.

Der System-Timer

In Fällen, in denen es beispielsweise darum geht, die Geschwindigkeit eines Programmteils zu messen, eignet sich die Windows-API-Funktion *GetTickCount* besser als die Bibliotheksfunktion *Time*. Sie liefert die Zahl der seit dem Systemstart vergangenen Millisekunden als LongInt-Wert (der 48 Tage lang ununterbrochen hochgezählt werden kann, ohne überzulaufen). Diese Funktion ist allerdings nicht portabel. Das Beispielprogramm *ThreadDemo1* aus Kapitel 4.6.1 etwa verwendet zur Zeitmessung doch lieber die *Time*-Funktion, damit es auch unter Linux unverändert läuft.

Jahrhundertwechsel-feste Programmierung

Bekanntlich bereitete die Umstellung auf das Jahr 2000 mancher Software große Probleme. Der Typ *TDateTime* kann alle vierstelligen Jahreszahlen der Zukunft speichern, daher ist bei der Programmierung Vorsicht nur dann geboten, wenn Sie den Benutzer Jahreszahlen in einem zweistelligen Format angeben lassen, denn dann ist es Interpretationssache, ob sich eine Angabe von »20« nun auf das Jahr 1920, auf 2020 oder vielleicht auf 2120 bezieht (dieser Fall kann ganz unverhofft eintreten, wenn man Ihr Programm länger verwendet als von Ihnen erwartet). Per Voreinstellung würde die VCL in der Funktion *StrToDate* eine »20« bis zum 31.12.2070 als 2020, danach als 2120, und die Jahreszahl »96« noch bis zum Jahr 2046 als »1996« interpretieren.

Sollten Sie mit dieser voreingestellten Interpretationsweise nicht einverstanden sein, können Sie sie über die Variable *TwoDigitYearCenturyWindow* ändern, die auf den Wert 50 voreingestellt ist. Ist der Wert dieser Variablen 0, erweitert die VCL eine zweistellige Jahreszahl immer auf das aktuell in der Systemzeit eingestellte Jahrhundert. Bei einem von Null verschiedenen Wert wird unter Umständen auch auf das vorherige oder nächste Jahrhundert erweitert. Nach Möglichkeit wird die Zahl der Vergangenheit zugerechnet, wobei *TwoDigitYearCenturyWindow* angibt, wie viele Jahre die erweiterte Jahreszahl maximal in die Vergangenheit reichen darf. Ein Wert von 1 in *TwoDigitYearCenturyWindow* bewirkt beispielsweise, dass im Jahr 2003 die Angabe »1.1.02« noch als 1.1.2002, der »1.1.01« bereits als 1.1.2101 interpretiert wird.

Abbildung 2.2 zeigt das Beispielprogramm *TwoDigitYears*, mit dem Sie diese Umwandlung aus der Perspektive des aktuellen Jahres interaktiv testen können. Es demonstriert auch, dass *TwoDigitYearCenturyWindow* keinerlei Auswirkungen auf Betriebssystemfunktionen – wie etwa das von Windows zur Verfügung gestellte Datumseingabeelement – hat (welches in Delphi durch die *TDateTimePicker*-Komponente gekapselt wird).

Abbildung 2.2: Ein Testprogramm für die Datumsumwandlung zeigt, dass die TDateTimePicker-Komponente bei zweistelliger Jahreseingabe das TwoDigitYearCenturyWindow nicht beachtet.

Hinweis: Die sicherste Form der Jahreszahleneingabe ist natürlich nach wie vor die vierstellige Eingabe. Diese können Sie in *TDateTimePicker* erzwingen, indem Sie das Property *Format* auf »dd.MM.yyyy« setzen. Je nach Betriebssystem ist dieses Format bereits standardmäßig voreingestellt.

2.8.4 Formatierungs-/Stringfunktionen

Dieses Kapitel befasst sich mit einer kleinen (überwiegend in den Programmen dieses Buchs verwendeten) Auswahl von Routinen zum Umgang mit Strings.

Formatierung von Strings

Delphi stellt neben *Format* auch einige andere Routinen zur Stringformatierung zur Verfügung (beispielsweise für die Verwendung von nullterminierten Strings oder vorgegebenen String-Puffern), allen Funktionen gemeinsam ist jedoch die Formatierung des Strings über einen *Formatstring*. In *Format* geben Sie diesen als ersten Parameter an, beispielsweise:

```
Label.Caption := Format('Testlauf %3d - Diagnose: %s.',[Count, Diag]);
```

Der Formatstring besteht aus einem Textgerüst, das auf jeden Fall in den Ergebnisstring aufgenommen wird, und Platzhaltern, in die weitere Parameter in einem bestimmten Format eingefügt werden. Ein Platzhalter beginnt mit einem »%« und endet mit einem Zeichen, das den Typ des einzufügenden Parameters angibt, beispielsweise »d« für eine Integer-Zahl, »e« für eine Fließkommazahl und »s« für einen String.

Zwischen diesem Zeichen und dem »%« kann sich unter anderem eine Angabe befinden, wie viele Zeichen der eingefügte Parameter in Anspruch nehmen soll.

Nach dem Formatstring geben Sie ein in eckigen Klammern eingefasstes Array an, in das Sie für jeden Platzhalter im Formatstring einen Parameter schreiben, der den einzufügenden Wert angibt. Im obigen Beispiel muss es sich bei *Count* um einen Integer-Wert, bei *Diag* um einen String handeln.

Der Ergebnisstring könnte im obigen Beispiel etwa »Testlauf 33 – Diagnose: keine Fehler.« lauten.

Formatieren von Fließkommazahlen und Datumsangaben

Mit den Funktionen *FormatDateTime*, *FormatFloat* und anderen Routinen können Sie genaue Kontrolle darüber ausüben, wie Zahlenwerte und Datumsangaben ausgegeben werden. Die beiden genannten Funktionen arbeiten prinzipiell genauso wie *Format*, erhalten aber einen Formatstring, der nach einem anderen Muster aufgebaut ist, und nur einen einzelnen zu formatierenden Wert (statt des Arrays).

Funktionen für Standard-Strings

Die folgenden Funktionen werden bei einfachen Aufgaben besonders häufig benötigt und stehen teilweise mit *Format* in Konkurrenz: So können Sie mit *IntToStr* eine Zahl schneller in Text umwandeln als mit *Format*; je mehr Zahlen jedoch aneinander gereiht werden, desto umständlicher wird der Umgang mit *IntToStr*, die für jede Zahl einmal aufgerufen werden muss.

Funktion in Kurzschreibweise	**Aufgabe**
function AnsiCompareStr(s1, s2): Integer;	vergleicht zwei Strings unter Berücksichtigung der Groß- und Kleinschreibung und gibt 0 zurück, falls die Strings übereinstimmen. Das Ergebnis ist kleiner als 0, falls *s1* kleiner als *s2* ist, und größer als 0, falls umgekehrt (berücksichtigt wie alle *Ansi...*-Funktionen auch länderspezifische Umlaute).
function AnsiCompareText(s1, s2): Integer;	vergleicht wie *CompareStr*, betrachtet aber Groß- und Kleinschreibung als identisch.
procedure AnsiLowerCase(Str);	wandelt einen String in Kleinbuchstaben um.
procedure AnsiUpperCase(Str);	wandelt einen String in Großbuchstaben um.
function Copy(Str, Index, Count);	kopiert einen Teil aus *Str* in einen neuen String und liefert diesen als Ergebnis zurück. *Copy* kopiert *Count* Zeichen ab der Position *Index*.

Funktion in Kurzschreibweise	Aufgabe
procedure Delete(Str, Index, Count);	löscht *Count* Zeichen von *Str* ab der Position *Index*. *Delete* ist keine Funktion, sondern ändert den Var-Parameter *Str* direkt.
procedure Insert(Teilstring, Zielstring, Zielposition);	fügt einen *Teilstring* in einen *Zielstring* ein, dabei gelangt das erste Zeichen des einzufügenden Strings an den Index *Zielposition* im Zielstring.
function IntToHex(Int): String;	wandelt den Zahlenwert in einen String um, wobei sie das hexadezimale Zahlensystem verwendet.
function IntToStr(Int): String;	wandelt den Zahlenwert in einen String um.
function Length(Str): Byte;	liefert die Länge von *Str*.
function Pos(SubStr, Str): Byte;	sucht *SubStr* im String *Str* und liefert die Position des gefundenen Teilstrings oder 0, falls *SubStr* nicht in *Str* enthalten ist.
procedure SetLength(Str, Laenge);	ändert bei einem langen String die Größe des reservierten Speichers und die Längenangabe selbst; bei einem String im alten 16-Bit-Format passt es nur das Längenbyte an (siehe Kapitel 2.4.4).
function StrToInt(Str): LongInt;	wandelt den String in einen Zahlenwert um und erzeugt eine *EConvertError*-Exception, falls der String fehlerhafte Zeichen enthält.

Neben den Routinen *AnsiCompareStr*, *AnsiCompareText*, *AnsiLowerCase* und *AnsiUpperCase* gibt es noch vier weitere Routinen für die gleichen Aufgaben, die jedoch kein »Ansi« im Namen haben. Diese Routinen berücksichtigen die Umlaute der Sprache, auf die das System konfiguriert ist, nicht.

Funktionen für nullterminierte Strings

Die folgende Tabelle listet nur die elementarsten Routinen für nullterminierte Strings auf, mit denen Sie unter anderem einen Übergang zu den Pascal-Strings herstellen können:

Funktion in Kurzschreibweise	Beschreibung
function StrLen(Str): Word;	ermittelt die Länge des Strings.
function StrCat (Dest, Source): PChar;	hängt *Source* an *Dest* an.
function StrUpper(Str): PChar;	wandelt einen String in Großbuchstaben um.
function StrLower(Str): PChar;	wandelt einen String in Kleinbuchstaben um.

Funktion in Kurzschreibweise	Beschreibung
function StrPCopy (Dest, PasStr): PChar;	kopiert den Pascal-String *PasStr* in den nullterminierten Stringpuffer *Dest*, der vorher bereits reserviert oder als festes Array deklariert worden sein muss.

Wenn eine Stringfunktionen einen *PChar*-Zeiger zurückliefert, ist das meistens kein neuer String (wie bei den Routinen der herkömmlichen Strings), sondern einer der schon als Parameter übergebenen Strings. Sie können die Funktionsaufrufe auf diese Weise einfach verschachteln. (Neue nullterminierte Strings werden von den Funktionen *StrNew* und *StrAlloc* erzeugt, siehe Online-Hilfe.)

2.8.5 Sonstige Funktionen

Die abschließende Tabelle fasst einige ständig benötigte Funktionen der Unit *System* zusammen. Sie sind in den meisten Fällen so einfach, dass der Compiler statt eines Funktionsaufrufs höchstens ein paar Maschinenbefehle zu generieren braucht:

Funktion	Aufgabe
function Addr(x): Pointer;	liefert die Adresse einer Variablen, wie der Adressoperator @.
function Assigned(x): Boolean;	liefert *True*, falls der als Parameter übergebene Zeiger nicht *nil* ist, bzw. falls das übergebene Objekt reserviert wurde und die Objektvariable nicht aus einem *nil*-Zeiger besteht.
function Chr(x: Byte):Char;	funktioniert wie die Typenumwandlung Char(x).
procedure Dec(var x);	verringert x um eins.
procedure Dec (var x, n: LongInt);	verringert x um *n*.
procedure FillChar (var x; Count; Val);	füllt *Count* Bytes von *x* mit dem Byte-Wert *Val*.
function High(x);	liefert den höchsten Index eines offenen Arrays oder den größten Wert eines Aufzählungstyps.
procedure Inc(var x);	erhöht x um eins.
procedure Inc (var x, n: LongInt);	erhöht x um *n*.
function IncludeTrailingPathDelimiter(s): String	in den Beispielprogrammen dieses Buchs häufig verwendete plattformunabhängige Funktion, die einen Pfadbegrenzer (unter Windows »\«, unter Linux »/«) an eine Pfadangabe anhängt, falls noch kein solcher vorhanden ist.
function Low(x);	liefert den zu *High(x)* gehörenden niedrigsten Wert.

Funktion	**Aufgabe**
function Ord(x): LongInt;	liefert die Ordinalzahl des Parameters und kommt einer Typenumwandlung mit *LongInt(x)* gleich.
function ParamCount: Word;	liefert die Zahl der Parameter, die die Anwendung als Start- oder Befehlszeilenparameter erhalten hat. (Startparameter in der Delphi-IDE: START | PARAMETER...)
function ParamStr(i): String;	liefert einen bestimmten Befehlszeilenparameter als String zurück.
function Random(Grenze): Word;	liefert eine nicht negative ganze Zufallszahl, die kleiner als *Grenze* ist.
procedure Randomize;	initialisiert den Zufallszahlengenerator.
function Round(r): LongInt;	rundet einen Fließkommawert auf eine ganze Zahl.
function SizeOf(x);	liefert die Größe einer Variablen oder eines Typs.
function Trunc(r): LongInt;	schneidet den Nachkommateil einer Fließkommazahl ab und gibt das Ergebnis als LongInt zurück.
function UpCase(c): Char;	wandelt das Zeichen *c* in einen Großbuchstaben um (ohne Umlaute).

3 Die Visual Component Library

Die Visual Component Library (VCL) spielt sowohl beim visuellen Entwurf einer Delphi-Anwendung als auch bei der tiefer gehenden Programmierung eine große Rolle. Wie der Name schon andeutet, enthält die VCL die mit Delphi mitgelieferten Komponenten (ausgenommen die Komponenten der Beispiel-Seite und die von anderen Herstellern beigesteuerten Komponenten in Delphis Lieferumfang). Die VCL ist jedoch weit mehr als eine normale austauschbare Komponentensammlung, denn unter der Oberfläche definiert sie die allgemeine Funktionsweise der Komponenten, einer Delphi-Anwendung und der Einbindung der Komponenten in die Delphi-IDE. Während Sie also die Komponenten der Komponentenpalette alle austauschen könnten, bleibt der Kern der VCL ein grundlegender Baustein einer jeden Delphi-Anwendung.

Kapitelüberblick

Dieses Kapitel beschäftigt sich sowohl mit den allgemeinen Grundlagen der VCL und einigen interessanten Aspekten ihrer internen Funktionsweise als auch mit speziellen VCL-Komponenten, die Sie in der Komponentenpalette finden: Der Übergang vom Allgemeinen zum Speziellen vollzieht sich dabei wie folgt:

- Die Abschnitte 3.1 bis 3.3 beschreiben Grundlagen der VCL wie die Beziehungen zwischen den Komponenten und die interne Verarbeitung der Windows-Nachrichten. Auch einige grundlegende Klassen werden näher betrachtet: die nicht-visuellen Klassen *TApplication* und *TScreen* und die abstrakten Klassen, die ihre Methoden, Properties und Ereignisse an die Komponenten der Komponentenpalette vererben.
- In den Abschnitten 3.4 und 3.5 geht es dann um den visuellen Grundbaustein jeder Delphi-Anwendung: das Formular. Während Kapitel 3.4 sich mit der Klasse *TForm* befasst und beschreibt, wie Sie mit den Objekten (Formularen) dieser Klasse umgehen (z.B. wie Sie sie dynamisch erzeugen und modal anzeigen können), erläutert Kapitel 3.5 allgemeine Techniken, mit denen Sie Formulare aufbauen können: Gruppieren von Steuerelementen, Verteilen der Komponenten auf mehrere Seiten etc. Kapitel 3.5 schließt mit der Verwendung der Standarddialoge, die Ihnen die Herstellung von alltäglichen Formularen ersparen.

- Der Abschnitt 3.6 beschäftigt sich mit einigen komplexeren Komponenten, die in Kapitel 1.9 noch nicht ausreichend gewürdigt werden konnten.
- Abschnitt 3.7 schließlich befasst sich mit Vereinfachungen des Formular-Entwurfs und behandelt verschiedene Techniken, mit denen gemeinsame Komponentengruppen in Formularen definiert werden können, hauptsächlich die Formularvererbung und die Frames.

Das Kapitel soll Ihnen einerseits in einer an der Vererbungshierarchie ausgerichteten Struktur die Elemente der VCL vorstellen, die besonders wichtig sind, als auch Ihnen einen Überblick über die seltener benötigten Funktionsbereiche der VCL geben, so dass Sie danach in der Lage sind, auch diese Bereiche schnell mit Hilfe der Online-Referenz zu erschließen.

3.1 Überblick über die VCL

Die VCL besteht aus einer Vielzahl von Units, von denen einige die Grundstruktur und eine Reihe abstrakter Klassen bereitstellen und einige andere sich auf die Implementierung von speziellen Komponenten konzentrieren. Wichtiger als die Aufteilung in Units ist die Klassenhierarchie, die sich durch die Zusammenführung aller Units ergibt.

Die Grundlage der Hierarchie sind abstrakte Klassen wie *TObject* und *TComponent*. Von diesen werden Sie wohl nie eine Instanz direkt herstellen, aber die meisten VCL-Klassen erben von ihnen wichtige Eigenschaften. Jede der abstrakten Klassen beschreibt ein Konzept der VCL auf kompakte Weise. Wollen Sie sich beispielsweise über das Wesen der Komponenten informieren, so gibt Ihnen die Klasse *TComponent* eine genaue Definition, durch welche Eigenschaften sich eine Komponente auszeichnet und welche Fähigkeiten alle Komponenten haben. Interessieren Sie sich als weiteres Beispiel für die Gemeinsamkeiten von Editierfeldern und Memos, so finden Sie diese in der abstrakten Klasse *TCustomEdit*.

Abbildung 3.1 zeigt die wichtigsten Klassen der VCL-Hierarchie und die Vererbungslinie, der die nächsten Kapitel folgen werden. Die Klassen in dieser Linie sind die wichtigsten, da sie durch die Vererbung zum Wesen der Klasse *TForm* und *TWinControl* beitragen, wobei *TWinControl* die Gemeinsamkeiten aller interaktiven Steuerelementkomponenten beschreibt und *TForm* mit dem Formular das visuelle Fundament einer Delphi-Applikation legt (das nicht-visuelle Fundament ist das Objekt *Application*, ein Objekt der Klasse *TApplication*, siehe hierzu Kapitel 3.2.3).

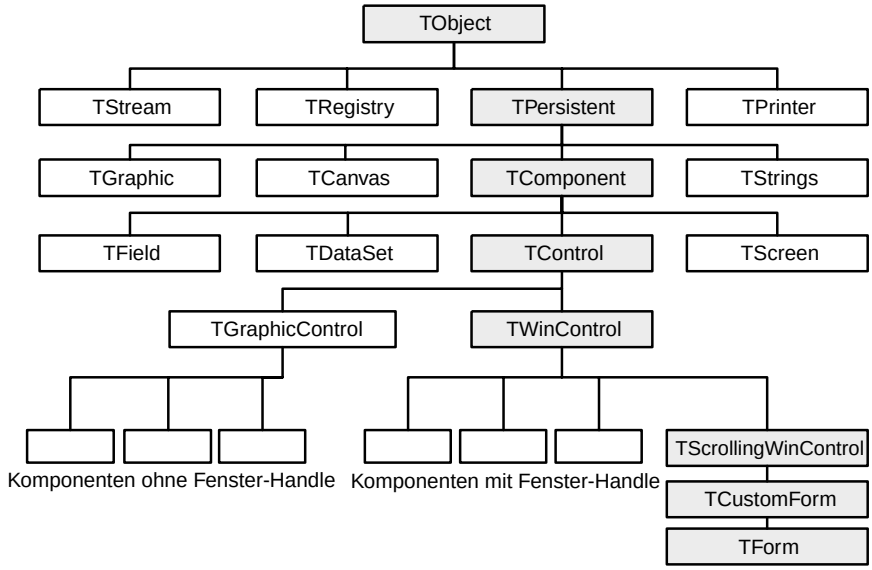


Abbildung 3.1: Das Gerüst der VCL-Klassenhierarchie von Delphi (detaillierte Grafiken siehe Anhang B)

3.1.1 Die grundlegenden Klassen

Wir beginnen in diesem Abschnitt mit grundlegenden Klassen, die im Hierarchiediagramm als oberste Klassen angesiedelt sind.

TObject

TObject ist die Wurzel des Hierarchiebaums. Von dieser Klasse wird jede neue Klasse automatisch abgeleitet, bei der nicht explizit eine andere Basisklasse angegeben ist. *TObject* ist so grundlegend, dass sie zum innersten Teil der Laufzeitbibliothek, der Unit *System* gehört. *TObject* hat große Verdienste auf dem Gebiet der Polymorphie: Da selbst die verschiedensten Klassen einen gemeinsamen Kern haben, nämlich die Elemente, die in *TObject* deklariert sind, können Methoden und Funktionen geschrieben werden, die mit allen nur denkbaren Objekten umgehen können. Das häufigste Beispiel dafür sind sicher die Ereignisbearbeitungsmethoden, deren Parameter *Sender* vom Typ *TObject* ist.

Die wohl wichtigsten Methoden von *TObject* sind der Konstruktor *Create* zum Initialisieren des Objekts und die Prozedur *Free*, die das Objekt per Destruktoraufruf freigibt (Allgemeines zur Initialisierung und Freigabe von Objekten finden Sie in Kapitel 2.3). In der VCL tritt der *Create*-Konstruktor von *TObject* häufig in den Hintergrund, da die abgeleiteten Klassen ihn mit eigenen, zum Teil virtuellen Konstruktoren überschreiben, die allerdings intern auch *TObject.Create* aufrufen.

Hinzu kommen einige grundlegende Methoden zur Laufzeit-Typinformation, die in Kapitel 2.3.6 beschrieben sind (*ClassName*, *ClassType* etc.) und die im alltäglichen Einsatz sehr nützlich sein können. Von den weiteren *TObject*-Methoden sollen hier noch drei grundlegende Methoden für die Speicherverwaltung erwähnt werden, die bei der alltäglichen Programmierung weniger gebraucht werden, aber dennoch interessant sind, da sie im Hintergrund wichtige Dienste verrichten:

virtuelle Methode	**Aufgabe**
destructor Destroy	übernimmt alle Aufgaben eines Destruktors und wird häufig überschrieben, jedoch normalerweise nicht direkt aufgerufen (statt *Destroy* sollten Sie *Free* aufrufen).
function NewInstance: TObject	reserviert die von *InstanceSize* zurückgegebene Speichermenge und gibt einen Zeiger darauf zurück. Durch Überschreiben dieser Methode kann man tief in die Speicherverwaltung einer Klasse eingreifen, so etwa wie in C++ durch Überschreiben des *new*-Operators. Überschreiben wäre z.B. dann sinnvoll, wenn viele kleine Objektinstanzen sich einen großen Speicherblock teilen sollen, so dass nicht für jedes Objekt eine eigene Speicherreservierung vorgenommen werden muss.
procedure FreeInstance	gibt den von *NewInstance* reservierten Speicher frei.

TObject-Ableitungen

Da *TObject* die voreingestellte Basisklasse für alle Klassen ist, befinden sich unter ihren Ableitungen viele Klassen, die ansonsten sehr wenig miteinander zu tun haben wie z.B. *TPrinter* und *TList*. Ohne *TObject* wären sie völlig eigenständige Klassen ohne Verwandtschaft. Allerdings sind viele dieser Klassen nicht immer ansprechbar – manche residieren in speziellen Units, die Sie extra einbinden müssen, z.B. die Klasse *TClipboard* in der Unit *Clipbrd*. (Die Units, in denen sich die visuellen Komponenten befinden, werden von Delphi automatisch zur *uses*-Anweisung hinzugefügt, wenn Sie die entsprechende Komponente in ein Formular einfügen.)

Manche *TObject*-Ableitungen verfügen über einige wenige Nachkommen, wobei besonders die *TFiler*- und *TStream*-Klassen (siehe Kapitel 4.3 und 4.3.4) wichtig sind. Alles in allem enden die meisten Vererbungslinien von *TObject* bereits nach spätestens zwei Schritten und sind damit leicht überschaubar.

Zwei *TObject*-Ableitungen bilden eine Ausnahme von dieser Regel: Die Klassen *Exception* und *TPersistent* sind ihrerseits die Wurzel eines sehr großen Ableitungsbaums. Da die Exceptions neben ihrer Klasse kaum weitere wichtige Informationen enthalten (siehe Kapitel 2.6.2), braucht diese Hierarchie nicht genauer untersucht zu werden.

Anders verhält es sich mit der *TPersistent*-Hierarchie: Fast alle häufig verwendeten Klassen sind von *TPersistent* abgeleitet, darunter auch alle Komponenten.

Die abstrakte Klasse TPersistent

Die Klasse *TObject* ist so allgemein gehalten, dass sie noch nicht einmal die Unterstützung von Dateioperationen beinhaltet. Diese kommt erst in der Klasse *TPersistent* dazu, allerdings ist sie dort nicht für allgemeine, sondern nur für interne Nutzung in der VCL selbst (beim Speicher- und Ladeprozess von Formularen) gedacht. Die Persistenz, die der Name andeutet, bezieht sich auf die Lebenszeit der *TPersistent*-Objekte: Sie kann durch das Speichern in einer Datei länger dauern als die Programmlaufzeit.

Natürlich können Sie auch Klassen, die nicht von *TPersistent* abstammen, in Dateien speichern; das besondere der *TPersistent*-Klassen ist, dass sie im Zusammenhang mit den *TFiler*- und *TStream*-Klassen zu polymorphen Dateioperationen fähig sind, was eben gerade beim Speichern eines Formularentwurfs wichtig ist.

Assign und DefineProperties

Die Klasse *TPersistent* enthält drei neue virtuelle Methoden, von denen nur eine dazu gedacht ist, direkt von Ihnen aufgerufen zu werden: Mit *Assign* weisen Sie einem Objekt ein anderes Objekt zu. Während Sie mit

```
Objekt1 := Objekt2;
```

lediglich zwei Variablen erhalten, die auf dasselbe Objekt weisen, kopiert *Assign* den *Inhalt* eines Objekts in ein anderes:

```
Objekt1.Assign(Objekt2);
```

Allerdings muss *Objekt1* vorher per Konstruktoraufruf initialisiert worden sein, da *Assign* kein neues Objekt *Objekt1* erstellt, sondern den Inhalt des bestehenden *Objekt1* ersetzt. Die genaue Funktionsweise von *Assign* hängt von den abgeleiteten Klassen ab. So können Sie beispielsweise der Methode *TClipboard.Assign* Grafikobjekte der Klassen *TBitmap*, *TIcon* und *TMetafile* übergeben, die diese dann in die Zwischenablage kopiert.

Die zweite Methode, *AssignTo*, wird intern von *Assign* verwendet, so dass nur noch die dritte Methode bleibt: *DefineProperties* wird normalerweise nur von der VCL aufgerufen und kann bei der Programmierung eigener Komponenten überschrieben werden (siehe Kapitel 6.4.2 und 6.6.6).

TPersistent-Ableitungen

Zu den Erben von *TPersistent* gehören neben der abstrakten Stringliste *TStrings* und deren Nachkommen einige Klassen, die mit Grafik zu tun haben, wobei in diesem Buch die folgenden Klassen behandelt werden:

- *TGraphicsObject* als Klasse für Zeichenwerkzeuge, mit denen Sie auf Zeichenflächen zeichnen können (siehe Kapitel 4.4)

- *TCanvas*, die allgemeine Zeichenfläche, die sowohl in den Fenstern des Bildschirms als auch in fensterunabhängigen Bitmaps (*TBitmap*) zu finden ist (ebenfalls Kapitel 4.4)
- *TGraphic* als allgemeine Klasse für die drei Grafiktypen Icon, Bitmap und Metadatei, denen das Kapitel 4.5 gewidmet ist
- die Klasse *TPicture*, die zur leichteren Verwaltung von *TGraphic* gedacht ist (ebenfalls Kapitel 4.5) und
- darüber hinaus alle Komponenten (Basisklasse *TComponent*)
- *TStrings*, die Basisklasse für Stringlisten (beschrieben in Kapitel 4.1.1)
- *TClipboard*, die Klasse für die Zwischenablage (behandelt in Kapitel 8.1).

3.1.2 Komponenten

Nachdem wir auch für die meisten *TPersistent*-Unterklassen bereits ein Kapitel gefunden haben, das sich diesen vollständig widmen kann, bleibt wieder eine Klasse übrig, die so viele Nachkommen hat, dass ein weiterer Überblick fällig ist. Wir folgen also der in Abbildung 3.1 hervorgehobenen Linie und setzen den Überblick mit der Klasse *TComponent* fort.

Die Klasse TComponent

Zu den grundlegenden Eigenschaften dieser Klasse gehört vor allem die Art, wie mehrere Komponenten miteinander verknüpft werden. Diese wird in Kapitel 3.2 beschrieben. Seit Delphi 3 enthält *TComponent* auch einige Methoden und Properties, die für die ActiveX-Funktionalität erforderlich sind. Da Delphi jedoch darauf ausgelegt wurde, die ActiveX-Unterstützung weitgehend automatisch zu liefern, werden wir diese intern verwendeten *TComponent*-Erweiterungen hier nicht weiter untersuchen.

TComponent ist die erste Klasse in der Hierarchie, die Properties veröffentlicht. Die folgenden zwei Properties sind somit für alle Komponenten das Minimum der im Objektinspektor editierbaren Eigenschaften:

- In *Name* befindet sich der Name, unter dem die Komponente im Object-Pascal-Code angesprochen werden kann. Sie können ihn auch zur Laufzeit ändern, wobei dann dieselben Regeln gelten wie zur Entwurfszeit: Sie dürfen einer Komponente keinen Namen geben, den bereits eine andere Komponente besitzt, sonst verursachen Sie eine Exception.
- Das Property *Tag* ist ein Allzweck-Property, in dem Sie nach Belieben einen 32 Bit großen Wert ablegen können. Ein häufiger Verwendungszweck ist die Unterscheidung zwischen verschiedenen Komponenten in einer einzigen Ereignisbearbeitungsmethode (siehe beispielsweise Kapitel 3.5.2).

Die folgende Übersicht zeigt weitere Elemente, auf die Sie nur zur Laufzeit zugreifen können; die meisten davon betreffen die Besitzverhältnisse, die in Kapitel 3.2.1 besprochen werden:

Element	Bedeutung
Components (Property) ComponentCount(Property) ComponentIndex(Property) Owner	Properties, die die Besitzverhältnisse der Komponenten definieren (Kapitel 3.2.1)
ComponentState (Property)	Menge von Flags, die über einige Vorgänge Bescheid geben, in die die Komponente gerade eingebunden ist. Neben vielen anderen gehören dazu als besonders wichtige Flags: *csLoading* (die Komponente wird aus einer Formulardatei geladen), *csDestroying* (wird unmittelbar vor dem Ablauf des Destruktors gesetzt), *csDesigning* (die Komponente ist Teil eines Formulars, dass gerade in der Delphi-IDE bearbeitet wird).
FindComponent, InsertComponnet, RemoveComponent (Methoden)	Methoden, die mit der Komponentenliste *Components* arbeiten (Kapitel 3.2.1)

Verschiedene Arten von Komponenten

Um es gleich vorwegzunehmen: Auch auf dieser Ebene der Klassenhierarchie gibt es wieder eine Klasse, die fast alle weiteren Nachkommen an sich zieht: *TControl*. Neben *TControl* gibt es eine noch relativ große Gruppe von Klassen, die von *TCommonDialog* ausgeht, und einige kleinere Gruppen oder einzelne Klassen wie *TTimer*, *TScreen*, *TMenuItem* und *TMenu*, die alle in einem eigenen Kapitel dieses Buchs behandelt werden. Auch viele wichtige nicht-visuelle Datenbankklassen sind Komponenten (*TField*, *TDataSet*, *TDataSource* etc.).

Alle *TComponent*-Typen außer *TControl* sind »nicht-visuell«, d.h. sie werden zur Entwurfszeit nur als Icon dargestellt und können dort nur verschoben, aber nicht anderweitig manipuliert werden. Das heißt jedoch nicht, dass von diesen Klassen zur Laufzeit des Programms nichts zu sehen ist, wie einige Beispiele zeigen sollen:

- *TMenuItem* und *TMenu* bezeichnen zwar Objekte, die zur Laufzeit des Programms sichtbar sind (Menüpunkte und ganze Menüs), sie werden jedoch nicht innerhalb des Formulars, sondern in einem eigenen Editor editiert.
- *TTimer* ist ein Beispiel für eine vollständig nicht-visuelle Komponente, sie ist also zur Progammlaufzeit nicht sichtbar und wird beim Formularentwurf durch ein Icon repräsentiert.

- *TScreen* schließlich gehört zu den Komponenten, die beim Oberflächenentwurf in der IDE überhaupt nicht zugänglich sind. Sie repräsentiert den Bildschirm, dessen Sichtbarkeit sich jedoch normalerweise nicht bestreiten lässt. Mehr zu *TScreen* finden Sie in Kapitel 3.2.4.

3.1.3 Visuelle Komponenten

Es bleibt nun die größte Untergruppe von *TComponent*, der Hierarchiebaum von *TControl*. Die Eigenschaft, in der sich *TControl* von den anderen direkten *TComponent*-Nachfahren unterscheidet, ist, dass *TControl*-Elemente zur Entwurfszeit innerhalb des Formulars manipuliert werden können.

Wie Sie aus Abbildung 3.1 erkennen können, sind die Dialogelement-Komponenten noch nicht direkt von *TControl* abgeleitet, *TControl* dient lediglich als Basisklasse für zwei verschiedene Typen von Dialogelementen: *TWinControl* und *TGraphicControl*.

Die Unterschiede zwischen TWinControl und TGraphicControl

Der grundlegende Unterschied zwischen diesen beiden Klassen ist, dass eine *TWinControl*-Instanz mit einem echten Windows-Fenster verbunden ist (ein Fenster, das von Windows verwaltet wird), während *TGraphicControl*-Instanzen von der VCL alleine verwaltet werden. In Delphi können Sie sie mit einigen Einschränkungen wie ein Fenster ansprechen, für Windows ist eine *TGraphicControl*-Komponente jedoch nur ein Bereich innerhalb eines richtigen Fensters, der nicht von den übrigen Fensterbereichen unterscheidbar ist.

Dies hat einige weitere Folgen:

- Von *TGraphicControl* abstammende Komponenten verbrauchen weniger (meistens gar keine) Windows-Ressourcen als von *TWinControl* abstammende.
- *TWinControl*-Elemente können den Tastatureingabefokus erhalten, Tastatureingaben in *TGraphicControl*-Elemente sind nur auf Umwegen möglich.
- *TWinControl*-Elemente können andere Komponenten enthalten und damit zum Eltern-Fenster werden.
- Andererseits benötigen *TGraphicControl*-Elemente unbedingt ein »Wirt-Fenster« (ein *TWinControl*), durch das sie erst sichtbar werden.

Das Fensterhandle von TWinControl

Normalerweise brauchen Sie sich nicht weiter um das Fensterhandle zu kümmern, wenn Sie aber auf Windows-API-Funktionen zurückgreifen müssen, die ein Fensterhandle als Parameter erwarten, benötigen Sie auch eine Komponente, die ein solches Fensterhandle besitzt, und das sind eben die *TWinControl*-Ableitungen. Das Handle

liegt günstigerweise in Form eines Properties vor und ist damit immer gültig. Wenn Sie also versuchen, *Handle* auszulesen, wenn das Fenster noch nicht existiert, kümmert sich die VCL zuerst um die Erstellung des Fensters, bevor sie den gültigen Wert von *Handle* zurückgibt (im Fehlerfall erzeugt sie eine Exception der Klasse *EOutOfResources*).

Grafikausgabe in TGraphicControl

Anstelle des Handles hat die Klasse *TGraphicControl* einen anderen wichtigen Vorteil: Sie verfügt über ein *Canvas*-Property, das die Ausgabe von Grafik im Fensterbereich, den die Komponente belegt, stark vereinfacht (siehe Kapitel 4.4). Dazu gehört die Methode *Paint*, die Sie in eigenen abgeleiteten Komponenten überschreiben können, und das Ereignis *OnPaint*, das es Ihnen ermöglicht, die Zeichenfunktionalität vom Objektinspektor aus in einer Formularmethode unterzubringen.

Die Klassen, die von *TGraphicControl* abgeleitet sind, nutzen diese Eigenschaften bereits und geben damit gute Beispiele für eigene *TGraphicControl*-Ableitungen. Sie sind von sich aus nicht interaktiv, können aber durch Bearbeitung der entsprechenden Ereignisse so gemacht werden.

Grafikausgabe in TWinControl

Der grundsätzliche Unterschied zwischen den beiden Control-Typen bedeutet jedoch nicht, dass Sie nicht auch in *TWinControl*-Elementen Grafik ausgeben können. Wenn Sie eine eigene Steuerelementkomponente erstellen wollen, die auch über ein Fensterhandle verfügt, sollten Sie sie von *TCustomControl* ableiten, einer Spezialisierung von *TWinControl*, die wie *TGraphicControl* über ein *Canvas*-Property verfügt. Für die Grafikausgabe vom Formular aus sollten Sie die Komponente *TPaintBox* verwenden, die von *TGraphicControl* abgeleitet ist.

3.1.4 Der Nachrichtenfluss

Ein sehr zentrales Thema der VCL ist der Weg, den eine Windows-Nachricht durch die verschiedenen Komponenten und Klassen durchläuft, bis sie schließlich zum *Event* wird und die Ereignisbearbeitungsmethode eines Formulars erreicht. Auf diesem Weg haben Sie viele weitere Eingriffsmöglichkeiten, die bei den folgenden Gelegenheiten besonders nützlich sind:

- zum Bearbeiten von Nachrichten, für die die VCL kein Event definiert
- beim Schreiben eigener Komponentenklassen, in denen Sie normalerweise keine an Events geknüpfte Ereignisbearbeitungsmethoden schreiben, da dieser Verknüpfungspunkt dem Benutzer der Komponente freistehen soll
- bei verschiedenen Aufgaben, die anwendungsglobaler Natur sind, z.B. das Anzeigen von Hinweisen in der Statuszeile oder die globale Bearbeitung von Nachrichten.

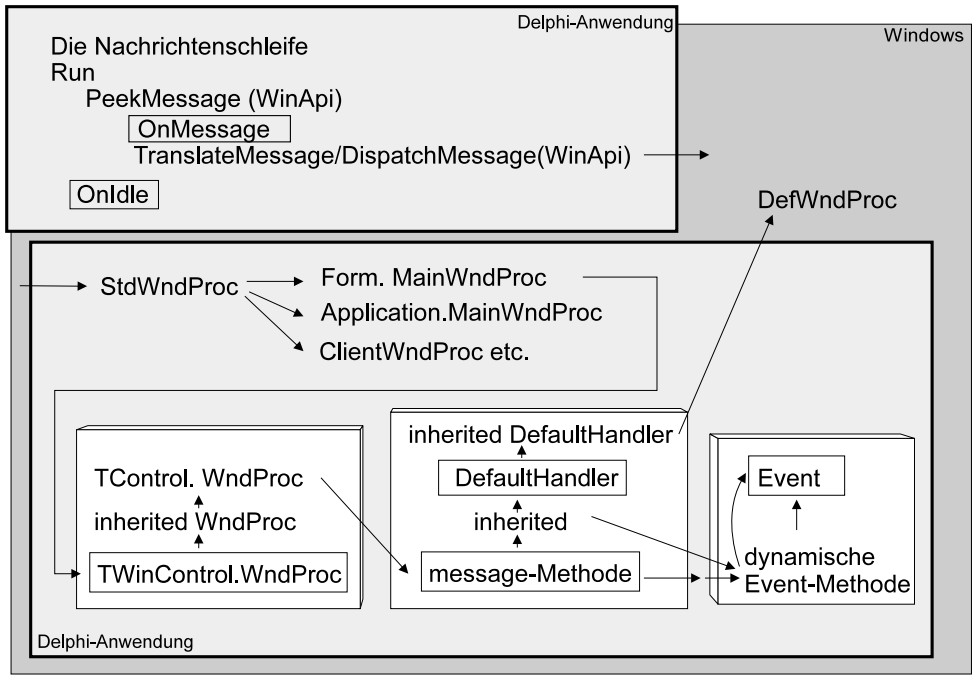


Abbildung 3.2: Die Wege einer Nachricht in einer Delphi-Anwendung. An den eingerahmten Positionen (z.B. bei OnMessage) gibt Ihnen die VCL die Möglichkeit, in das Geschehen einzugreifen.

Abbildung 3.2 zeigt den Nachrichtenfluss innerhalb einer Delphi-Anwendung und die Eingriffsmöglichkeiten, die Ihnen die VCL bietet. Bedingt durch die Schnittstelle von Windows wird fast jede Nachricht nicht nur von einem Fenster, sondern zuvor auch noch von der Anwendung bearbeitet. In einer Delphi-Anwendung übernimmt das *Application*-Objekt diese globale Nachrichtenbearbeitung.

Globale Nachrichtenbearbeitung

Normalerweise läuft die globale Nachrichtenbearbeitung durch das *Application*-Objekt völlig automatisch ab und führt dazu, dass die Nachricht an das zuständige Fenster gesendet wird. Falls Sie jedoch aus irgendeinem Grund alle Nachrichten, die die Anwendung auf diese Weise empfängt, in einer Methode bearbeiten wollen, können Sie eine solche Methode mit dem Ereignis *OnMessage* von *Application* verknüpfen. Manche Nachrichten, die nicht zur *MainWndProc* des Formulars oder der Steuerelemente gelangen, können Sie nur so bearbeiten (siehe Hinweis im nächsten Abschnitt).

Ein weiteres Ereignis, das während der globalen Nachrichtenbearbeitung auftritt, ist *OnIdle*; es zeigt an, dass die Anwendung zurzeit gerade nichts zu tun hat, und gibt Ihnen die Möglichkeit, die Prozessorzeit mit irgendwelchen Routineaufgaben zu fül-

len. Wie Sie diese beiden Ereignisse bearbeiten können, lesen Sie in Kapitel 4.7.1 (*OnIdle*) und Kapitel 5.8.3 (*OnMessage*); welche Ereignisse *TApplication* noch erzeugt, fasst Kapitel 3.2.3 zusammen.

Globale Nachrichtenbearbeitung intern

Intern besteht die Nachrichtenbearbeitung durch *Application* aus einer Variante der unter Windows üblichen Nachrichtenschleife. Windows lagert alle für eine Anwendung eintreffenden Nachrichten in der von Windows verwalteten Nachrichtenwarteschlange dieser Anwendung. Diese Warteschlange gleicht einem Briefkasten, aus dem die Anwendung immer die Nachricht herausnimmt, die schon am längsten wartet (für Win32 ist diese Darstellung eine starke Vereinfachung). In Abbildung 3.2 ist dieser Schritt im Aufruf von *PeekMessage* zu sehen. Mit *PeekMessage* sieht die Anwendung nach, ob überhaupt Nachrichten vorhanden sind. Falls das nicht der Fall ist, erzeugt sie das an anderer Stelle beschriebene *OnIdle*-Event. Erst wenn sich auch dadurch keine sinnvolle Beschäftigung für den Prozessor finden lässt, gibt die Anwendung die Kontrolle an Windows, so dass andere Anwendungen ihre Nachrichten bearbeiten können.

Jedes Mal, wenn *PeekMessage* eine neue Nachricht zutage fördert, generiert *Application* das schon erwähnte *OnMessage*-Event. In den meisten Fällen gibt sie dann die Nachricht einfach wieder an Windows zurück, das diese in seiner *DispatchMessage*-Funktion an das zuständige Fenster leitet, wo sie erneut von der VCL angenommen wird.

Hierzu muss die VCL eine Prozedur zur Verfügung stellen, die die Nachricht von Windows annimmt. Diese Prozedur heißt *StdWndProc* und hat die Aufgabe, die Nachricht wieder in objektorientierte Bahnen zu leiten. Also ruft sie eine *Methode* auf, und zwar die Methode *MainWndProc* des zuständigen Fensters.

Hinweis: Nicht alle Nachrichten, für die das *OnMessage*-Ereignis aufgerufen wird, kommen auch bei der Methode *MainWndProc* des Formulars an. *Application* registriert das Anwendungs-Icon, das im minimierten Zustand angezeigt wird, als zusätzliches Fenster. Auch an dieses sind einige Nachrichten adressiert, bei denen die VCL statt *MainWndProc* des Fensters die Methode *WndProc* der Anwendung aufruft. In MDI-Anwendungen ist außerdem die *ClientWndProc* des MDI-Formulars ein weiteres Ersatzziel der Nachrichten.

Von der Fensterprozedur zu WndProc

Auf diesen komplizierten Pfaden gelangt eine Nachricht nun also zu einer Steuerelementkomponente der VCL oder einer Ihrer selbst geschriebenen Komponenten. Die Nachricht liegt zunächst in Form eines *TMessage*-Parameters vor, der alle Originalbestandteile der Windows-Nachricht enthält bis auf das Fensterhandle, das ja aus dem Property *Handle* des Steuerelementobjekts abgerufen werden kann:

- *Msg* enthält einen Code, der die Art der Nachricht angibt,
- *WParam* und *LParam* enthalten je nach Nachricht unterschiedliche Daten. Unter Win32 sind beides als *LongInt* deklarierte 32-Bit-Parameter, unter 16-Bit-Windows hatte nur *LParam* den Typ *LongInt*, *WParam* war dort ein 16-Bit-*Word* (daher das *W* im Namen)[8].

Aufschluss über die Bedeutungen der Parameter *LParam* und *WParam* zu jeder einzelnen Windows-Nachricht geben Microsofts Referenzen zum Windows-API, die in allen Delphi-Versionen auch online vorliegen.

Die Aufgabe von *MainWndProc* ist es, in einem Block zur Exception-Behandlung alle Exceptions abzufangen, die von den anderen Methoden (beispielsweise von den Ereignisbearbeitungsmethoden des Formulars) nicht bearbeitet wurden. Bei jeder Exception ruft *MainWndProc* die Methode *HandleException* des *Application*-Objekts auf, die wiederum das *TApplication*-Ereignis *OnException* aufruft oder ein Meldungsfenster über die Exception anzeigt.

Von WndProc zur Standardprozedur

Zur eigentlichen Bearbeitung gibt *MainWndProc* das *TMessage*-Nachrichtenpaket an die Methode *WndProc* weiter. Da *WndProc* eine virtuelle Methode ist, können Sie sie überschreiben, um Zugriff auf *jede* Nachricht zu erhalten, die an Ihr Fenster gesendet wird. Dies ist allerdings nur in seltenen Fällen erforderlich, da die Nachricht an den noch folgenden Stellen des Nachrichtenflusses viel bequemer abgefragt werden kann.

Die *WndProc*-Methoden der VCL (beispielsweise definiert von *TWinControl* und *TForm*) suchen sich jeweils einige der Nachrichten heraus, um sie zu bearbeiten, und rufen dann die geerbte *WndProc*-Version auf, bis die Kontrolle schließlich an *TControl.WndProc* gelangt. Von dieser aus geht es dann zur nächsten Station, und zwar zu einer dynamischen Methode, die mit der *message*-Direktive für diese spezielle Nachricht deklariert wurde. Mehr zu *message*-Methoden erfahren Sie in Kapitel 6.4.1.

Falls keine derartige *message*-Methode für die Nachricht vorhanden ist, wird die Methode *DefaultHandler* aufgerufen, die die Standardbearbeitung für alle Nachrichten durchführt. Diese besteht normalerweise darin, dass sie die Nachricht an Windows zurückgibt (an die API-Funktion *DefWndProc*), so dass Windows das Standardverhalten von Fenstern und Steuerelementen verwirklichen kann.

Da auch die Methode *DefaultHandler* virtuell ist, können Sie auch diese überschreiben, dies sollte aber nur dann erforderlich sein, wenn sie statt *DefWndProc* eine andere Standardbearbeitung wünschen, was dann schon ein sehr spezieller Fall sein müsste.

---

8 Selbst unter Windows 98 gibt es noch einige API-Funktionen, die von *wParam* nur die Hälfte verwerten.

Events

Schließlich bleiben noch die aus dem Objektinspektor bekannten Events. Diese sind nicht einfach eine weitere Station in dieser Nachrichtenkette, sondern können an beliebigen Stellen auftreten. Die Bearbeitungsmethoden für Drag&Drop-Events (siehe Kapitel 5.8.3) werden beispielsweise schon von *TControl.WndProc* erzeugt, während andere Events erst in *message*-Methoden entstehen und wieder andere überhaupt keinen Zusammenhang zu einer Windows-Nachricht haben. Mehr zu den Events erfahren Sie in Kapitel 6.3.2.

Eilsendungen

Nicht alle Nachrichten, die an ein Fenster gelangen, sind vorher durch den oben beschriebenen globalen Teil der Nachrichtenbearbeitung von *TApplication* gelaufen. Nachrichten, die direkt mit der Windows-API-Funktion *SendMessage* gesendet werden, erreichen sofort den zweiten Teil der Nachrichtenbearbeitung, also die VCL-Prozedur *StdWndProc*, ohne vorher in der Nachrichtenschlange warten zu müssen. Ein Programm kann sich auch selbst Nachrichten mit *SendMessage* zusenden. Die Bearbeitung dieser Nachricht findet dann quasi geschachtelt in der aktuellen Nachrichtenbearbeitung statt. Diese wird erst fortgesetzt (mit der Anweisung, die dem Aufruf von *SendMessage* folgt), wenn die Bearbeitung der geschachtelten Nachricht abgeschlossen ist.

Noch schneller, da ganz ohne Vermittlung durch das Betriebssystem, geht es durch einen Aufruf der Methode *Perform(Msg, WParam, LParam)*, deren drei Parameter den Parametern der Fensterprozedur entsprechen. *Perform* ruft die für das Steuerelement zuständige Fensterprozedur innerhalb der Delphi-Anwendung direkt auf. Windows erhält nur dann die Kontrolle, wenn die Fensterprozedur die angegebene Nachricht der Standardverarbeitung (*DefWndProc*) zuleitet.

Asynchrone Nachrichten-Sendungen *R147*

Die Geschwindigkeit, mit der eine mit *SendMessage* abgeschickte Nachricht ihren Empfänger erreicht, bringt jedoch auch einen Nachteil mit sich: Da das aufrufende Programm bzw. die aufrufende Prozedur pausiert, während die Nachricht bearbeitet wird, können unerwünschte Wartezeiten entstehen. Wenn Sie eine Nachricht versenden wollen, von deren Bearbeitung der weitere Verlauf Ihres Programms nicht abhängt, senden Sie diese mit *PostMessage* statt mit *SendMessage*. *PostMessage* legt die Nachricht lediglich in der Nachrichtenwarteschlange des Empfängerfensters ab und kehrt dann zur aufrufenden Prozedur zurück.

Diese Vorgehensweise bietet sich besonders auch für die Organisation des Programmflusses innerhalb einer Anwendung an: Wenn z.B. in Ihrer Anwendung eine Aufgabe ansteht, die so bald wie möglich erledigt werden soll, die aber nicht bearbeitet werden darf, solange die aktuelle Nachricht noch nicht verarbeitet wurde; können Sie Ihrem

eigenen Fenster mit *PostMessage* eine selbst definierte Nachricht über die zu erledigende Aufgabe schicken. Auf diese Weise ist sichergestellt, dass diese erst dann bearbeitet wird, wenn die gerade laufende Ereignis-Bearbeitungsmethode und sämtliche damit zusammenhängenden Methoden der VCL (in Abbildung 3.2 z.B. *WndProc* sowie eventuelle *message*-Methoden und dynamische Event-Methoden) abgeschlossen sind.

Der TreeDesigner aus Kapitel 5 macht in einigen Fällen von dieser Art der Nachrichtenversendung Gebrauch (die im zweiten Punkt genannten Fälle sind im neuen TreeDesigner 3.5 nicht mehr relevant, geben hier aber trotzdem ein gutes Beispiel, das Sie im Code des alten TreeDesigners 3.0 auf der CD finden können):

- Das Automationsobjekt (siehe Kapitel 8.7) kann von einem Automations-Controller einen Auftrag erhalten, der längere Zeit in Anspruch nimmt, der aber nicht im Controller abgewartet werden soll. Konkret tritt dieser Fall ein, wenn der Benutzer des COM-Controllers dem Anwender, auf dessen Bildschirm der TreeDesigner läuft (verschiedene über DCOM verbundene Computer angenommen), eine Nachricht schicken will und der TreeDesigner daraufhin ein Meldungsfenster öffnet, dessen Schließen nicht im Automations-Controller abgewartet werden soll.
- Bei einigen Befehlen aus dem DATEI-Menü findet ein Formularwechsel (*TScreen.OnActiveFormChange*) statt. Da es sich um eine MDI-Anwendung handelt, wird bei dieser Gelegenheit das Menü neu aufgebaut, was im speziellen Fall des TreeDesigners wiederum dazu führt, dass die Liste der zurückliegenden Dateien im DATEI-Menü aktualisiert wird. Das Problem besteht nun darin, dass das DATEI-Menü erst dann erfolgreich aktualisiert werden kann, wenn die Bearbeitung des Menübefehls (in diesem Fall ist das die aktuelle Nachricht) abgeschlossen ist.

Weitere Detailschilderungen des letztgenannten Spezialfalls sind hier nicht interessant, daher schreiten wir zur Praxis: Die Bearbeitung des Menüpunkts DATEI | DRUCKEN im TreeDesigner soll erst dann durchgeführt werden, wenn das Menüereignis vorbei ist und das Menü sich wieder im Normalzustand befindet. Daher besteht der Ereignis-Handler lediglich aus einem Aufruf von *PostMessage*:

```
procedure TDocumentForm.DruckenClick(Sender: TObject);
begin
  PostMessage(Handle, CM_DATEI_DRUCKEN, 0, 0);
end;
```

Der Nachrichten-Code *CM_Datei_Drucken* ist selbst definiert:

```
const
  CM_DATEI_DRUCKEN = WM_APP + 104;
```

WM_App ist eine von Windows vorgegebene Konstante, die angibt, welche Nachrichtennummern die Anwendungen für selbst definierte Nachrichten verwenden können. Der Bereich zwischen *WM_App* (8000hex) und *WM_App+3FFFhex* (BFFFhex) gerät

nämlich garantiert nicht in Konflikt mit vordefinierten Windows-Nachrichten wie *WM_MouseMove*. (Im TreeDesigner werden übrigens auch die Nachrichten *WM_App + 1* bis *WM_App + 3* verwendet).

Die Definition einer Methode für die selbst definierte Nachricht sieht wie folgt aus (zur Definition von *message*-Methoden siehe auch Kapitel 6.4.1):

procedure CMDateiDrucken(var Msg: TMessage); message CM_DATEI_DRUCKEN;

Innerhalb dieser Methode finden nun genau die Aktionen statt, die normalerweise direkt im Ereignis-Handler *DruckenClick* hätten ausgeführt werden können:

```
procedure TDocumentForm.CMDateiDrucken(var Msg: TMessage);
begin
  Document.DeselectAll;
  PrintDialog.Doc := Document;
  if PrintDialog.ShowModal=mrOK then begin
    ...
```

Sie können sich selbst von der Wirksamkeit dieser Vorgehensweise überzeugen, indem Sie den Aufruf von *PostMessage* durch *SendMessage* ersetzen. So simulieren Sie den Normalfall, dass der Drucken-Dialog innerhalb des Ereignis-Handlers ausgeführt wird. Öffnen Sie direkt nach dem Schließen des Dialogs wieder das DATEI-Menü. Durch die oben bereits grob erläuterte Fehlersituation sind alle Menüpunkte bis auf die Liste der zurückliegenden Dateien verloren gegangen (gilt nur für den TreeDesigner 3.0).

3.2 Die Beziehungen der Komponenten

Während das letzte Kapitel die Vererbungshierarchie untersuchte, beschäftigt sich dieses mit der Besitzhierarchie. Normalerweise besitzt ein Formular alle Komponenten, die von Delphi automatisch zu der Klassendeklaration des Formulars hinzugefügt werden, jedoch ist das Hinzufügen zur Klassendeklaration noch nicht ausschlaggebend dafür, ob eine Komponente im Besitz einer anderen ist oder nicht.

Die Unterschiede zwischen dieser Besitzhierarchie und der Fensterhierarchie sowie die Art, wie diese Beziehungen in der VCL dargestellt werden, sind Thema dieses Kapitels.

3.2.1 Besitzhierarchie und Komponentenlisten

Zwar können Sie die Komponenten, die sich im Besitz eines Formulars befinden, jederzeit direkt mit dem Namen ansprechen, in manchen Fällen ist es jedoch wünschenswert, alle »besessenen« Komponenten in einer Schleife behandeln zu können, z.B., um sie in einer Datei zu speichern, wie die VCL es beim Speichern des Formulars tun

muss. Zu diesem Zweck enthält die Klasse *TComponent* eine Liste, die den gesamten Komponentenbesitz, der normalerweise schon in Formularvariablen zu finden ist, auf eine zweite Weise speichert.

Eine wichtige Bedeutung des Komponentenbesitzes liegt darin, dass die Besitzerkomponente normalerweise dafür verantwortlich ist, die ihr gehörenden Komponenten freizugeben, wenn auch sie selbst freigegeben wird.

Properties von TComponent zur Besitzverwaltung

Die Organisation dieser Komponentenliste ist dabei unwichtig, da sie in Form eines indizierten Properties vorliegt. Damit jede Komponente nicht nur weiß, was sie besitzt, sondern auch, wem sie gehört, verfügt *TComponent* außerdem über das Property *Owner*. Diese und zwei weitere Properties sind in der folgenden Tabelle zusammengefasst – die Properties sind allesamt nur zur Laufzeit ansprechbar:

TComponent-Property	Typ	Bedeutung	Bemerkung
Components	array of TComponent	Liste der Komponenten, die der Komponente gehören	nur lesen
ComponentCount	Integer	Zahl der Komponenten in der Liste *Components*	nur lesen
ComponentIndex	Integer	Index der Komponente in der *Components*-Liste des Besitzers	lesen und schreiben
Owner	TComponent	weist auf den Besitzer der Komponente	nur lesen

Veränderung der Komponentenliste zur Laufzeit

Wenn Sie zur Laufzeit eine neue Komponente erzeugen, sorgen Sie für die richtige Verknüpfung zu seiner Besitzerkomponente, indem Sie diese als Parameter für den *Create*-Konstruktor der neuen Komponente angeben. Zu diesem Zweck erwarten alle Komponentenkonstruktoren einen Parameter *AOwner* des Typs *TComponent*, dem Sie auch den Wert *nil* übergeben können, falls die Komponente keinen Besitzer haben soll. Auf diese Weise können Sie Komponenten erzeugen, die völlig unabhängig von allen anderen Komponenten sind.

Der Komponentenkonstruktor setzt also die Properties *Owner* und *ComponentIndex* der eigenen Komponente und sorgt dafür, dass die Properties *Components* und *ComponentCount* der Besitzerkomponente angepasst werden. Obwohl die Properties *Components* und *Owner* nicht beschreibbar sind, können Sie die Komponenten-Besitzverhältnisse zur Laufzeit auch auf andere Weise beeinflussen:

- Wenn Sie eine Komponente nachträglich (also nach der Konstruktion mit *Create*) in den Besitz einer anderen Komponente einfügen wollen, rufen Sie beim neuen Besitzer die Methode *InsertComponent* auf, die auch vom *Create*-Konstruktor verwendet wird.
- Um eine Komponente aus dem Besitz einer anderen Komponente zu löschen, müssen Sie, statt *Owner* auf *nil* zu setzen, die Methode *RemoveComponent* des Besitzers aufrufen, z.B. *Owner.RemoveComponent(self)*. Dadurch wird *Owner* indirekt auf *nil* gesetzt. Die *Components*-Liste funktioniert genauso wie *TList*, füllt also Lücken, die durch Entfernen von Elementen entstehen, sofort durch Nachschieben aller folgenden Elemente auf.
- Mit dem einzigen beschreibbaren Property aus der obigen Liste, *ComponentIndex*, können Sie schließlich die Position der Komponente in der Liste des Besitzers verändern.

Wenn Sie eine Komponente zur Laufzeit löschen, sorgt der Destruktor selbstverständlich automatisch dafür, dass sie aus der Liste des Besitzers gestrichen (mit *RemoveComponent*) und dass ihr eigener Komponentenbesitz gelöscht wird.

Herkunft der Components-Liste

Interessant ist auch, wie die VCL die Komponenten eines Formulars zur Laufzeit wiederherstellt. Dazu genügt es nämlich nicht, einfach alle Komponenten wiederherzustellen und dabei die *Components*-Listen aufzubauen. Wenn eine Komponente dem Formular gehört, so ist in dessen Klassendeklaration höchstwahrscheinlich eine Objektvariable enthalten, die ebenfalls auf diese Komponente weist. Auch diese Tatsache ist in der `dfm`- bzw. in der EXE-Datei gespeichert. Wenn also eine solche Variable existiert, muss die VCL auch diese auf die neu erzeugte Komponente setzen. Enthält das Formular beispielsweise eine Komponente mit dem Namen *LogoImage*, setzt die VCL sowohl *LogoImage* selbst als auch einen Listeneintrag von *Components* auf diese Komponente (eine `dfm`-Datei enthält zu diesem Zweck den Namen der Variablen, die auf die neue Komponente weisen sollen. Um von diesem Namen auf die Adresse der Variablen schließen zu können, verwendet die VCL hier die Methode *TObject.FieldAddress*).

3.2.2 Die Fensterhierarchie

Bei einfachen Formularen ist es klar, dass die im letzten Abschnitt beschriebene Besitzhierarchie auch der Fensterhierarchie entspricht: Das Elternfenster ist also nicht nur »Vater und Mutter«, sondern auch Besitzer der Komponenten. In manchen Fällen ist es jedoch sinnvoll, diese Beziehungen etwas zu differenzieren. Ein sehr häufiges Beispiel dafür sind die Panels und Gruppen von Steuerelementen.

Haben Sie beispielsweise mehrere Radioschalter, die Sie zu einer Gruppe zusammenfassen wollen, so sollten Sie sie in einer *TPanel*-Komponente platzieren. Auf diese Weise wird das Panel zum Elternfenster der Radioschalter, obwohl das Formular deren Besitzer wird.

Properties zur Fensterhierarchie

Auch die untergeordneten Fenster müssen oft so effektiv über eine Liste angesprochen werden können wie die Komponenten der *Components*-Liste. So kann ein Fenster beispielsweise die Liste seiner Unterfenster nach einem bestimmten Komponententyp durchsuchen (um beispielsweise alle Aktionsschalter in einer Schleife zu deaktivieren).

Da jedoch die Klasse *TComponent* noch nichts mit Fenstern zu tun hat, enthält sie auch keine Elemente zur Speicherung der Fensterhierarchie. Die Klassen, die diese Elemente einführen, sind *TControl* und *TWinControl*. *TWinControl* verfügt über eine Liste aller enthaltenen Steuerelemente (Unterfenster) und *TControl* besitzt das entsprechende *Parent*-Feld:

Property	Typ	ab Klasse	Bedeutung	Bemerkung
Controls	array of TControl	TWinControl	Liste der untergeordneten *TControl*-Elemente	nur lesen
ControlCount	Integer	TWinControl	Zahl der Elemente in *Controls*	nur lesen
Parent	TWinControl	TControl	übergeordnetes Fensterelement eines *TControl*-Elements	beschreibbar

Dies ist ein weiterer Unterschied zur Besitzhierarchie: Nicht alle Komponenten, die Kinder sein können, können auch Eltern sein. Alle grafischen Steuerelemente, deren Klassen von *TGraphicControl* abgeleitet sind, können lediglich in einem anderen Fenster enthalten sein, aber keine Unterfenster mehr besitzen. Dies ist eine Fähigkeit der Vertreter des *TWinControl*-Teilbaums der Klassenhierarchie. Da *TWinControl* natürlich das *Parent*-Feld erbt, können *TWinControl*-Komponenten beliebig geschachtelt sein.

Veränderungen zur Laufzeit

Für die Komponenten, die Sie schon zur Entwurfszeit in das Formular einfügen, sorgt die VCL automatisch für den Aufbau der Eltern-Kind-Beziehungen. Wenn Sie eine Komponente zur Laufzeit selbst erzeugen, müssen Sie sie explizit mit einem Elternfenster verbinden, indem Sie ihr *Parent*-Property auf das Elternfenster setzen, sonst erhält sie überhaupt kein Elternfenster. Eine Ausnahme sind MDI-Kindformulare. Zu diesen sucht die VCL automatisch das MDI-Hauptfenster (von dem es nur ein einziges geben kann) aus den bestehenden Formularen als Elternfenster heraus – wenn Sie hier

das *Parent*-Property selbst setzen, kommt es unter Umständen zu merkwürdigen kleinen Fehlern (siehe Kapitel 5.7.4). Bei Formularen, die kein übergeordnetes Formular haben, bleibt das *Parent*-Property *nil*.

Den Methoden *InsertComponent* und *RemoveComponent* entsprechen die *TControl*-Methoden *InsertControl* und *RemoveControl*. Anders als *Owner* ist *Parent* jedoch ein beschreibbares Feld, so dass Sie nicht die Methode *InsertControl* verwenden müssen, um diese Verbindung nachträglich herzustellen. Ebenso sparen Sie sich einen Aufruf der Methode *RemoveControl*, indem Sie *Parent* einfach auf *nil* setzen. Ein weiterer Unterschied zur Komponentenliste ist schließlich, dass Sie die Position eines Elements innerhalb von *Controls* nicht beliebig festlegen können, ein Property *ControlIndex* gibt es also nicht. Indirekt können Sie auf den Index im *Controls*-Array Einfluss nehmen, indem Sie eine Komponente mit *SendToBack* in den Hintergrund setzen oder mit *BringToFront* in den Vordergrund holen (siehe *Z-Reihenfolge von Komponenten* in Kapitel 3.3.1).

In Kapitel 5.7.5 finden Sie ein Beispiel, bei dem das *Parent*-Property mehrfach zur Laufzeit geändert wird.

3.2.3 Die oberste Komponente: TApplication

Nachdem wir nun die Komponentenverknüpfungen innerhalb von Formularen genau durchleuchtet haben, bleibt die Frage offen, wer Besitzer und Elternfenster des Formulars sind. Während das *Parent*-Property eines Hauptfensterformulars *nil* bleibt, gibt es eine Komponente, die das Formular besitzt:

Sie wird in der Unit *Forms* deklariert und erzeugt, hat den Typ *TApplication* und heißt *Application*. Da die Unit *Forms* automatisch in jede Formular-Unit eingebunden wird, können Sie in Ereignisbearbeitungsmethoden ohne weitere Vorbereitung auf das *Application*-Objekt zugreifen. *TApplication* definiert viele Methoden und Properties, die globale Bedeutung haben und ohne OOP in globalen Funktionen oder globalen Variablen implementiert werden müssten. Zu den Aufgaben von *Application* gehören neben der Nachrichtenbearbeitung z.B. die Anzeige von Hinweisfenstern, der Aufruf der Hilfefunktion und die endgültige Behandlung von Exceptions.

Wie andere Komponenten auch, benachrichtigt Sie *Application* über Events von seinen eigenen Aktionen und lässt Sie in gewissem Umfang Einfluss auf diese Aktionen nehmen, teilweise mit Properties, teilweise über die Events. Die Anzeigen der Hinweisfenster, die Sie auch in der Delphi-IDE erhalten (wenn Sie die Maus über die Schalter unter der Menüzeile halten), wird beispielsweise grob durch die Properties *ShowHint*, *HintColor* und *HintPause* gesteuert (wenn Sie das Event *OnShowHint* bearbeiten, können Sie noch mehr Einfluss ausüben). Die folgende Tabelle listet zunächst die wichtigsten Properties von *Application* auf:

Property	Typ	Bedeutung
Active	Boolean	gibt an, ob ein Fenster dieser Anwendung aktiv ist; wenn *False*, ist eine andere Anwendung aktiv.
DialogHandle	Word	erlaubt die nicht-modale Einbindung fremder Dialogfenster in die Delphi-Anwendung, bei der die VCL einige Nachrichten an diese Dialogfenster weitergeben muss (siehe Online-Hilfe).
ExeName	String	Name der EXE-Datei der Anwendung
Handle	HWnd	Handle des Anwendungsfensters, das die Anwendung als Icon darstellt (das Anwendungsfenster ist nicht mit dem Hauptfenster identisch).
HelpFile	String	Name der Hilfedatei, auf die sich die *HelpContext*-Werte in anderen Properties und an anderen Stellen des Quelltextes beziehen
Hint	String	der Hinweistext, der gerade zu dem Element, über dem sich die Maustaste befindet, angezeigt werden soll. Zur Verwendung mit dem Ereignis *OnHint*, siehe Tabelle unten.
HintColor	TColor	Farbe des Hinweistextes
HintHidePause	Integer	Anzahl der Millisekunden, nach denen das Hinweisfenster ausgeblendet wird (Voreinstellung 2,5 Sekunden).
HintPause	Integer	Anzahl der Millisekunden, die die VCL wartet, bis sie ein Hinweisfenster anzeigt (Standard ist 500).
HintShortCuts	Boolean	Wenn *True*, werden die Hinweistexte in den Hinweisfenstern automatisch um das mit der Komponente verknüpfte Tastenkürzel ergänzt.
HintShortPause	Integer	Wenn bereits ein Hinweis eingeblendet wurde, gibt dieses Property die Wartezeit an, bis ein anderer Hinweis eingeblendet werden kann (Standard: 50 ms).
Icon	TIcon	Icon, das die Anwendung als Ganzes repräsentiert; wird im Explorer und in der Task-Leiste angezeigt.
MainForm	TForm	weist auf das Hauptformular der Anwendung.
ShowHint	Boolean	schaltet die Hinweisfunktion für die gesamte Anwendung ein oder aus. Ein Hinweis zu einer Komponente wird immer dann angezeigt, wenn sowohl das *ShowHint*-Property der Anwendung als auch das *ShowHint*-Property der Komponente den Wert *True* haben (bzw. wenn *ParentShowHint* und die Elternkomponente es erlauben).
ShowMainForm	Boolean	Hiermit können Sie die automatische Anzeige des Hauptformulars beim Starten der Anwendung verhindern und Ihre Anwendung so im Hintergrund laufen lassen (eignet sich z.B. für COM-Automations-Server).
Terminated	Boolean	wird von der VCL auf *True* gesetzt, wenn die Anwendung beendet wird.
Title	String	Beschriftung des Anwendungs-Icons

Property	Typ	Bedeutung
UpdateFormatSettings	Boolean	bestimmt, ob Delphis Formateinstellungen für Datums- und Zeitangaben während der Programmlaufzeit an Änderungen in der Systemkonfiguration angepasst werden sollen (Voreinstellung: *True*).
UpdateMetricSettings	Boolean	Die Voreinstellung *True* sorgt für die Anpassung von *Screen.IconFont* (siehe Kapitel 3.2.4) bei jeder entsprechenden Änderung der Systemeinstellungen.

Die interessantesten Methoden von *TApplication* sind:

Methode	Beschreibung
BringToFront	macht die Anwendung zur aktiven Anwendung, bringt also ihre Fenster in den Vordergrund.
Minimize	minimiert alle Fenster der Anwendung zu einem Symbol, so als wäre das Hauptfenster der Anwendung minimiert worden.
NormalizeTopMosts	setzt zeitweise die Wirkung des Formular-Stils *fsStayOnTop* für alle Fenster der Anwendung außer dem Hauptfenster aus.
NormalizeAllTopMosts	wie *NormalizeTopMosts*, wirkt jedoch auch auf das Hauptfenster.
ProcessMessages	ist eine von mehreren Möglichkeiten, kooperatives Multitasking zu realisieren (siehe Kapitel 4.7.1). Für ein einfaches Beispiel siehe Kapitel 7.3.7.
Restore	macht *Minimize* wieder rückgängig.
RestoreTopMosts	reaktiviert das mit *NormalizeTopMosts* deaktivierte Flag.
ShowException(E: Exception)	meldet dem Benutzer des Programms eine Exception und deren Fehlermeldungs-Inhalt (Property *Message*).
Terminate	beendet die Ausführung des Programms.

Weitere Gruppen von Methoden befassen sich (je nach Delphi-Version) unter anderem mit dem Aufrufen der Hilfefunktion (*HelpContext, HelpCommand* und *HelpJump*), mit dem manuellen Anzeigen von Hinweisfenstern (*ActivateHint* und *CancelHint*) sowie mit dem manuellen Aufrufen von Aktionen (*ExecuteAction*).

Ereignisse

TApplication ist eine der Klassen, die nicht im Objektinspektor bearbeitet werden können, die aber dennoch Events zur Verfügung stellen. Sie können diese Events mit Methoden Ihres Formulars oder eines anderen Objekts verknüpfen, müssen diese Methoden aber selbst in die Deklaration des Formulars bzw. der Klasse des anderen Objekts einfügen und auch deren Gerüst (Methodenkopf, *begin...end*) selbst erstellen. Ein Beispiel dafür finden Sie in Kapitel 1.9.5, R57 (Seite 170). Die folgende Tabelle gibt eine Übersicht über die zur Wahl stehenden Ereignisse:

Event	Parameter für die Methode	Beschreibung
OnActionExecute, OnActionUpdate	Action: TBasicAction; var Handled: Boolean	treten für jede aufgerufene Aktion auf, die nicht bereits in einer *OnExecute*- bzw. *OnUpdate*-Methode der Aktionsliste erledigt wurde.
OnActivate	Sender: TObject	tritt auf, wenn die Kontrolle von einem Fenster aus einer anderen Anwendung zu einem Fenster dieser Anwendung übertragen wird.
OnDeactivate	Sender: TObject	tritt auf, wenn ein Fenster dieser Anwendung deaktiviert und dafür eine andere Anwendung aktiviert wird.
OnException	Sender: TObject; E: Exception	wird von der Methode *HandleException* erzeugt.
OnHelp	Command: Word; Data: LongInt; var CallHelp: Boolean	wird aufgerufen, bevor das *Application*-Objekt auf automatischem Wege die Hilfedatei aufruft.
OnHint	Sender: TObject	tritt auf, wenn die Maus sich über einer Komponente befindet, zu der ein Hinweistext angezeigt werden *könnte*. Siehe Beispiel in Kapitel 1.9.5, R57 (Seite 170).
OnIdle	Sender: TObject; var Done: Boolean	wird in nachrichtenfreien Zeiten aufgerufen, siehe auch *Globale Nachrichtenbearbeitung*, Seite 321 und Kapitel 4.7.1.
OnMessage	var Msg: TMsg; var Handled: Boolean	wird für jede Nachricht aufgerufen, die von der Anwendung empfangen wird.
OnMinimize	Sender: TObject	tritt bei jeder Minimierung des Hauptfensters bzw. der Anwendung auf.
OnRestore	Sender: TObject	tritt bei jeder Wiederherstellung eines minimierten Hauptfensters bzw. der Anwendung auf.
OnShortCut	var Msg: TMsg; var Handled: Boolean	tritt bei jeder Tastatureingabe vor dem Erzeugen des *OnKeyDown*-Ereignisses auf, eignet sich also zur Realisierung anwendungsweiter Tastenkürzel.
OnShowHint	var HintStr: string; var CanShow: Boolean; var HintInfo: THintInfo	tritt auf, wenn ein Hinweistext in einem kleinen Hinweisfenster gezeigt werden soll.

Viele weitere Themen von anwendungsglobaler Bedeutung werden nicht vom *Application*-, sondern vom *Screen*-Objekt behandelt, dem das folgende Kapitel gewidmet ist.

Application-Ereignisse als Komponente

Seit Delphi 5 finden Sie auf der Paletten-Seite ZUSÄTZLICH die Komponente *TApplicationEvents*, die Ihnen zunächst alle oben aufgeführten Ereignisse von *TApplication*

bequem im Objektinspektor zur Verfügung stellt, so dass Sie diese nicht mehr manuell im Quellcode mit Methoden verknüpfen müssen.

Eine weitere Besonderheit ist, dass Sie mehrere *ApplicationEvents*-Komponenten in Ihrer Anwendung verwenden können und dadurch *mehrere Bearbeitungsmethoden* für dasselbe Ereignis von *TApplication* schreiben können. Die VCL sorgt automatisch dafür, dass bei einem *TApplication*-Ereignis die dafür vorgesehenen Methoden aus *allen* existierenden *ApplicationEvent*-Objekten aufgerufen werden. (Im Gegensatz dazu können Sie ja bei einer direkten Verknüpfung zwischen Ereignis und Methode immer nur eine einzelne Methode mit einem Ereignis verknüpfen. Wenn Sie ein Ereignis direkt mit einer neuen Methode verknüpfen, hebeln Sie damit die alte Methode aus der Ereignisbearbeitung aus.)

Hinweis: Mit Hilfe eines eigenen *TApplicationEvents*-Objekts können auch Komponentenentwickler die Ereignisse von *TApplication* bearbeiten, unabhängig davon, welche Ereignisse der Entwickler, der die Komponente später in sein Formular einbaut, selbst bearbeiten will. Die Verwendung von *TApplicationEvents* setzt allerdings voraus, dass die gewünschten *TApplication*-Ereignisse *nur* über *TApplicationEvents*-Komponenten bearbeitet, also nicht etwa noch direkt mit einer Methode verknüpft werden. Verwendet also eine Komponente ein *TApplicationEvents*-Objekt, so muss auch der Benutzer der Komponente, der die Anwendungsereignisse bearbeiten will, diese mit Hilfe eines solchen Objekts bearbeiten.

3.2.4 TScreen

Obwohl der gesamte Bildschirm unter Windows mit einem Fenster-Handle ansprechbar ist, gelten die Hauptfenster und damit die Formulare nicht als Kindfenster des Bildschirms. Trotzdem ist hier eine gute Gelegenheit, einen Blick auf die Komponente *Screen* zu werfen, die in der Klasse *TScreen* definiert.

TScreen beschäftigt sich mit einer Reihe weiterer Themen, die global für die gesamte Anwendung gelten. Mehr noch als *TApplication* befasst sich *TScreen* dabei mit Dingen, die auf dem Bildschirm sichtbar sind. Auch einige wichtige Systeminformationen wie die Auflösung des aktuellen Grafikmodus, die installierten Schriften oder die in diesem Buch nicht weiter besprochenen IMEs (Eingabeeditoren für asiatische Schriftzeichen) erhalten Sie vom *Screen*-Objekt.

Besonders interessant sind die drei Properties *MenuFont*, *HintFont* und *IconFont*, die Sie auch ändern und mit denen Sie so das Erscheinungsbild Ihrer Anwendung anpassen können. Zwei davon (*MenuFont* und *HintFont*) gibt es erst seit Delphi 5, *IconFont* bereits seit Delphi 3.

Property	Typ	Beschreibung
ActiveControl	TWinControl	bezeichnet die Komponente, die innerhalb der Delphi-Anwendung den Tastaturfokus hat bzw. hatte, bevor die Anwendung deaktiviert wurde. Bei einem Fokuswechsel wird *ActiveControl* gesetzt, bevor das *OnExit*-Event der verlassenen Komponente aufgerufen wird.
ActiveCustomForm und ActiveForm	TCustomForm bzw. TForm	bezeichnet das oberste aller Formulare innerhalb der Delphi-Anwendung. Wenn die Delphi-Anwendung aktiv ist, ist *ActiveCustomForm* bzw. *ActiveForm* das aktive Fenster auf dem Windows-Desktop.
Cursor	TCursor	gibt einen Cursor für alle Fenster der Anwendung an und ist dafür gedacht, zeitweise alle anderweitigen Einstellungen (*Cursor*-Properties der Komponenten) zu überschreiben. Wenn *Cursor* auf *crDefault* gesetzt ist, bleiben die *Cursor*-Properties der Komponenten gültig.
Cursors	Cursor-Handle-Array, indiziert über *TCursor*	enthält die Cursor-Ressourcen zu jeder *cr...*-Konstante, siehe Kapitel 3.3.3.
CustomForms/ CustomFormCount	Integer/array of TCustomForm	beziehen sich auf die allgemeineren *TCustomForms*, ansonsten wie die Properties *Forms* und *FormCount*.
DataModuleCount	Integer	Zahl der in der Anwendung verwendeten *TDataModule*-Objekte
DataModules	array of TDataModule	enthält Verweise auf die einzelnen *TDataModule*-Objekte.
FormCount	Integer	Zahl der gerade bestehenden »normalen« Formulare der Anwendung (die Formulare, die von *TForm* abstammen)
Forms	array of TForm	enthält Verweise auf die bestehenden *TForm*-Formulare.
Fonts	TStrings	Stringliste, die alle im System installierten Schrifttypen beim Namen nennt
Height	Integer	Höhe des Bildschirms in Pixeln
HintFont	TFont	beschreibbares Property, gibt die in Hinweisfenstern verwendete Schriftart an (ab Delphi 5)
HintFont	TFont	Schriftart für die Tooltip-Hinweisfenster
IconFont	TFont	enthält die für die Beschriftung der Icons im Windows-Explorer eingestellte Schrift (beschreibbares Property, siehe auch *TControl.DesktopFont*).
MenuFont	TFont	beschreibbares Property, gibt die Schriftart für die Menüs der Anwendung an (ab Delphi 5).
MonitorCount	Integer	Zahl der an das System angeschlossenen Monitore

Property	Typ	Beschreibung
Monitors	array of TMonitor	für Betriebssysteme, die mehrere Monitore unterstützen: enthält Informationen über die einzelnen Monitore.
DesktopLeft, DesktopTop, DesktopHeight, DesktopWidth	Integer	Seit Windows 98 können mehrere Bildschirme zu einem großen virtuellen Bildschirm zusammengeschlossen werden. Diese Properties geben die Koordinaten der Darstellungsfläche des aktuellen Monitors innerhalb dieser virtuellen Fläche an.
PixelsPerInch	Integer	Auflösung des Bildschirms in Punkten pro Zoll, eingestellt vom Anwender oder eingeschätzt vom Betriebssystem, unter Windows typischerweise 96, unter Linux 75 ppi
Width	Integer	Breite des Bildschirms in Pixeln
WorkAreaLeft WorkAreaTop WorkAreaRight WorkAreaBottom	Integer	Koordinaten des Arbeitsbereichs des Bildschirms (ab Delphi 6; entspricht der gesamten Bildschirmfläche abzüglich der Taskleiste und ähnlicher vom Betriebssystem verwalteter Leisten)
WorkAreaRect	TRect	siehe WorkArea*-Einzelkoordinaten (ab Delphi 6)

Ereignisse

Schließlich finden Sie in *TScreen* auch noch zwei Events: *OnActiveControlChange* und *OnActiveFormChange*. Diese werden jedes Mal ausgelöst, wenn der Tastaturfokus innerhalb eines Formulars bewegt wird oder wenn ein anderes Formular aktiviert wird. Während die Komponentenereignisse wie *OnEnter* und *OnActivate* nur bei Aktivierung bestimmter Komponenten oder Formulare eintreten, gelten diese Ereignisse global für die gesamte Delphi-Anwendung. So können Sie z.B. bei jeder *OnActiveControlChange*-Nachricht den Hinweistext zum aktivierten Steuerelement in einer Statuszeile anzeigen und hätten so nicht nur für die Maus-, sondern auch für die Tastaturbedienung eine Hinweisfunktion.

3.3 Grundlegende Gemeinsamkeiten von Steuerelementen

Die verschiedenen Komponententypen haben viele übereinstimmende Properties, Methoden und Ereignisse, die auf die Basisklassen *TControl* und *TWinControl* zurückzuführen sind. Nach der allgemeinen Vorstellung dieser Klassen in Kapitel 3.1.3 beschäftigt sich dieser Abschnitt genauer mit deren Fähigkeiten.

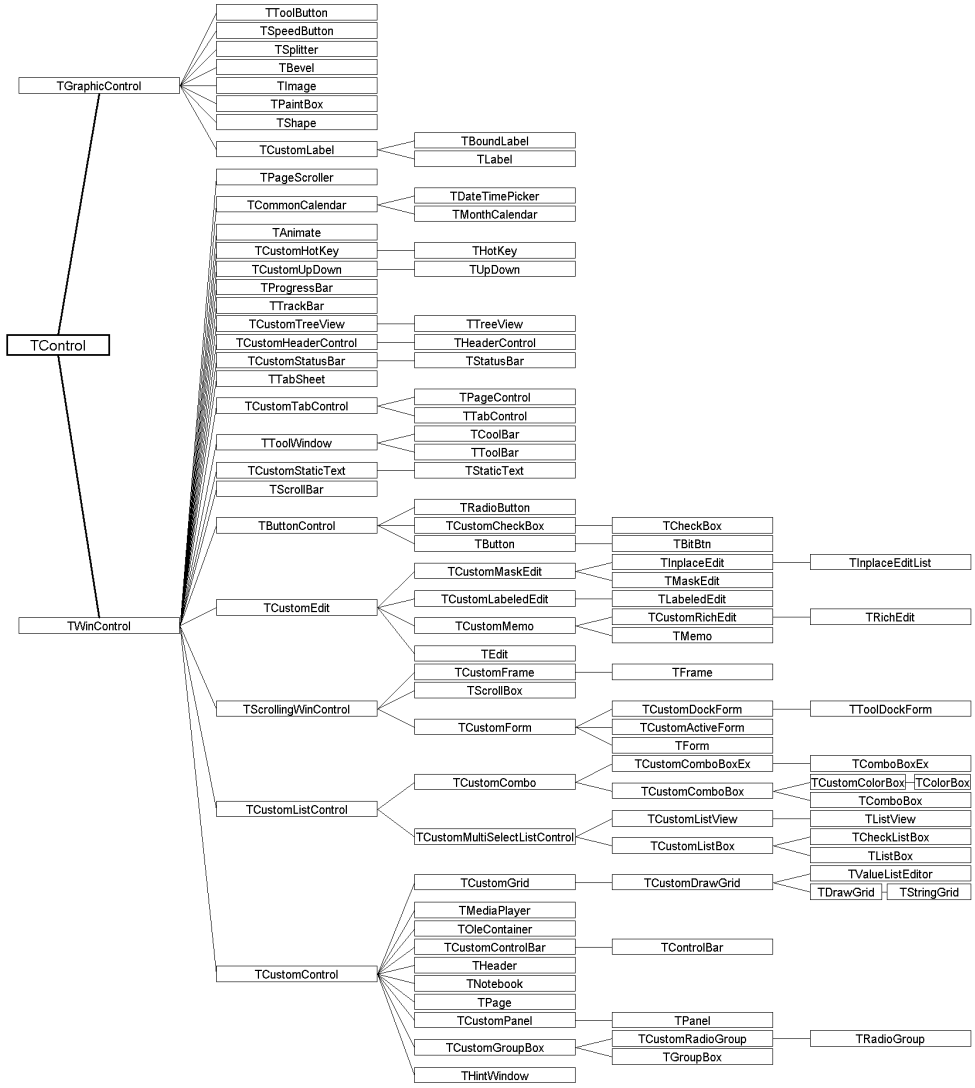


Abbildung 3.3: Die visuellen Komponenten befinden sich unter den von TControl abgeleiteten Klassen (die hier gezeigten Klassen stehen in allen Ausgaben von Delphi 6 zur Verfügung).

Reichweite der TControl- und TWinControl-Properties

TControl und *TWinControl* definieren einige Properties, die zwar in den Steuerelementen immer vorhanden, aber nicht immer veröffentlicht sind. So enthält *TControl* z.B. ein Property *Font* für die Schriftart, ohne dies zu veröffentlichen. Sowohl *TShape* als auch

TSpeedButton erben *Font* von *TControl*, aber nur *TSpeedButton* holt die Veröffentlichung nach, da *TShape* mangels Text gar keine Anwendungsmöglichkeit für eine Schriftart hat.

Das hat zur Folge, dass Sie im Objektinspektor nur das *Font*-Property von *TSpeedButton* verändern können, nicht aber das von *TShape*. Im Programmcode könnten Sie auch *TShape.Font* verändern, würden damit aber keine Wirkung erzielen.

Die meisten Properties, die zwischen verschiedenen Komponenten übereinstimmen, sind in *TControl* oder *TWinControl* definiert. Nur wenige Properties, die einen gleichen Namen haben, sind nicht auf eine gemeinsame Basisklasse zurückzuführen. So haben z.B. alle Steuerelemente ein *Caption*-Property, das sie von *TControl* erben. Da die Klasse *TMenuItem* jedoch nicht von *TControl* abgeleitet ist, muss sie für ihre Menüpunkte ein eigenes *Caption*-Property einführen.

3.3.1 Grundlegende Eigenschaften

Die Vielzahl an Properties, Methoden und Ereignissen, die bereits zum Standardvorrat aller (Fenster-)Elemente gehört, wird erst dann einigermaßen übersichtlich, wenn man sie nach bestimmten Funktionsbereichen gruppiert. Da Sie eine alphabetische Übersicht über *alle* Properties einer Komponente auch in Delphi entweder im Objektinspektor oder in der Online-Hilfe bekommen können, verzichtet dieses Kapitel auf eine tabellarische Aufstellung. Es berücksichtigt neben Events und Properties aber auch die im Objektinspektor nicht sichtbaren Methoden – je nachdem, was für einen bestimmten Zweck am ehesten benötigt wird.

Aktiviert und deaktiviert

Alle Steuerelemente sind per Voreinstellung aktiviert, Sie können sie jedoch auch deaktivieren, indem Sie das Property *Enabled* verwenden. Ein deaktiviertes Element (*Enabled=False*) ist zwar sichtbar, kann aber keine Eingaben vom Benutzer empfangen. Deaktivierte Schalter können also nicht gedrückt werden und deaktivierte Eingabefelder können nicht editiert werden.

Meistens dient die Deaktivierung von Elementen zur Vereinfachung der Bedienung für den Anwender, denn normalerweise zeigen die Elemente ihre Deaktivierung durch ein graues Erscheinungsbild an. So deaktiviert Delphi z.B. den Mauspalettenschalter zum Anhalten des Programms, wenn das Programm gar nicht läuft.

Wie in diesem Beispiel vermeidet die Deaktivierung unerlaubter Schalter oft unnötige Frustration des Benutzers, denn wenn der Schalter nicht gedrückt werden darf, müsste eine Zuwiderhandlung gegen diese Regel mit einer belehrenden Meldung geahndet werden.

In manchen Fällen ist die Deaktivierung von Steuerelementen sogar notwendig, um die völlige Verwirrung des Benutzers zu vermeiden, so z.B. wenn Steuerelemente nicht im sichtbaren Bereich des Bildschirms liegen, wie das in Kapitel 1.8.2 der Fall ist (nicht zu verwechseln mit Elementen, die vollkommen unsichtbar sind, da *Visible=False*).

Sichtbar und unsichtbar

Statt ein Steuerelement zu deaktivieren, könnten Sie es auch völlig unsichtbar machen, indem Sie das Property *Visible* auf *False* setzen. Allerdings dient das Unsichtbarmachen meistens anderen Zwecken, so zum Verstecken von Steuerelementen oder Fenstern, die zurzeit nicht gebraucht werden oder die der Benutzer nicht sehen *will*. So kann das Verstecken einer Mauspalette z.B. dazu dienen, mehr Arbeitsfläche auf dem Bildschirm zu schaffen. Derartige Maßnahmen mit der Mauspalette werden normalerweise auf Anweisung vom Benutzer getroffen.

Ein programmgesteuertes Verstecken und Sichtbarmachen von Komponenten wird z.B. dann benötigt, wenn ein Fenster zwar ständig existieren, aber nur bei Gelegenheit sichtbar sein soll. Und es geht schneller, ein Fenster zu verstecken und es später wieder anzuzeigen, als es freizugeben und später wieder ganz neu zu erzeugen. Hierbei ist jedoch grundsätzlich Vorsicht geboten, da auch unsichtbare Fenster viele wertvolle Windows-Ressourcen blockieren können.

Als Alternative zum Setzen des *Visible*-Properties können Sie sich auch der Methoden *Hide* und *Show* bedienen. Während *Hide* nichts anderes bewirkt als die Zuweisung *Visible:=False*, sorgt *Show* auch durch einen Aufruf der Methode *BringToFront* dafür, dass das Steuerelement in den Vordergrund geholt wird.

Hinweis: Ein Element, dessen *Visible*-Property *True* ist, kann auch unsichtbar sein, wenn das *Visible*-Property eines übergeordneten Elements gesetzt ist. Über die *Showing*-Methode eines *TControls* können Sie abfragen, ob nicht nur das Control selbst, sondern auch alle übergeordneten Elemente *Visible* sind. *Showing* gibt in diesem Fall *True* zurück, was aber noch nicht bedeutet, dass der Benutzer das Element wirklich sieht, denn es kann ja immer noch von anderen Komponenten oder Fenstern verdeckt sein.

Z-Reihenfolge von Steuerelementen *R19*

Ob sich eine Komponente im Vordergrund oder im Hintergrund befindet, hängt von der im Englischen als *Z-Order* bezeichneten Reihenfolge bezüglich einer gedachten Z-Achse ab, die aus dem Bildschirm herauszeigt und an der sich übereinander liegende Fenster auftürmen. Wie diese Reihenfolge zustande kommt, können Sie an den praktischen Erfahrungen beim Formularentwurf erkennen: Wenn Sie mehrere *TButton*-Komponenten überlappend einzeichnen, so liegt der zuletzt eingezeichnete Schalter immer

oben. Ändern können Sie diese Reihenfolge nur mit den lokalen Menübefehlen NACH HINTEN SETZEN und NACH VORNE SETZEN. Dieselbe Wirkung wie diese beiden Menübefehle haben die Methoden *TControl.BringToFront* und *TControl.SendToBack*. Diese Methoden funktionieren auf jeder Ebene von Komponenten, also auch für ganze Fenster. Mit *Form1.SendToBack* senden Sie ein ganzes Fenster hinter alle anderen Fenster auf dem Desktop, mit *Komponente.BringToFront* holen Sie eine einzelne Komponente vor alle seine Geschwisterkomponenten (die Komponenten, die dieselbe Elternkomponente bzw. dasselbe Elternfenster haben).

Koordinaten

Um die Koordinaten von Steuerelementen festzustellen oder zu ändern, haben Sie mehrere Möglichkeiten:

- Die Einzelwerte stehen in den Properties *Left*, *Top*, *Width* und *Height* zur Verfügung (*Left/Top* gibt die Koordinaten der linken oberen Ecke des Elements an, relativ zum übergeordneten Element, *Width* und *Height* enthalten die Höhe des Elements).
- Um alle vier Werte auf einmal zu ändern, brauchen Sie nicht umständlich vier Zuweisungen zu schreiben, sondern Sie können die Werte als Liste an die Methode *SetBounds* übergeben.
- Im *Bounds*-Property sind diese vier Einzelwerte zu einer *TRect*-Struktur zusammengefasst.
- Sofern Sie sich nur für den Arbeitsbereich eines Steuerelements interessieren, verwenden Sie statt *Height* und *Width* die Properties *ClientHeight* und *ClientWidth*. Oft stimmen diese mit erstgenannten Properties überein, da der Arbeitsbereich die gesamte Komponente umfasst. Formulare sind ein Beispiel, wo *ClientHeight/Width* kleiner sind als *Height/Width*, denn hier werden Titelzeile, Rahmen, Menü und eventuell weitere Teile des Fensters vom Arbeitsbereich abgezogen.
- Die Position des Arbeitsbereichs relativ zum übergeordneten Fenster ist im Property *ClientOrigin* als *TPoint* gespeichert. Im Gegensatz zu den bisher genannten Properties kann dieses nur gelesen werden.
- Im *ClientRect*-Property sind die Einzelwerte *ClientHeight* und *ClientWidth* in einer *TRect*-Struktur zusammengefasst, die für die linke obere Ecke immer 0/0 angibt, da die linke obere Ecke des Steuerelements *relativ zu sich selbst* gemeint ist. *ClientRect* kann nicht beschrieben werden, Änderungen sind nur indirekt durch Änderung der Position und Größe des gesamten Fensters oder durch Zuweisungen an *ClientHeight* und *ClientWidth* möglich.

TRect und TPoint

Die Strukturen *TRect* und *TPoint* werden bei verschiedenen Gelegenheiten verwendet, um Koordinatenangaben zusammenzufassen. So erhält beispielsweise die Grafikfunktion *PolyLine* ein *TPoint*-Array. Während *TPoint* einfach aus den *X*/*Y*-Koordinaten besteht, enthält *TRect* zwei Varianten, so dass Sie es als vier Einzelkoordinaten oder als zwei diagonal gegenüberliegende Punkte ansprechen können. Die beiden Typen sind wie folgt deklariert:

```
type
  TPoint = record
    X: Integer;
    Y: Integer;
  end;
  TRect = record
    case Integer of
      0: (Left, Top, Right, Bottom: Integer);
      1: (TopLeft, BottomRight: TPoint);
  end;
```

Manchmal ist es nützlich, *TPoint*- und *TRect*-Strukturen temporär zu konstruieren, ohne eine Variable dafür deklarieren zu müssen. Dies können Sie mit den Funktionen *Point* und *Rect*, denen Sie die Einzelkoordinaten übergeben und die eine *TPoint* bzw. *TRect*-Struktur daraus machen und zurückliefern. Ein Beispiel dazu folgt im nächsten Abschnitt.

Koordinatensysteme R15

Wie die oben genannten Properties zeigen, verfügt jedes Steuerelement über ein eigenes Koordinatensystem, das einfach dadurch definiert ist, dass die linke obere Ecke des Elements die Koordinaten 0/0 erhält und die Koordinaten nach rechts bzw. nach unten ansteigen. Mit Hilfe der bereits genannten Properties können Sie leicht Koordinatenangaben vom System eines Kindelements in das System eines Elternelements umrechnen (in Delphi 6 gibt es hierfür auch die Methoden *TControl.ClientToParent* und *ParentToClient*). Meistens ist es aber wichtiger, dass Sie die Koordinaten zwischen einem Element-Koordinatensystem und dem Bildschirm-Koordinatensystem umrechnen können. Hierfür stellt *TControl* zwei Methoden zur Verfügung:

- *ClientToScreen* erwartet als Parameter eine Koordinatenangabe als *TPoint*-Struktur und wandelt diese in Bildschirmkoordinaten um.
- *ScreenToClient* arbeitet umgekehrt und wandelt Bildschirmkoordinaten in Steuerelement-Koordinaten um.

Wenn Sie beispielsweise wissen wollen, an welcher Position des Bildschirms sich die linke obere Ecke einer Komponente befindet, können Sie *ClientToScreen* aufrufen:

```
BildschirmPos := ClientToScreen(Point(Komponente.Left, Komponente.Top));
// was im Prinzip dasselbe ist wie:
BildschirmPos := ClientToScreen(Point(Komponente.Bounds.TopLeft));
```

Hinweis für API-Kenner: Wenn Sie bereits mit den gleichnamigen API-Funktionen von Windows gearbeitet haben, sollten Sie beachten, dass die Delphi-Methoden das Ergebnis als Funktionswert zurückliefern und nicht den Parameter verändern. Wenn Sie vergessen, das Ergebnis einer Variablen zuzuweisen, hat der Aufruf dieser Methoden überhaupt keinen Effekt.

Textinhalt der Steuerelemente

Viele Steuerelemente haben einen Textinhalt, der als Überschrift dienen (z.B. bei *TGroupBox*), das eigentliche Element ausmachen (z.B. bei *TLabel*) oder auch zur Benutzereingabe dienen kann (z.B. bei *TEdit*). Trotz der Verschiedenheit der Steuerelemente müssen Sie sich für den Textinhalt nur die ersten beiden der folgenden Properties merken:

- Wenn der Text nicht vom Benutzer geändert werden kann, steht er im Property namens *Caption*. Dies ist z.B. der Fall bei den Komponenten *TLabel*, *TButton*, *TPanel* und auch bei Formularen und Menüpunkten. Üblicherweise können Sie im *Caption*-Text erreichen, dass eines der Zeichen unterstrichen dargestellt wird, indem Sie ein »&« vor dem zu unterstreichenden Zeichen angeben.
- Vom Benutzer änderbarer Text ist über das Property *Text* zugänglich, z.B. in *TEdit* und *TComboBox*.
- Als Alternative zu *Caption* und *Text* gäbe es theoretisch noch *WindowText*. Allerdings handelt es sich hierbei um einen nullterminierten String, der nicht so bequem zu handhaben ist wie die Strings von *Caption* und *Text*. *WindowText* gibt den String an, den das Betriebssystem als »Fenstertext« bezeichnet. Dies entspricht in nicht editierbaren Elementen meist dem Property *Caption*, ansonsten dem Property *Text*. Im Zusammenhang mit dem Property *Text* sei außerdem auf die Online-Hilfe zu den Methoden *GetTextBuf*, *SetTextBuf* und *GetTextLen* verwiesen.

Align

Jedes Fenster der VCL verfügt über eine Funktion zum automatischen Anordnen seiner Unterfenster bzw. Steuerelemente bei jeder Änderung seiner eigenen Größe. Die Anordnung der Steuerelemente ist von den Einstellungen der *Align*-Properties abhängig. Per Voreinstellung haben diese den Wert *alNone*, was bedeutet, dass die Steuerelemente ihre Position und Größe nicht verändern.

Die anderen Werte von *Align* funktionieren wie folgt: Bei der Erstellung des Formulars und bei jeder nachfolgenden Größenänderung ordnet das Formular zuerst die Elemente, die die Werte *alLeft*, *alRight*, *alTop* oder *alBottom* haben, an seinen Rändern an (links, rechts, oben oder unten). Dabei werden Komponenten, die den gleichen *Align*-Wert haben, nebeneinander bzw. untereinander angeordnet. Der Bereich, der dann noch frei bleibt, wird den Komponenten mit der Ausrichtung *alClient* zugeordnet. Normalerweise ist das nur eine Komponente, da sich mehrere Komponenten sonst gegenseitig überdecken würden. Jedes Fenster, das Elternfenster ist, ordnet seine Unterfenster auf dieselbe Weise an, daher können Sie auch die Elemente innerhalb eines Panels auf diese Weise ausrichten.

Verwendung von Align zur automatischen Größenanpassung

Viele Beispielprogramme dieses Buchs verwenden das *Align*-Property zur automatischen Anpassung von Komponenten an die Größenänderung des Hauptfensters, z.B. das ListView-Demoprogramm in Kapitel 3.6.2 (Abbildung 3.12) und der CD-Player in Kapitel 3.6.4 (Abbildung 3.15). Sie verfügen jeweils über ein oder mehrere Panels, die mit *alTop* (oder *alLeft*, *alRight*, *alBottom*) ausgerichtet sind. Diese Panels verändern ihre Höhe (Breite) nicht, passen ihre Breite (Höhe) aber der des Fensters an. Ein Panel, das beliebig vergrößert werden kann, belegt den Hauptteil der Fenster und ist mit *alClient* ausgerichtet. Innerhalb einiger dieser Panels befindet sich als Kindfenster des Panels eine Komponente, die mit *alClient* ausgerichtet ist, die dann den gesamten Bereich des Panels einnimmt und damit alle Breiten- bzw. Größenänderungen desselben mitmacht. Um schließlich einen Rand um diese Komponenten zu lassen, wurde das *BorderWidth*-Property des Panels auf einen höheren Wert gesetzt.

Hinweise: Sie können die automatische Ausrichtung von Komponenten zeitweise deaktivieren und »von Hand« steuern, um beispielsweise während einer größeren programmgesteuerten Veränderung im Fensteraufbau eine chaotische Bildschirmanzeige zu verhindern. Siehe hierzu die Online-Hilfe zu den Methoden *TWinControl.EnableAlign*, *TWinControl.DisableAlign* und *TWinControl.Realign*.

Wenn Sie eine ganz andere als die vorgegebene Ausrichtungsfunktion benötigen, können Sie das Ereignis *OnResize* des Formulars selbst bearbeiten und die Größen seiner Elemente wie gewünscht anpassen.

TSplitter *R3*

Eine sehr nützliche Komponente ist *TSplitter*; sie sorgt dafür, dass Sie die Größe eines mit *alLeft*, *alTop*, *alRight* oder *alBottom* ausgerichteten Fensterbereichs zur Laufzeit so flexibel verändern können wie zur Entwurfszeit. Wie schon erwähnt, kann von einer

am Fensterrand (oder am Rand einer anderen Komponente) ausgerichteten Komponente immer nur eine Seite verschoben werden. *TSplitter* besteht nun aus einem Balken, der genau an dieser Seite der Komponente angelegt wird.

Nachdem Sie die in der Größe zu verändernde Komponente in das Formular eingefügt haben, fügen Sie einfach ein Exemplar von *TSplitter* hinzu (Seite *Zusätzlich* der Komponentenpalette) und setzen sein *Align*-Property auf denselben Wert wie bei der Komponente, die verändert werden soll. Dadurch erhalten Sie bereits den gewünschten Fensteraufbau, alles weitere erledigt die VCL zur Laufzeit automatisch.

Der Komplexität der Fensterstruktur, die Sie mit *TSplitter*-Komponenten einteilen können, sind theoretisch keine Grenzen gesetzt, denn jeder durch eine *TSplitter*-Komponente geteilte Fensterbereich kann horizontal und vertikal in weitere Teile geteilt werden (falls es sich bei diesem Bereich um eine Komponente handelt, die Kindelemente aufnehmen kann, wie z.B. eine *TPanel*-Komponente oder einfach einen Teilbereich des Formulars).

Drei besondere Properties runden die Fähigkeiten der Splitter-Komponente ab: Durch *ResizeStyle* können Sie unter anderem erreichen, dass die veränderten Fensterteile während des Verschiebens des Splitters ständig neu gezeichnet werden (*ResizeStyle = rsUpdate*). Das Property *AutoSnap* bewirkt, wenn es eingeschaltet ist, Folgendes: Wird eine bestimmte Minimalgröße der mit *alLeft*, *alTop*, *alRight* oder *alBottom* ausgerichteten Komponente unterschritten, so verschwindet diese Komponente ganz, indem ihre Größe auf Null gesetzt wird. Die Minimalgröße wird über das dritte Property angegeben: *MinSize* (*AutoSnap* und *MinSize* ab Delphi 5).

TSplitter wird z.B. im Beispielprogramm *RegBrows* von Kapitel 4.2.2, im *ShellExplorer* aus Kapitel 8.4 und für den Andockbereich an der rechten Seite des TreeDesigners-Hauptfensters verwendet (Kapitel 5.8.2).

3.3.2 Maus- und Tastatureingaben

In einfachen Dialogen müssen Sie sehr selten Maus- und Tastatureingaben direkt bearbeiten, da die Komponenten bereits alle wichtigen Eingaben zu *OnClick*- und *OnChange*-Ereignissen umformen. Dieses Kapitel beschreibt die acht Standard-Events, mit denen Sie Maus- und Tastatureingaben direkt abfragen können; Praxisbeispiele dazu gibt Kapitel 5.8.1 mit dem TreeDesigner.

Der Tastaturfokus

Während die Maus sich sehr frei über den Bildschirm bewegen kann, ist die Tastatur darauf angewiesen, dass ihre Eingaben immer an einen klar definierten Ort gelangen. Für den Benutzer ist dieser Ort meistens an der blinkenden Eingabemarkierung zu erkennen (auch Cursor oder *Caret* genannt). Im Programm wird diese Zielrichtung der

Tastatureingabe als *Tastaturfokus* bezeichnet. Das Fenster oder das Steuerelement, das die Eingaben erhält, *besitzt* den Tastaturfokus. In der Terminologie der VCL wird dieses Steuerelement auch als *aktives Element* bezeichnet, es befindet sich beispielsweise in den *ActiveControl*-Properties seines übergeordneten Formulars und des *Screen*-Objekts.

Wie bereits in Kapitel 3.1.3 erwähnt, können nur die Komponenten, die von *TWinControl* abstammen, den Tastaturfokus erhalten. Jedes Mal wenn der Tastaturfokus zu einem anderen Element übergeht, sei es durch einen Mausklick, durch die Tastaturbedienung oder durch die Steuerung des Programms, erhält die Komponente, die den Tastaturfokus verliert, ein *OnExit*-Event; die Komponente, die ihn erhält, wird mit einem *OnEnter*-Ereignis benachrichtigt.

Es zählt nicht als Wechsel des Tastaturfokus, wenn Sie zwischen verschiedenen Formularen wechseln. Solange ein Formular nicht aktiv ist, hat sein aktives Element (*ActiveControl*) zwar nicht den Tastaturfokus, sobald das Formular aber wieder aktiv wird, erhält es diesen zurück. Daher löst der reine Wechsel zwischen verschiedenen Formularen keine *OnEnter* / *OnExit*-Ereignisse beim Steuerelement, sondern nur *OnActivate* / *OnDeactivate*-Events beim Formular aus.

Tastatureingaben *R78*

Die VCL erlaubt Ihnen eine sehr genaue Abfrage der Tastatureingaben. So können Sie nicht nur mit *OnKeyPress* auf normale Zeicheneingaben reagieren, sondern mit den Ereignissen *OnKeyDown* und *OnKeyUp* sogar das Herunterdrücken und das Loslassen jeder einzelnen Taste erfahren. Der Unterschied wird deutlich, wenn wir untersuchen, welche Ereignisse bei der Eingabe des Großbuchstabens »A« auftreten:

- Sofern der Benutzer nicht die Feststelltaste verwendet, muss er zunächst ein *OnKeyDown* mit [Shift] auslösen.
- Für den Druck auf [A] gibt es ein weiteres *OnKeyDown*-Ereignis.
- Aus dem [A] und dem Zustand der Umschalt-Tasten kann die CLX bzw. Qt nun ein *OnKeyPress*-Ereignis machen für das ANSI-Zeichen »A« .
- Danach erfolgen noch zwei *OnKeyUp*-Ereignisse in einer vom Tastaturbenutzer abhängigen Reihenfolge.

Mit *OnKeyPress* können Sie alle Eingaben, die aus einem ANSI-Zeichen bestehen, bearbeiten. Die Bearbeitungsmethode für dieses Ereignis erhält dieses Zeichen im Parameter *Key*. Wenn Sie allerdings Funktionstasten, andere Steuertasten oder gar [Shift], [Strg] und weitere umschaltbare Tasten abfragen wollen, müssen Sie die anderen beiden Ereignisse bearbeiten.

OnKeyDown und *OnKeyUp* erhalten jeweils die gleichen Parameter:

- In *Key* ist diesmal kein ANSI-Zeichen, sondern ein 16 Bit großer Zeichencode gespeichert. Die Liste der möglichen Codes finden Sie in der Win32-Online-Hilfe unter dem Stichwort *Virtual-Key Codes*.
- In *Shift* ist der Status der Tasten Shift, Alt und Strg gespeichert. *Shift* besteht aus einer Menge, die die folgenden Elemente enthalten kann: *ssShift, ssAlt, ssCtrl, ssRight, ssLeft, ssMiddle, ssDouble*. Die letzten vier geben den Status der Maustasten an, wobei *ssDouble* nur bei den Mausereignissen verwendet wird.

Um also beispielsweise die Tastenkombination Shift+F3 abzufangen, können Sie die folgende Abfrage verwenden:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (Key = VK_F3) and (ssShift in Shift) then
    ...
```

Hinweis: Falls Sie am Drücken und Loslassen einzelner Tasten interessiert sind, brauchen Sie im Programm nicht zu speichern, welche Tasten gerade gedrückt sind, denn Windows führt ebenfalls darüber Buch und gibt Ihnen mit den API-Funktionen *GetAsyncKeyState, GetKeyState, GetKeyboardState* Auskunft darüber.

Mauseingaben

Wie bereits in Kapitel 1.5.1 demonstriert, gibt es »komplexe« Mausnachrichten, die besagen, dass die Maustaste einmal gedrückt und wieder losgelassen wurde (*OnClick*) oder gleich zweimal innerhalb kurzer Zeit hintereinander (*OnDblClk*), und einfache Mausnachrichten, die besagen, dass eine Taste gedrückt *oder* losgelassen wurde (*OnMouseDown* und *OnMouseUp*) oder dass die Maus bewegt wurde (*OnMouseMove*).

Die letzteren drei Ereignisse geben Ihnen auch Informationen über die Position der Maus und über den Status einiger wichtiger Tasten (der Parameter *Shift* stimmt mit dem *Shift*-Parameter der Tastaturereignisse überein). Ein Beispiel für *OnMouseMove* folgt auf Seite 352, ein Beispiel für die Abfrage aller drei einfachen Mausnachrichten finden Sie in Kapitel 5.4.1.

Weitere Mausfunktionen

In die Klasse *TControl* eingebaut ist auch Unterstützung für weitere wichtige Mausfunktionen:

- Popup-Menüs, die automatisch angezeigt werden können, wenn der Benutzer die rechte Maustaste drückt (Property *TControl.PopupMenu*).

- Der Start ist auch bei Drag&Drop automatisch möglich, im weiteren Verlauf müssen Sie aber ein paar Ereignisse bearbeiten. *TControl* besitzt zwei Properties (*DragCursor* und *DragMode*) und ein paar Methoden (*Dragging*, *BeginDrag* und *EndDrag*), die mit dem einfachen Drag&Drop in Zusammenhang stehen.
- Mit Delphi 4 kamen im Zuge der Docking-Funktionalität weitere Elemente zu *TControl* hinzu.

Alle Themen werden an anderer Stelle dieses Buchs ausführlich und mit Beispielen behandelt: Popup-Menüs in Kapitel 5.2.6, Drag&Drop in Kapitel 5.8.3 und Docking in den Kapiteln 5.2.2 und 5.8.2.

3.3.3 Aktionen beim Bewegen der Maus

Benutzerfreundliche Anwendungen warten nicht erst darauf, dass der Anwender irgendwelche Eingaben tätigt oder Elemente anklickt, sie reagieren bereits auf die *Bewegung* der Maus. Neben dem automatischen Anpassen der Mauszeigerform unterstützt die VCL automatisch angezeigte Hinweisfenster.

Mauszeigerformen

Auf grafischen Benutzeroberflächen ist es üblich, dass der Mauszeiger sensibel auf den Fensterbereich, auf den er zeigt, reagiert. Im Luftraum des Delphi-Editors nimmt er beispielsweise die Form des üblichen Textcursors *crIBeam* an, während er sich in einen doppelseitigen Pfeil verwandelt, sobald er über einem veränderbaren Fensterrahmen schwebt.

Die Komponenten der VCL machen es Ihnen besonders leicht, die Mauszeigerform zu verändern. Bereits die Klasse *TControl* deklariert das Property *Cursor*, in dem Sie eine Mauszeigerform angeben können, die immer dann aktiv wird, wenn sich die Maus über dem Steuerelement befindet. Der normale Mauszeiger und die Standardeinstellung des *Cursor*-Properties hat den Namen *crDefault*. Auch die anderen Mauszeigerformen werden über eine *cr...*-Konstante angesprochen, z.B. *crHourGlass* für die Sanduhr-Form. Diese Konstante hat nichts mit den Cursor-Handles zu tun, mit denen Sie das Windows-API konfrontiert, sondern weist in das *Cursors*-Array des *Screen*-Objekts.

Das Cursor-Array

Die verschiedenen verwendeten Mauszeigerformen schwirren in einer Delphi-Anwendung nicht als lose Einzelteile herum, sondern sind an einem passenden Ort aufbewahrt und zu einem Array zusammengefasst. Wie in Kapitel 3.2.4 schon erwähnt, handelt es sich um das Property *Cursors* der *TScreen*-Klasse, zu der es ein globales Objekt namens *Screen* gibt. *Cursors* enthält zu jeder *cr...*-Konstante, die Sie den *Cursor*-Properties zuweisen können, den Cursor, der für diese Konstante verwendet wird. Die

Elemente des *Cursors*-Arrays sind nun Handles auf echte Windows-Mauszeigerformen. Noch ohne von diesen Handles Gebrauch zu machen, können Sie z.B. die Bedeutung der Cursor-Konstanten *crIBeam* wie folgt verändern:

```
Screen.Cursors [crIBeam] := Screen.Cursors [crSize];
```

Nach der Ausführung dieser Zeile verwendet Ihr Programm immer, wenn Sie *crIBeam* angeben, den Cursor, der auch unter der Bezeichnung *crSize* ansprechbar ist. Eine sinnvollere Anwendungsmöglichkeit von *Screen.Cursors* ist jedoch die Definition eigener Cursorformen.

Eigene Mauszeigerformen *R14*

Viele Anwendungen begnügen sich nicht mit den vordefinierten Windows-Cursorn, sondern definieren ihre eigenen Mauszeigerformen, die dem Benutzer besonders deutlich machen, was er mit der Maus gerade tun kann. Intensiven Gebrauch von eigenen Zeigerformen macht beispielsweise Delphis Bildeditor. Wenn Sie das Füllwerkzeug oder das Vergrößern-Werkzeug auswählen, erhält der Mauszeiger über der Zeichenfläche eine völlig andere Form als bei normalen Werkzeugen.

Um eine eigene Mauszeigerform verwenden zu können, müssen Sie zuerst eine Cursor-Ressource erstellen (beispielsweise mit dem Bildeditor) und mit der *$R*-Compileranweisung in Ihre Anwendung einbinden. Sie können dann die API-Funktion *LoadCursor* aufrufen, um die Ressource zu laden. Daraufhin können Sie diese in das *Cursors*-Array schreiben, und zwar an einen von Ihnen gewählten positiven Index (die vordefinierten Cursorform-Konstanten *cr...* sind negative Zahlen). Um den Cursor auch über einer Komponente anzeigen zu lassen, geben Sie genau diesen Index in deren *Cursor*-Property an. Ein Beispiel hierfür gibt der TreeDesigner aus Kapitel 5, der sich in den folgenden Zeilen mit der Mauszeigerform befasst:

```
const
  crDrawCursor = 1;
(* Ausschnitt aus TDocumentForm.FormCreate: *)
  Screen.Cursors[crDrawCursor] :=
    LoadCursor(HInstance, 'DRAWCURSOR');
(* Ausschnitt aus TDocumentForm.SetCurShapeType: *)
  PaintBox.Cursor := crDrawCursor;
```

Sie dürfen einen im *Cursors*-Array gespeicherten Cursor nicht mit der sonst üblichen API-Funktion *DestroyCursor* löschen, da die VCL dies automatisch übernimmt.

Weitergehende Cursor-Differenzierung *R13*

Ein weiteres Beispiel des Bildeditors zeigt auch die Grenzen der Mauszeiger-Automatik der VCL auf. Wenn Sie mit dem Markieren-Werkzeug des Bildeditors einen Bereich markieren, können Sie diesen mit der Maus verschieben. Der Mauszeiger wird zu einer Hand, sobald die Maus den markierten Bereich erreicht hat.

Wenn Sie – wie der Bildeditor im Fall von markierten Bildausschnitten – die Maus in verschiedenen Bereichen *einer* Komponente verschieden aussehen lassen wollen, müssen Sie die *OnMouseMove*-Nachrichten selbst bearbeiten und je nach Mausposition selbst die Cursorform anpassen. Wenn beispielsweise der Mauszeiger bis zu einer Entfernung von 100 Pixeln vom linken Rand des Formulars die Form eines Fadenkreuzes haben soll, im Rest des Formulars aber die Standardform, könnten Sie die folgende *OnMouseMove*-Bearbeitung schreiben:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState;
  X,  Y: Integer);
begin
  if x < 100 then Cursor := crCross
    else Cursor := crDefault;
end;
```

Hinweisfenster

Eine weitere Funktion, die mit der Bewegung der Maus zusammenhängt, sind die Hinweisfenster, die auch die Delphi-IDE verwendet, um Ihnen kurze Informationen über die Bedeutung der Schalter unter der Menüzeile zu geben. Auch diese Funktion können Sie bereits durch Setzen von nur zwei Properties einschalten. Geben Sie dazu lediglich den Hinweistext im Property *Hint* eines beliebigen Steuerelements an, schalten Sie die Anzeige der Hinweise an (*ShowHint*-Property des Formulars=*True*), und die VCL zeigt zur Laufzeit automatisch einen Hinweistext für das gewählte Element an. Die Verwendung des *Hint*-Properties zur Angabe zweier Hinweisversionen und die Anzeige von Hinweisen in der Statuszeile wurden bereits in Kapitel 1.9.5 erläutert, ebenso wie das Aktivieren und Deaktivieren der Hinweisfunktion für nur einige Komponenten des Formulars.

Weitere Optionen für Hinweistexte sind die Properties *HintPause*, *HintShortPause*, *HintHidePause*, *HintFont*, *HintColor* und *HintShortCuts* von *TApplication* (siehe Kapitel 3.2.3).

3.3.4 Anzeigesteuerung

Neben den schon in Kapitel 3.3.1 genannten wichtigen Properties befasst sich eine Vielzahl von weiteren *T(Win)Control*-Properties mit dem äußeren Erscheinungsbild der Komponenten:

- *Color* bestimmt die Farbe des Steuerelements, falls *ParentColor=False*. Im Objektinspektor können Sie entweder eine der vordefinierten Farbkonstanten aus einer Liste oder eine beliebige Farbe aus dem per Doppelklick zu öffnenden Farbauswahldialog auswählen (beachten Sie auch den erfrischenden Farbton der in Delphi 6 neu eingeführten Farbkonstante *clSkyBlue*). Zu *TColor*, dem Typ dieser Werte, finden Sie in Kapitel 4.4.2 mehr Informationen.

- *Font* gibt die Schriftart an, die für den *Text* und/oder die *Caption*-Beschriftung verwendet werden soll. *Font* ist ein Objekt des Typs *TFont*. Im Objektinspektor erhalten Sie leicht einen Überblick über die zur Verfügung stehenden Attribute wie z.B. Schriftgröße, Farbe, Fett-, Kursivschrift und Unterstreichung. Die Voreinstellung von *Font* ist die Systemschriftart für Fenstertext, die Sie in der Systemsteuerung von Windows einstellen und in Delphi auch über *Screen.IconFont* abfragen können (zu *TScreen* siehe Kapitel 3.2.4).
- *DesktopFont:* Wenn Sie dieses Property auf *True* setzen und auch *Application.UpdateMetricSettings* auf *True* gesetzt ist, wird *Font* bei jeder Änderung der Systemschriftarten automatisch aktualisiert.
- Wenn Sie *ParentColor* auf *True* setzen, ignoriert das Steuerelement den Wert seines eigenen *Color*-Properties und übernimmt statt dessen die Farbe des Elternelements (bei dem ebenfalls *ParentColor* auf *True* gesetzt sein könnte, mit der entsprechenden logischen Wirkung).
- *ParentFont* entspricht dem Property *ParentColor*, bezieht sich jedoch auf die Schriftart des Elements.
- Das Property *Brush* legt üblicherweise fest, mit welchem Pinsel die Hintergrundfarbe des Steuerelements gezeichnet werden soll. Es ist erst in *TWinControl* deklariert und wird von dort an die Nachkommen weitervererbt. Nachkommen der Klasse *TControl* müssen bei Bedarf ihr eigenes *Brush*-Property deklarieren. Zu den Untereigenschaften eines Pinsels siehe Kapitel 4.4.2.
- Auch *Ctl3D* ist erst in *TWinControl* deklariert. Es legt fest, ob das Steuerelement im 3D-Stil oder im flachen Stil gezeichnet werden soll. Bedenken Sie aber, dass mit »flacher Stil« hier nicht der moderne flache Stil gemeint ist, der unter Windows 98 wieder anzutreffen ist, sondern der alte flache Stil, der in der Entwicklung der Windows-Mode dem 3D-Stil voranging.
- Passend zu *Ctl3D* gibt es auch wieder ein Property *ParentCtl3D*, durch das Sie erreichen können, dass sich das Element in seiner 3D-Darstellung ganz nach seinem übergeordneten Element richtet.
- In Delphi 6 wurde *TWinControl* um Properties erweitert, mit denen Sie die dreidimensionale Komponentenumrandung optisch weiter verfeinern können: *BevelEdges* (lässt Sie die vier Kanten einzeln von den Verfeinerungsmaßnahmen ein- und ausschließen) sowie *BevelInner, BevelOuter, BevelKind* und *BevelWidth* zur Festlegung der Umrandungsart.

Neuzeichnen von Elementen *R93*

Wenn Sie das Aussehen der Komponente modifizieren, beispielsweise indem Sie eines der gerade genannten Properties ändern, muss die Komponente neu gezeichnet werden. Üblicherweise sorgt die VCL automatisch dafür, indem sie die *Invalidate*-Methode aufruft. *Invalidate* ist eine von vier *TControl*-Methoden, die Sie in manchen Fällen vielleicht auch direkt aufrufen wollen, beispielsweise wenn Ihnen die automatische Neuausgabe durch *Invalidate* zu lange dauert:

- *Invalidate* erklärt die Komponente für ungültig, was dazu führt, dass sie neu gezeichnet wird, wenn die in der Nachrichtenschlange der Anwendung wartenden Nachrichten abgearbeitet sind.
- *Repaint* zeichnet den Fensterinhalt sofort neu.
- *Refresh*: wie *Repaint*, allerdings ohne vor dem Neuzeichnen den bestehenden Fensterinhalt zu löschen.
- *Update*: wie *Refresh*, allerdings werden nur ungültige Bereiche neu gezeichnet (ähnlich wie *Invalidate* den gesamten Bereich für ungültig erklären kann, können Fensterbereiche auch nur teilweise für ungültig erklärt werden).

Da die meisten Anwendungen die eintreffenden Eingaben und Nachrichten sehr schnell bearbeiten, entsteht bei der Aktualisierung des Fensterinhalts normalerweise keine störende Verzögerung, so dass Sie wahrscheinlich selten in den Standardablauf (Aufruf von *Invalidate*) eingreifen müssen. Mehr Informationen zum Ungültig-Erklären von Fensterbereichen finden Sie in den Kapiteln 5.5.1 und 5.5.3.

3.3.5 Kontrolle über Größe und Position

Die als Nächstes untersuchten Properties von *TControl* definieren auf verschiedene Weise Regeln zur Größeneinstellung und -veränderung von Steuerelementen oder Formularen (alle sind sie erst ab Delphi 4 verfügbar).

Beschränkungen für die Größenänderung

Das *Constraints*-Property ist seinerseits wieder ein Objekt, was sich im Objektinspektor darin zeigt, dass es aus vier Unter-Properties besteht, die die minimale und die maximale Breite und Höhe des Steuerelements angeben: *MaxHeight*, *MaxWidth*, *MinHeight* und *MinWidth*. Solange Sie diese bei ihrer Voreinstellung belassen, darf das Steuerelement unbegrenzt verkleinert oder vergrößert werden. Wenn Sie aber einen dieser Werte auf eine Zahl größer 0 setzen, wird diese Vorgabe unerbittlich eingehalten: So ist es beispielsweise schon zur Entwurfszeit nicht mehr möglich, ein Formular breiter zu machen, als in *Constraints.MaxWidth* festgelegt.

Bei einigen Komponenten haben Sie auch die Möglichkeit, Größenbeschränkungen dynamisch festzulegen, indem Sie das Ereignis *OnConstrainedResize* mit einer Methode bearbeiten, die neue Minimal- und Maximalmaße zurückgibt. Beim Ereignis *OnCanResize* können Sie eine unmittelbar bevorstehende Größenveränderung durch direkte Größenangaben beeinflussen oder ganz unterbinden. Beide Ereignisse gibt es beispielsweise für *TForm* und *TPanel*.

Diese dynamischen Größenbeschränkungen haben den Vorteil, dass Sie darin leichter auf eventuelle Unterschiede zwischen verschiedenen Systemkonfigurationen reagieren können. Schon durch Wahl einer anderen Menü-Schriftart (*TScreen.MenuFont*) kann sich z.B. die Höhe der Menüzeile ändern. Da sich *Constraints.MaxHeight* jedoch auf die Gesamthöhe des Fensters bezieht, verkleinert sich in dieser Situation bei fester *MaxHeight* die Höhe des Arbeitsbereichs.

Regeln für automatische Größen- und Positionsanpassung

Bevor man mit Hilfe des *Constraints*-Property die Veränderlichkeit der Formulargröße einschränkt, sollte man überlegen, ob das Formular nicht auch so gestaltet werden kann, dass die Größenänderung Sinn macht – z.B. dadurch, dass sich auch die im Formular enthaltenen Komponenten dieser Größe anpassen. Hierbei kann das *Anchors*-Property helfen (als eine spezielle Alternative zum bereits in Kapitel 3.3.1 besprochenen und wahrscheinlich häufiger verwendeten *Align*-Property).

Anchors ist ein Mengen-Property mit vier möglichen Elementen*: akLeft, akRight, akTop* und *akBottom*. Jedes davon bewirkt, dass eine Seite des Steuerelements an einer Seite des Elternelements verankert wird. »Verankerung« bedeutet, dass der Abstand zwischen zwei bestimmten Begrenzungen von Eltern- und Kindelement gleich bleibt. Die Voreinstellung von *akLeft* und *akTop* besagt beispielsweise, dass der Abstand zwischen der linken oberen Ecke der Elternkomponente und der linken oberen Ecke der Kindkomponenten konstant gehalten wird. *AkRight* und *akBottom* können in gleicher Weise dazu verwendet werden, einen konstanten rechten oder unteren Rand zur Elternkomponente zu bestimmen.

Mit Hilfe von *Anchors*-Einstellungen ist es möglich, ohne die Verwendung des *Align*-Properties die Fläche eines Fensters in jeder Größeneinstellung voll auszunutzen, jedoch ziehen die Beispielprogramme dieses Buchs dank seiner intuitiveren Funktionsweise das *Align*-Property vor.

Automatische Größeneinstellung

Das neue *AutoSize*-Property von *TControl* ist nur bei ausgewählten Komponenten veröffentlicht. Seine Wirkung können Sie testen, indem Sie eine *TPanel*-Komponente auf ein leeres Formular platzieren, *AutoSize* auf *True* setzen und eine neue Komponente in

das Panel setzen: Sofort schrumpft das Panel auf die Größe dieser Komponente zusammen. Wenn Sie mehrere Komponenten im Panel unterbringen wollen, sollten Sie *AutoSize* daher erst setzen, wenn alle Komponenten eingefügt sind.

Ein sinnvoller Anwendungszweck für *AutoSize* sind alle Platzhalterkomponenten, die dazu dienen, zur Laufzeit andere Komponenten aufzunehmen, so zum Beispiel Panels, die zum Andocken von Fenstern oder Toolbars gedacht sind. Jedes Mal, wenn sich die Größe der Kindelemente ändert oder wenn Kindelemente hinzugefügt oder entfernt werden, berechnet eine Komponente, bei der *AutoSize* eingeschaltet ist, ihre Größe neu. Im TreeDesigner wird *AutoSize* für das Andock-Panel auf der rechten Seite des Hauptfensters verwendet (siehe Kapitel 5.8.2).

3.3.6 TWinControl

Das meiste, was *TWinControl* der Klasse *TControl* hinzufügt, folgt unmittelbar aus den schon dargelegten Unterschieden zwischen *TWinControl* und *TControl*. So können *TWinControl*-Elemente den Tastaturfokus erhalten ...

Tastaturfokus *R74*

Normalerweise entscheidet der Benutzer, welches Steuerelement den Tastaturfokus erhält, an welches Element also die Eingaben der Tastatur geleitet werden. Verwendet er Tabulatortaste und Pfeiltasten, entscheidet das Property *TabOrder*, in welcher Reihenfolge die einzelnen Elemente in einem Fenster angesteuert werden. Der Wert 3 im *TabOrder*-Property besagt beispielsweise, dass das Steuerelement innerhalb der übergeordneten Komponente als Drittes den Tastaturfokus erhält. Dies wurde, zusammen mit dem Property *TabStop*, bereits in Kapitel 1.4.5 unter *Rücksichtnahme auf die Tastaturbedienung* erläutert.

Um den Tastaturfokus programmgesteuert zu bewegen, bedienen Sie sich der Methoden von *TWinControl*:

- Die wichtigste Methode ist sicher *SetFocus*, die den Fokus auf das Steuerelement selbst setzt. So erreichen Sie zum Beispiel durch den Aufruf *NextButton.SetFocus*, dass ein Schalter namens *NextButton* fokussiert wird, so dass der Benutzer nur noch die Eingabetaste drücken muss, um diesen Schalter zu betätigen.
- Die Funktion *Focused* gibt mit ihrem booleschen Ergebnis an, ob das Steuerelement den Tastaturfokus besitzt.
- Außerdem gibt es noch die Funktion *CanFocus*, die besagt, ob das Element überhaupt fokussiert werden kann, was nur der Fall ist, wenn es sichtbar *und* aktiviert ist.

Unterelemente

Die wichtigsten Eigenschaften und Methoden für die Unterelemente eines *TWinControl* – *ControlCount* und das Array *Controls* sowie die Methoden *InsertControl* und *RemoveControl* – wurden bereits in Kapitel 3.2.2 erläutert. Auch die folgenden Methoden befassen sich in irgendeiner Weise mit den Kindelementen im Array *Controls*:

Methode	**Wirkung**
function ContainsControl(Control): Boolean;	überprüft, ob das als Parameter angegebene *Control* ein Kindelement des aktuellen *TWinControl*-Elements ist.
function ControlAtPos(Pos, AllowDisabled): TControl;	gibt das Steuerelement zurück, das sich an der Position des *TPoint*-Parameters *Pos* befindet. Falls *AllowDisabled=False*, ignoriert die Methode deaktivierte Elemente. Befindet sich kein oder nur ein inaktiviertes Element an der Position, liefert die Funktion *nil*.
procedure FindNextControl(...)	erlaubt die »Vorhersage«, wie sich Tabulator- und Pfeiltasten von einem bestimmten Startpunkt auf den Tastaturfokus auswirken, siehe Online-Hilfe.
procedure ScaleBy(a, b);	skaliert das Steuerelement um den Faktor a/b.
procedure ScrollBy(DeltaX, DeltaY);	führt programmgesteuerten Bildlauf durch.

Fenster-Handle

Das Fenster-Handle stellt die Verbindung zwischen einer *TWinControl*-Komponente und dem zugehörigen Windows-Fenster dar. Obwohl sie es selbst bereits perfekt verwaltet und Sie sich normalerweise nie darum zu kümmern brauchen, gewährt Ihnen die VCL freien Zugriff auf dieses Handle, und zwar über die Properties *Handle* (nur Lesen) und *WindowHandle* (Lesen und Schreiben, nur für Komponentenentwickler gedacht).

Methode	**Wirkung**
function HandleAllocated: Boolean	gibt an, ob die Komponente gerade mit einem Windows-Fenster verbunden ist.
procedure CreateHandle; virtual;	verbindet die Komponente mit einem zugehörigen Windows-Fenster, erlaubt in Zusammenhang mit *DestroyHandle* die gezielte Kontrolle des Ressourcen-Verbrauchs einer Komponente, siehe Kapitel 3.4.4.
procedure DestroyHandle;	löscht das zur Komponente gehörige Windows-Fenster, siehe *CreateHandle* und Kapitel 3.4.4.

Hinweis: Eng mit dem Fenster-Handle hängt auch das Property *DefWndProc* zusammen, das auf eine Funktion innerhalb von Windows weist, welche die Standardbehandlung von Nachrichten für dieses Fenster-Handle durchführt (siehe auch Kapitel 3.1.4).

Docking

Während die in *TControl* definierten Docking-Properties die Eigenschaft eines Steuerelements beschreiben, an anderen Steuerelementen angedockt zu werden, besitzt *TWinControl* die Docking-Properties, die sich auf Andockstellen beziehen (nur *TWinControls* können als Andockstelle arbeiten). Die Properties sind *DockSite*, *DockManager* und *UseDockManager* sowie verschiedene Events, die zum großen Teil in den Kapiteln 5.2.2 und 5.8.2 besprochen werden.

3.4 Formulare, TScrollingWinControl und TScrollBox

In diesem Kapitel erreichen wir das Ziel auf dem Weg durch die Vererbungshierarchie der VCL: die Klasse *TForm*, die die Elemente von sieben Vorfahrklassen erbt und über den gemeinsamen Vorfahr *TWinControl* eng mit anderen visuellen Komponenten verwandt ist. Auf dem Weg zu *TForm* machen wir noch bei einer weiteren Klasse Halt, der Klasse *TScrollingWinControl*. Diese Klasse sorgt bei *TForm* dafür, dass Sie mehr Komponenten in einem Formular unterbringen können, als in das Fenster passen, und dass Sie mit Hilfe von Bildlaufleisten doch alle Elemente erreichen können – sowohl zur Laufzeit als auch zur Entwurfszeit.

3.4.1 TScrollingWinControl und TScrollBox

In der VCL befinden sich mittlerweile drei von *TScrollingWinControl* abgeleitete Klassen: *TScrollBox*, *TCustomForm* und *TCustomFrame* (ab Delphi 5). Wie der Name schon sagt, ist das wichtigste gemeinsame Merkmal der drei Klassen, das sie vom üblichen *TWinControl* unterscheidet, die besondere Unterstützung des Scrollings. Diese besteht in der automatischen Erzeugung von Scrollbars, die fest in die Fenster dieser Klassen eingebunden sind, also nicht als einzelne (*TScrollBar*-)Komponenten vorliegen, und in der automatischen Durchführung des Scrollens.

TScrollingWinControl deklariert nur drei neue Properties, von den Methoden ist besonders die *ScrollInView*-Methode interessant:

Element	Beschreibung
property AutoScroll; Boolean;	wenn True, scrollt die Komponente automatisch alle Elemente in Sichtweite, die den Tastaturfokus erhalten.

Element	Beschreibung
property HorzScrollBar;	weist auf ein TControlScrollBar-Objekt für die horizontale Bildlaufleiste.
property VertScrollBar;	weist auf eine TControlScrollBar-Instanz für die vertikale Bildlaufleiste.
procedure ScrollInView(TControl);	verändert die Bildlaufleisten so, dass das als Parameter angegebene Element sichtbar wird.

Für die beiden *TControlScrollBar*-Objekte *HorzScrollBar* und *VertScrollBar* können Sie sowohl zur Laufzeit als auch zur Entwurfszeit die folgenden Properties verändern:

- Seit Delphi 4 haben Sie im Property *Style* neben dem gewohnten Stil zwei weitere zur Auswahl: *ssFlat* für die neumodischen flachen Leisten ohne 3D-Effekt und *ssHotTrack* für flache Leisten, deren verschiedene Bereiche auf die Bewegung der Maus reagieren (ähnlich wie flache Schalter in Symbolleisten, allerdings ohne 3D-Effekt).
- Ob eine Bildlaufleiste überhaupt angezeigt werden soll, legen Sie in *Visible* fest. Solange der gesamte Bereich sichtbar ist, wird die Bildlaufleiste unabhängig von *Visible* versteckt.
- *Range* enthält den horizontalen bzw. vertikalen Bereich, über den gescrollt wird. Falls Sie ihn zur Entwurfszeit manuell ändern, wird er nicht mehr automatisch angepasst, wenn Sie Komponenten einfügen oder entfernen.
- Die Schrittweite der Bewegung mit den Pfeilschaltern können Sie im Property *Increment* angeben; falls Sie eine automatische Anpassung dieser Schrittweite an die Größe des scrollbaren Bereichs wünschen, schalten Sie das Property *Smooth* ein.
- In *Margin* können Sie einen zusätzlichen Abstand angeben, der zum Bereich *Range* hinzugezählt werden soll. Wenn Sie *Margin* bei der Voreinstellung von 0 Pixeln lassen, wird nur so lange gescrollt, bis die am weitesten rechts bzw. unten befindlichen Elemente gerade noch ganz sichtbar sind.
- *Position* gibt die aktuelle Position der Bildlaufleiste an, sie liegt innerhalb des Bereichs *Range*. Der maximale Wert von *Position* ergibt sich, wenn Sie von *Range+Margin* die Breite des Fensters abziehen.
- Das Flag *Tracking* bezieht sich auf das Ziehen des Reglers mit der Maus. Ist es eingeschaltet, wird der Bereich auch während des Ziehens sichtbar gescrollt, sonst findet das Scrolling erst nach dem Loslassen der Maustaste statt.
- Weitere Äußerlichkeiten legen Sie mit den überwiegend selbst erklärenden Properties *ButtonSize*, *Color*, *ParentColor*, *Size* und *ThumbSize* fest.

Die eigenständige Scrollbar-Komponente *TScrollBar* weist übrigens eine andere Arbeitsweise als die Klasse *TControlScrollBar* auf. *TScrollBar*-Komponenten erlauben Ihnen darüber hinaus eine weitaus genauere Kontrolle des Scrollings über das Ereignis *OnScroll*.

TScrollBox

Die Klasse *TScrollBox* hat den Eigenschaften von *TScrollingWinControl* kaum etwas hinzuzufügen. Ein Beispiel zur Anwendung von *TScrollBox* zum Scrollen über einen großen virtuellen Zeichenbereich geben die Kapitel 5.5.4 und 5.6.3.

3.4.2 Die verschiedenen Arten von Formularen

Formulare können nach den folgenden grundlegenden Arten unterschieden werden:

- nach dem Ablauf, in dem sie angezeigt werden: als normales oder als modales Fenster;
- nach dem grundlegenden Rahmentyp, der auch Auswirkungen auf die Funktion des Fensters hat;
- in MDI-Anwendungen wird schließlich zwischen den MDI-Kindfenstern und dem MDI-Hauptfenster unterschieden.

TCustomForm

Ein Delphi-Formular wird normalerweise von der Klasse *TForm* abgeleitet. Was die deklarierten Properties, Ereignisse und Methoden angeht, ist diese jedoch – bis auf ein paar MDI-spezifische Methoden – mit der Klasse *TCustomForm* identisch. *TCustomForm* ist die gemeinsame Basisklasse von *TForm*, den in Delphi 3 neu eingeführten Klassen *TActiveForm* und *TPropertyPage* (für den ActiveX-Bereich) und der in Delphi 4 hinzugekommenen *TCustomDockForm*. Dass die allermeisten Elemente von *TForm* schon in *TCustomForm* deklariert sind, bedeutet nichts weiter, als dass sie auch für die anderen *TCustomForm*-Nachfahren unverändert gültig sind. In der Praxis brauchen Sie sich normalerweise nicht weiter um den Unterschied zwischen *TCustomForm* und *TForm* zu kümmern.

Modal und nicht modal

Wie in Kapitel 1.9.1 am Beispiel gezeigt wurde, können Sie ein Fenster modal machen, indem Sie seine *ShowModal*-Methode aufrufen. *ShowModal* arbeitet intern so, dass es die *Enabled*-Properties der nicht-modalen Formulare auf *False* setzt. Die VCL umgeht dadurch die von Windows vorgegebene unflexible *DialogBox*-Funktion zur Anzeige modaler Dialoge und erreicht so, dass Sie *jedes* Formular modal anzeigen können. Eine

Voraussetzung ist allerdings, dass das Formular unsichtbar ist, bevor Sie *ShowModal* aufrufen. Also benötigen Sie theoretisch zwei Zeilen, um ein beliebiges Fenster modal anzuzeigen:

```
FormX.Visible := False;
if FormX.ShowModal = mrOK then
  ...
```

Das Ergebnis von *ShowModal* gibt den Ergebniscode des Formulars an, wie z.B. *mrOK* oder *mrCancel*, wenn es mit dem *OK*- oder *Cancel*-Schalter beendet wurde. Nach der Ausführung von *ShowModal* ist das Fenster wieder unsichtbar, existiert aber weiterhin mit all seinen Daten (und verbraucht auch weiter Systemressourcen).

Ein modales Formular wird »geschlossen« (bzw. unsichtbar), nachdem das Formular-Property *ModalResult* einen Wert ungleich *mrNone* erhält. Dazu gibt es drei Möglichkeiten:

- Wenn der Benutzer das Formular über das Systemmenü schließt, setzt die VCL *ModalResult* selbst auf *mrCancel*.
- Sie können den Wert von *ModalResult* zur Laufzeit setzen, wodurch Sie bewirken, dass das Formular bei der nächsten Gelegenheit geschlossen wird (wenn die aktuelle Nachricht fertig bearbeitet ist).
- Schließlich können Sie im *ModalResult*-Property eines Schalters einen Wert eintragen. Der Schalter kopiert diesen, sobald er gedrückt wird, in das *ModalResult*-Property des Formulars (siehe Kapitel 1.9.1).

BorderStyle

Der Rahmen eines Fensters wird durch die Properties *BorderStyle* und *BorderIcons* festgelegt. In *BorderStyle* haben Sie die Möglichkeit, zwischen sechs verschiedenen Rahmenarten zu wählen. Nicht immer wirkt sich diese Einstellung nur auf das äußere Erscheinungsbild des Formulars aus. Abgesehen davon, dass Sie Fenster ohne breiten Rand nicht mit der Maus vergrößern und desgleichen Fenster ohne Titelzeile nicht verschieben können, verweigert die VCL den Fenstern, deren Rahmen im Dialogstil *bsDialog* gezeichnet wird, eine Menüzeile. Menüs, die Sie im Property *Menu* angeben, erscheinen einfach nicht, wenn Sie den *BorderStyle* auf *bsDialog* eingestellt haben.

Abbildung 3.4 zeigt eine Übersicht über die sechs Stile, die zur Wahl stehen. Zur Entwurfszeit werden Formulare immer mit dem Stil *bsSizeable* angezeigt. Um das Fenster also in seiner wahren Gestalt sehen zu können, müssen Sie das Programm zuerst starten. Die Stile sind im Einzelnen:

- *bsNone*: Mit diesem Stil erhalten Sie ein Fenster ohne Rahmen und Titelzeile. Falls Sie auch kein Hauptmenü angeben, besteht das gesamte Fenster nur aus seinem

Arbeitsbereich. Derartige Fenster werden häufig für Bildschirmschoner oder für den Hintergrund von Installationsprogrammen oder bei anderen speziellen Aufgaben wie Tutorials eingesetzt. Da ein solches Fenster auch kein Systemmenü hat, müssen Sie es mit der Tastenkombination Alt+F4 schließen, falls Sie kein Steuerelement zum Schließen in ihm unterbringen.

- *bsSingle*: Fenster mit diesem Stil haben eine Titelzeile, aber einen mit der Maus nicht veränderbaren Rahmen. Wenn Sie einen größenunveränderlichen Dialog haben wollen, aber ein Hauptmenü benötigen, sollten Sie statt des Stils *bsDialog* diesen Stil wählen.
- *bsSizeable* ist die Voreinstellung für ein leeres Formular mit frei manipulierbarem Rahmen. Fenster dieses Typs sind ganz normale Windows-Fenster und verfügen damit über all seine Fähigkeiten.
- *bsDialog* gibt Ihnen ein Dialogfenster mit optisch speziell gestaltetem Rahmen, das keine Hauptmenüzeile haben darf. Für die Delphi-Anwendung gibt es keine weiteren Unterschiede, der Rahmenstil spielt für die Funktion der sichtbaren Komponenten des Formulars keine Rolle.
- *bsToolWindow* ist eine Variation des Stils *bsSingle*, bei der die Titelzeile des Fensters wie die Titelzeile von Delphis Objektinspektor kleiner dargestellt wird.
- Mit *bsSizeToolWin* erhalten Sie ebenfalls ein Fenster mit kleiner Titelzeile, das jedoch größenveränderlich ist.

Hinweis: Vom Erscheinungsbild her gibt es noch einen weiteren Rahmentyp: Bei MDI-Hauptfenstern wird der Rahmen automatisch so gezeichnet, dass das Innere des Hauptfensters tiefer als der Rahmen erscheint.

BorderIcons

Das zweite Property, das bei der Rahmengestaltung eine Rolle spielt, ist *BorderIcons*. Es enthält eine Auswahl der Flags *biSystemMenu*, *biMinimize*, *biMaximize* und *biHelp*, von denen die ersten drei angeben, ob das Systemmenü und die Schalter zum Verkleinern des Fensters auf Symbolgröße und zum Maximieren des Fensters angezeigt werden sollen.

BiHelp sorgt dort dafür, dass rechts in der Titelleiste ein Hilfeschalter mit der Beschriftung »?« erscheint, aber nur, wenn gleichzeitig *biMinimize* und *biMaximize* abgeschaltet sind (siehe Abbildung 3.4). Dieser Schalter ist nur dann sinnvoll, wenn Sie die entsprechende Funktion von WinHelp 4.0 nutzen, wofür hier aber auf die Online-Dokumentation verwiesen werden muss.

Abbildung 3.4: Verschiedene Formularstile, demonstriert von StyleBtn

BiMinimize und *BiMaximize* entfallen in Formularen mit dem Rahmenstil *bsDialog* auf jeden Fall, im Rahmenstil *bsNone* können gar keine Rahmenicons dargestellt werden. Fenster der Stile *bsToolWindow* und *bsSizeToolWin* haben immer nur ein Rahmenicon, und zwar das Schließenfeld.

Die Abwesenheit eines Rahmenicons bedeutet jedoch an sich noch nicht die Abwesenheit der mit dem Icon verbundenen Funktionen. So können Sie, wie schon erwähnt, Fenster im Stil *bsNone* mit Alt+F4 schließen.

FormStyle

FormStyle enthält verschiedene weitere Variationsmöglichkeiten. Abgesehen vom voreingestellten Wert *fsNormal*, der keine Besonderheiten aufweist, haben Sie die drei folgenden Werte zur Auswahl:

- *fsMDIForm* und *fsMDIChild*: *fsMDIForm* macht ein Formular zum MDI-Hauptfenster, von dem es nur eines in jeder Anwendung geben darf; zu ihm gehören ein oder mehrere Formulare mit dem Stil *fsMDIChild* (siehe Kapitel 5.7).
- Mit *fsStayOnTop* erreichen Sie, dass ein Fenster immer vor allen Fenstern derselben Anwendung zu sehen ist, die nicht diesen Stil aufweisen. Da Sie diesen Stil zur Laufzeit beliebig zwischen *fsStayOnTop* und *fsNormal* umschalten können, bietet es sich an, dafür in der Anwendung einen Popup-Menüpunkt abzubieten, wie Sie es z.B. auch in der Delphi-IDE für den Objektinspektor finden (Menüpunkt IMMER IM VORDERGRUND).

Wenn Sie diesen Stil dem Hauptfenster Ihrer Anwendung zuweisen, bleibt dieses sogar vor allen anderen Anwendungen sichtbar, die nicht ebenfalls diesen Stil besitzen.

Veränderungen zur Laufzeit

Delphis Formulare weisen auch zur Laufzeit eine Flexibilität auf, mit der sie ein normales Windows-Fenster weit in den Schatten stellen. Sie können die Art des Formulars nahezu beliebig zur Laufzeit ändern. Da Windows mit den Fenstern eher restriktiv umgeht und nur wenige grundlegende Änderungen an einem bestehenden Fenster zulässt, löscht die VCL bei einer Änderung von *BorderStyle*, *BorderIcons* und *FormStyle* einfach das Windows-Fenster und erstellt ein neues. (Sie können das mit dem Programm *StyleBtn* [Abbildung 3.4] testen: Jedes Mal wenn Sie eine andere Option wählen, verschwindet das Fenster kurz, um dann in neuem Outfit wieder zu erscheinen.)

Die Tatsache, dass bei Änderung einer der genannten Optionen für kurze Zeit gar kein Fenster für das Formular existiert, berührt dieses jedoch nicht: Die Ereignisbearbeitungsmethoden merken überhaupt nichts von diesem Vorgang, für sie scheint es, als werde das Fenster lediglich erneut angezeigt, es tritt also ein *OnShow*-Ereignis auf. Ein *OnCreate*-Ereignis wird dagegen nicht erzeugt, da das Formular gar nicht zusammen mit dem Windows-Fenster gelöscht wurde.

Es gibt jedoch gewisse Grenzen: Sie können beispielsweise ein normales Formular in ein MDI-Formular umwandeln, ihm ein MDI-Kindfenster geben und dann seinen *BorderStyle* von *bsSizeable* in *bsDialog* umwandeln. Durch den letzten Schritt verliert es zunächst sein Kindfenster, das also erst einmal vom Bildschirm verschwindet. Wenn Sie den Rahmenstil des Hauptfensters dann wieder zurück auf *bsSizeable* setzen und das verloren gegangene MDI-Kindfenster erneut mit dem Hauptfenster verknüpfen, indem Sie sein *Parent*-Property auf das Hauptfenster setzen, erscheint es zwar wie gewünscht als MDI-Kindfenster, allerdings haben nicht alle seine Properties diese Transformation überstanden: Beispielsweise sind Fenstertitel und Editierfeld-Inhalte verloren.

Diese haarsträubenden Ereignisse können Sie mit den vier Schaltern des Demoprogramms *VCL Stress* testen:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FormStyle := fsMDIForm; { bisher: fsNormal }
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  F := TForm2.Create(self); { neues MDI-Fenster }
end;
```

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  BorderStyle := bsDialog; { Kindfenster geht verloren }
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  BorderStyle := bsSizeable; { MDI wieder funktionsfähig }
  F.Parent := self;   { Kindfenster wird wiedergefunden }
end;
```

Noch mehr überschreiten Sie die Grenzen des Möglichen in einem ebenfalls unrealistischen Fall: Wenn Sie ein MDI-Hauptformular, das schon Kindfenster hat, in ein normales Formular umwandeln, erzeugt die VCL eine Exception (wenn keine MDI-Kindfenster vorhanden sind, funktioniert es noch). Falls Portabilität in Ihren Anwendungen eine Rolle spielt, sei schließlich auch noch angemerkt, dass die Formulare der CLX von Kylix (zumindest in der vom Autor getesteten ersten Version) derartige Laufzeitänderungen nicht so gut verdauen wie die der VCL.

3.4.3 Eigenschaften und Ereignisse der Formulare

Die folgende Tabelle zeigt eine Zusammenfassung der wichtigsten Properties, die ein Formular seinen Vorfahrklassen voraus hat, ausgenommen die bereits bei der automatischen Größeneinstellung/-beschränkung in Kapitel 3.3.5 erläuterten sowie die mit dem Docking (Kapitel 5.2.2 und 5.8.2) in Zusammenhang stehenden.

Property	Typ	Bedeutung
Active	Boolean	gibt an, ob das Fenster das gerade aktive Fenster der Anwendung ist.
ActiveControl	TControl	weist auf das Steuerelement, das den Tastaturfokus hat.
BorderIcons	TBorderIcons	siehe Kapitel 3.4.2.
BorderStyle	TFormBorderStyle	siehe Kapitel 3.4.2.
Canvas	TCanvas	Zeichenfläche, die den gesamten Client-Bereich umfasst (von dem z.B. Mauspaletten und Statuszeilen abgezogen sind).
DefaultMonitor	(dmDesktop, dmPrimary, dmMainForm, dmActiveForm)	Wenn Sie es unter Windows 98 nicht den äußeren Umständen (Benutzer, Betriebssystem ...) überlassen wollen, auf welchem von mehreren Bildschirmen ein Fenster erscheinen soll, stellen Sie dieses Property nach Anleitung der Online-Hilfe ein.
FormStyle	TFormStyle	siehe Kapitel 3.4.2.
Icon	TIcon	Icon für die Symboldarstellung des Formulars, standardmäßig wird das Icon der Anwendung verwendet.
KeyPreview	Boolean	Wenn *True*, ist die Vorschaufunktion auf die Tastatureingaben der Steuerelemente eingeschaltet, siehe Kapitel 5.8.1.

Property	Typ	Bedeutung
Menu	TMainMenu	Komponente, die das Hauptmenü für das Fenster enthält.
ModalResult	TModalResult (Integer)	Rückgabewert für ein modales Formular (siehe Kapitel 3.4.2), vordefinierte Werte sind *mrNone*, *mrOK*, *mrCancel*, *mrAbort*, *mrRetry*, *mrIgnore*, *mrYes*, *mrNo*, *mrAll*.
ObjectMenuItem	TMenuItem	Verweis auf einen Menüpunkt, an den das Menü eines OLE-Objekts angehängt wird (in Verbindung mit der Komponente *TOleContainer*).
OldCreateOrder	Boolean	Hierzu sollten Sie sich in der Online-Hilfe näher informieren, wenn Sie ein Delphi 1 – 3-Projekt in einer höheren Delphi-Version bearbeiten wollen, das vom Ablauf der Objekterzeugung und -destruktion abhängig ist.
PixelsPerInch	Integer	Vom System definierte Auflösung des Bildschirms in Punkten pro Zoll (unter Windows normalerweise 96, unter Linux 75).
Position	TPosition	gibt mit folgenden Werten an, ob Windows zur Laufzeit die Fensterposition und/oder die Größe des Fensters selbst festlegen soll (*Default*) oder ob die zur Entwurfszeit angegebene Position bzw. Größe verwendet werden soll (*Designed*): *poDesigned*, *poDefault*, *poDefaultPosOnly*, *poDefaultSizeOnly*. Darüber hinaus gibt es noch *poScreenCenter* für bildschirmzentrierte Anzeige, *poDesktopCenter* und *poMainFormCenter* und *poOwnerFormCenter* (ab Delphi 5). Zwischen der Zentrierung auf den Bildschirm und auf den Desktop gibt es natürlich nur dann einen Unterschied, wenn Bildschirm und Desktop verschiedene Bereiche abdecken (bei virtuellen Bildschirmen oder Desktops der Fall).
PrintScale	TPrintScale	gibt an, wie die Methode *Print* den Formularinhalt druckt: bei *poNone* wird das Formular ohne Skalierung gedruckt, bei *poProportional* versucht *TForm*, es im Ausdruck so groß wie auf dem Bildschirm darzustellen, und bei *poPrintToFit* wird es so groß gedruckt, dass es gerade noch auf die Seite passt.
WindowState	TWindowState	*wsNormalized*, *wsMaximized* und *wsMinimized* geben an, ob das Fenster normal, im Vollbild oder in der minimierten Darstellung angezeigt wird.

Dazu kommen natürlich alle von *TScrollingWinControl*, *TWinControl* und den anderen Klassen geerbten Properties, selbst wenn diese für Formulare manchmal ungewöhnlich sind. So hat auch ein Formular ein *Enabled*-Property, das aber meistens nur gesetzt wird (und dann automatisch von der VCL), wenn ein anderes Fenster modal angezeigt wird.

Die folgenden Properties sind ebenfalls Bestandteil von *TForm*, gelten aber nur für MDI-Hauptfenster und sind bis auf *WindowMenu* nicht zur Entwurfszeit ansprechbar (mehr zu MDI-Anwendungen in Kapitel 5.7):

MDI-Property	**Typ**	**Bedeutung**
ActiveMDIChild	TForm	gibt bei einem MDI-Hauptfenster das aktive MDI-Kindfenster an.
MDIChildCount	Integer	Zahl der MDI-Kindfenster
MDIChildren	array of TForm	Liste aller MDI-Kindfenster
TileMode	TTileMode	*tbHorizontal* oder *tbVertical*: bestimmt, ob die Fenster nach der Aufteilung mit *Tile* eher breit und niedrig (*tbHorizontal*) oder schmal und hoch sein sollen (*tbVertical*).
WindowMenu	TMenuItem	Menü, an das die Liste der MDI-Kindfenster angehängt wird

Wichtige Ereignisse

Ein Formular hat als Elternfenster und Besitzer zahlreicher Komponenten viele Verwaltungsaufgaben, von denen die unbedingt notwendigen natürlich in der VCL bereits implementiert sind. Um Ihnen die Möglichkeit zu geben, am Management des Formulars mitzuwirken, stellt *TForm* Ihnen eine Reihe von Ereignissen zur Verfügung, die über die von den meisten Komponenten verwendeten Ereignisse hinausgehen:

Ereignis	**Bedeutung**
OnActivate	Benachrichtigung, dass das Formular aktiviert wurde (beim Wechsel zwischen Anwendungen wird statt dessen das *OnActivate*-Ereignis des *Application*-Objekts erzeugt, siehe Kapitel 3.2.3).
OnCanResize	Hier können Sie unmittelbar vor einer Größenänderung des Fensters diese noch verhindern oder eine neue Größe vorgeben.
OnClose	Hierin entscheiden Sie, wie das Formular geschlossen werden soll, siehe Kapitel 3.4.4.
OnCloseQuery	Voranfrage, ob Formular geschlossen werden darf.
OnConstrainedResize	Dient zum dynamischen Überschreiben der im *Constraints*-Property festgelegten Größeneinschränkungen.
OnCreate	Initialisierungs-Ereignis: Das Formular wurde gerade erzeugt, ist aber noch nicht sichtbar.
OnDeactivate	Ein anderes Formular der Anwendung wird aktiviert.
OnDestroy	Gegenstück zu *OnCreate*: Das Formular wird gerade freigegeben.
OnPaint	Auftrag an das Formular, sich neu zu zeichnen.
OnResize	Die Größe des Formulars (*Width*/*Height* bzw. *ClientWidth*/*ClientHeight*) hat sich geändert.

Ereignis	Bedeutung
OnShortCut	Hier können Sie Tastenkürzel definieren, die nicht schon aufgrund eines *ShortCut*-Properties eines Menüs, Schalters oder einer Aktionsliste automatisch von der VCL abgefragt werden. *OnShortCut* wird für jede Tastatureingabe aufgerufen, noch bevor die Standardverarbeitung mit *OnKeyDown* usw. einsetzt.
OnShow	Das Formular wird sichtbar, sobald die *OnShow*-Bearbeitungsmethode ihre Arbeit beendet hat.

Neue Transparenz- und Überblendeffekte

In Delphi 6 haben die Formulare eine optisch sehr interessante Neuerung erfahren, die allerdings als Betriebssystem Windows 2000 oder einen seiner Nachfolger sowie auch einiges an Rechenleistung voraussetzen. Die Effekte teilen sich in zwei Kategorien auf:

- Überblendeffekte (Alphablending): Hierbei ist ein Fenster ganz oder teilweise durchsichtig, lässt also die hinter ihm liegenden Fensterteile durchscheinen. Diesen Effekt schalten Sie über das Property *AlphaBlend* ein; in *AlphaBlendValue* geben Sie die Stärke der Durchsichtigkeit an (zwischen 0 für komplett durchsichtig und 255, was einem normalen undurchsichtigen Fenster entspricht). Durch kontinuierliche Anpassung von *AlphaBlendValue* von einem zum anderen Extremwert erreichen Sie, dass ein Fenster wie in einer Dia-Überblendshow ein- bzw. ausgeblendet wird.
- Transparenzeffekte: Hier wird nur eine bestimmte Farbe des Fensters als vollständig transparent definiert. Dies bewirkt, dass anstelle der ausgewählten Farbe immer die hinter dem Fenster liegende Bildschirmansicht erscheint. Diesen Effekt schalten Sie mit dem Property *TransparentColor* ein, die Farbe wird in *TransparentColorValue* eingestellt. Ein Anwendungsgebiet für diese Funktion sind Fenster mit scheinbar nicht-rechteckigen Umrissen, wie das folgende Beispiel zeigen wird.

Hinweis: Während *AlphaBlendValue* den Wert Null hat, kann ein Formular keine Mausklicks empfangen.

Beispiel für den Einsatz der Effekte *R29*

Das Projekt *AlphaBlendingDemo* auf der CD demonstriert beide erwähnten Effekte: Das Fenster besitzt eine nicht rechteckige Form (siehe Abbildung 3.5) und wird bei Programmstart langsam eingeblendet. Für das Einblenden wird eine Timer-Komponente verwendet, deren *Intervall* zur Entwurfszeit auf 20 Millisekunden eingestellt wurde. Ebenfalls schon zur Entwurfszeit wurde das Property *AlphaBlend* des Formulars auf *True* gesetzt.

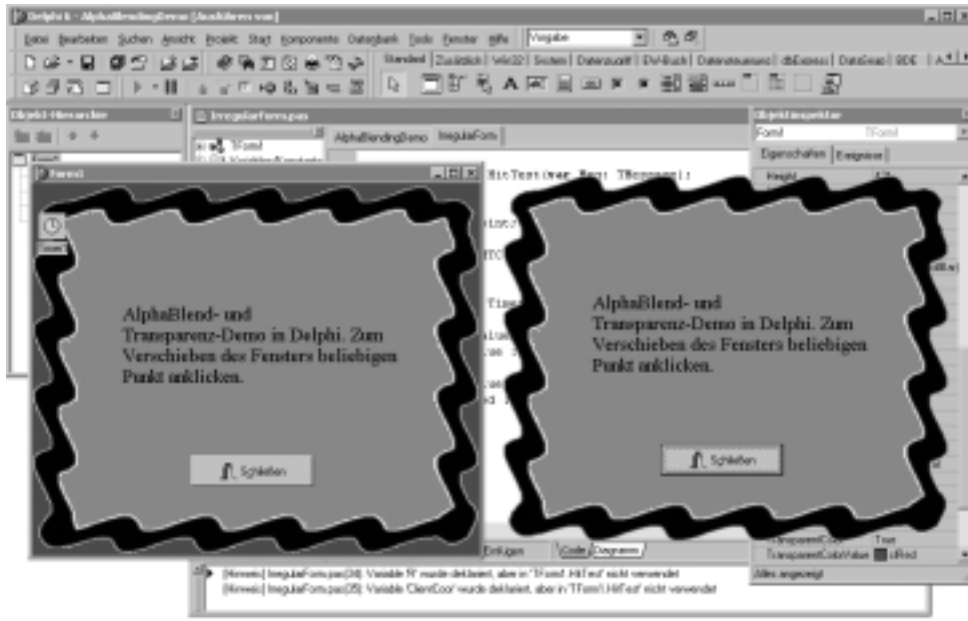


Abbildung 3.5: Das individuell geformte Fenster zur Entwurfszeit und zur Laufzeit mit wirksamer Transparenz-Einstellung

Die Methode, die bei jedem Timer-Tick ausgeführt wird, ist die Folgende:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if AlphaBlendValue < 245 then
    AlphaBlendValue := AlphaBlendValue + 10
  else begin
    AlphaBlendValue := 255;
    Timer1.Enabled := False;
  end;
end;
```

AlphaBlendValue wird also in 26 Schritten bis zum Maximalwert von 255 erhöht, was wegen des Timers 520 Millisekunden dauern sollte (wenn der Rechner die einzelnen Schritte nicht schnell genug ausführt, kann es auch länger dauern). Wenn der Maximalwert erreicht ist, hat der Timer seinen Zweck erfüllt und wird deaktiviert.

Die unregelmäßige Form erreicht das Fenster dadurch, dass

- der *BorderStyle* des Formulars auf *bsNone* gesetzt wurde, so dass kein Standardrahmen angezeigt wird

- der Client-Bereich des Formulars ganz durch eine *TImage*-Komponente ausgefüllt wurde, die eine Bilddatei anzeigt, welche als Rahmenersatz und Transparenzmaske dient. Für das Beispiel wurde ein Rahmenassistent des Micrografx Picture Publishers verwendet, um den gewellten Rahmen zu erzeugen. Der außerhalb des Rahmens liegende Bereich wurde mit einer Farbe gefüllt, die innerhalb des Formulars nicht vorkommt.
- Diesese Farbe wurde auch im *TransparentColorValue*-Property des Formulars angegeben.
- Schließlich musste natürlich das Property *TransparentColor* des Formulars auf *True* gesetzt werden.

Die transparente Farbe wird nun von Windows 2000 und seinen Nachfolgern durch Transparenz ersetzt. Wichtig ist, dass die Farbe innerhalb des Rahmens an keiner Stelle verwendet wird, denn sonst würde auch hier der Hintergrund durchscheinen, und ein durchlöchertes Fenster war ja nicht das Ziel des Beispielprogramms.

Hinweis: Beachten Sie, dass Windows so intelligent ist, Mausklicks an das Fenster zu leiten, das gerade unter der Maus sichtbar ist. Dabei kommt es auf den einzelnen Bildpunkt unter dem Mauszeiger an: Ist er transparent, geht der Klick an das unter dem Delphi-Fenster befindliche Fenster und führt zu dessen Aktivierung. Dies gilt auch für den erwähnten Fall eines »durchlöcherten« Formulars, selbst wenn nur ein einzelner Pixel im Inneren des Formulars transparent ist.

Ein Fenster ohne Rahmen kann unter Windows normalerweise nicht mit der Maus verschoben werden. Mit einer kleinen Zusatzmaßnahme können Sie jedoch erreichen, dass auch ein rahmenloses Fenster nicht starr auf dem Bildschirm fixiert ist. Das Beispielprogramm etwa kann an jeder Stelle des Formulars außer im Bereich des Buttons angeklickt und verschoben werden. Es bearbeitet dazu die Windows-Botschaft *WM_NCHITTEST*, mit der Windows bei jeder Mausbewegung fragt, ob sich unter der Maus irgendwelche nicht-standardmäßigen »NonClient« -Bereiche wie etwa eine selbst gezeichnete Titelleiste befinden. Das Formular gibt jeden beliebigen Punkt als Titelleiste aus, was dazu führt, dass jeder Punkt wie eine normale Titelleiste als Angriffspunkt für Verschiebeaktionen verwendet werden kann:

```
// Abfangen einer Windows-Botschaft per message-Direktive (siehe K. 6.4.1)
// in der Formulardeklaration:
    procedure HitTest(var Msg: TMessage); message WM_NCHITTEST;

procedure TForm1.HitTest(var Msg: TMessage);
begin
  Msg.Result := HTCaption;
end;
```

3.4.4 Arbeiten mit mehreren Formularen

Obwohl die Verwendung mehrerer Formulare grundsätzlich ganz einfach ist, gibt es einige beachtenswerte Dinge. Zunächst eine kurze Zusammenfassung der Erkenntnisse aus der Projektverwaltung (siehe Kapitel 1.6.3):

- In den Projektoptionen geben Sie ein *Hauptformular* an, das in der Delphi-Anwendung eine besondere Rolle spielt und das als Erstes erstellt wird.
- Alle anderen Formulare, die Sie in den Projektoptionen dafür bestimmt haben, werden zur Laufzeit automatisch erzeugt.
- Die Formulare, deren *Visible*-Property Sie zur Entwurfszeit auf *True* gesetzt haben, werden auch automatisch angezeigt.

Zur Laufzeit können Sie das Hauptformular zwar nicht mehr auswechseln, aber doch dynamisch neue Formulare erzeugen und das Property *Visible* verändern.

Schließen des Hauptformulars und anderer Formulare *R16*

Zwar funktioniert das Schließen eines Fensters zur Laufzeit wie gewohnt, Sie müssen also nur auf das Systemmenü doppelklicken oder Strg+F4 drücken, hinter den Kulissen gibt es jedoch etwas zu bedenken:

Wenn Sie das Hauptformular schließen, wird die Anwendung beendet und alle Formulare werden freigegeben. Wenn Sie ein anderes Formular schließen, so wird die Anwendung nicht geschlossen. Wie kann der Benutzer nun ein aus Versehen geschlossenes »Sekundärformular« wieder öffnen, ohne die gesamte Anwendung neu zu starten?

Die Lösung der VCL für Fenster, die nicht das Hauptformular sind, ist einfach: Standardmäßig gibt sie ein solches Fenster nicht frei, wenn der Benutzer es in der üblichen Art schließt, sondern versteckt es nur. Sie können daher im Hauptfenster z.B. in einem Ansicht-Menü einen Menüpunkt zur Verfügung stellen, mit dem der Benutzer das Formular wieder sichtbar machen kann. Rufen Sie in der Methode für diesen Menüpunkt einfach die Methode *Show* des versteckten Formulars auf. Eine andere Lösung wäre, dem Benutzer nicht zu erlauben, ein solches Fenster zu schließen. Sie müssten dazu das Ereignis *OnQueryClose* entsprechend bearbeiten.

Sind die Ressourcen noch knapp?

Im Folgenden sind drei verschiedene Methoden beschrieben, mit denen Sie den Ressourcenverbrauch Ihrer Formulare gering halten können. Zwar sollte man annehmen, dass solche Sparmaßnahmen unter 32-Bit-Windows nicht mehr erforderlich sind, aufgrund der Architektur von Windows 95/98 kann es dort jedoch wie unter Windows 3.1

leicht zur zeitweiligen Erschöpfung der Ressourcen kommen, selbst wenn Sie keine 16-Bit-Programme verwenden. Mehr zum Ressourcenverbrauch von Fenstern und Dialogen finden Sie am Ende von Kapitel 3.5.4 im Abschnitt *Ressourcenbedarf*.

Ansatzpunkt zum Ressourcensparen: unsichtbare Fenster

Ein Nachteil der automatischen Verwaltung der Windows-Ressourcen durch die VCL ist, dass auch ein verstecktes Fenster Teile der Windows-Ressourcen belegt. Wenn Ihr Projekt aus zahlreichen Formularen besteht, kann es mit den Ressourcen schon einmal knapp werden, falls Sie alle Formulare automatisch erzeugen lassen (was der Voreinstellung in den Projektoptionen entspricht). Wenn aber von diesen Formularen immer nur wenige sichtbar sind, ist es leicht, diese Ressourcen zu schonen. Sie haben dazu unter Delphi verschiedene Möglichkeiten.

Ressourcen sparen durch dynamische Formularerstellung *R30*

So können Sie das Formularobjekt für die Zeit, in der es nicht angezeigt wird, freigeben – oder anders ausgedrückt: Sie schalten die automatische Erzeugung des Formulars in den Projektoptionen aus und konstruieren das Formular erst bei Bedarf, um es wieder freizugeben, sobald der Bedarf gedeckt ist.

Diese Vorgehensweise ist sehr gut für Formulare geeignet, die modal angezeigt werden, denn diese können Sie noch in der gleichen Methode, in der sie konstruiert werden, wieder freigeben. Angenommen, Sie haben ein Dialogformular der Klasse *TDialogForm* entworfen, dann können Sie dieses wie folgt dynamisch erzeugen:

```
procedure TForm1.MenuePunktClick(Sender: TObject);
var
  DynamischerDialog: TDialogForm;
begin
  DynamischerDialog := TDialogForm.Create(self);
  try
    DynamischerDialog.ShowModal;
  finally
    DynamischerDialog.Free;
  end;
end;
```

Ressourcen sparen durch geändertes Schließverhalten *R36*

Formulare, die nicht modal angezeigt werden, können Sie weniger gut von außen schließen, da es hier keinen Aufruf von *ShowModal* gibt, dessen Beendigung Sie einfach abwarten könnten. Statt dessen können Sie jedoch das oben erwähnte Standard-Schließverhalten des Formulars verändern.

Normalerweise wird ein Formular, das nicht das Hauptformular ist, nur unsichtbar gemacht, wenn der Benutzer das Systemmenü doppelklickt oder eine gleichwertige Aktion durchführt, die normalerweise zum Schließen des Fensters führt. Um das Formular bei einem solchen Ereignis freizugeben, können Sie das Ereignis *OnClose* bearbeiten. Sie haben dabei die Wahl, ob das Fenster versteckt (*caHide*) oder freigegeben werden soll (*caFree*) oder ob überhaupt nichts geschehen soll (*caNone*). Weisen Sie den Wert Ihrer Wahl dem Variablenparameter *Action* zu, wie im folgenden Beispiel geschehen:

```
procedure TSekundaerFormular2.FormClose
   (Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Natürlich dürfen Sie keine Methoden (wie z.B. *Show*) eines freigegebenen Formulars aufrufen oder Properties und Variablen lesen oder verändern. Da Sie nicht vorhersehen können, wann der Benutzer ein solches nicht-modales Formular schließt, empfiehlt sich eine Bearbeitung von *OnClose* am ehesten für MDI-Kindfenster (siehe Kapitel 5.7.4), deren Vorhandensein für das Hauptfenster keine große Rolle spielt.

Wenn Sie ein Formular, das durch den Benutzer freigegeben werden kann, in anderen Formularen programmgesteuert ansprechen, so brauchen Sie eine Möglichkeit, zu erkennen, ob das Formular existiert oder nicht. Besonders einfach ist dies zu realisieren, wenn Sie eine feste Variable für das dynamisch konstruierte Formular verwenden und diese auf *nil* setzen, sobald das Formular geschlossen wird:

```
var
  Form2: TForm2;

procedure TForm2.FormClose
   (Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
  Form2 := nil;
end;
```

Hinweis: Das Setzen von *Form2 := nil* hat keinerlei Auswirkungen auf das Formular selbst, da *Form2* intern nur ein Zeiger auf das tatsächliche Formular ist. Das Setzen von *nil* dient also ausschließlich der Information anderer Programmteile.

In jedem Fall gehen bei der Freigabe eines Formulars mit *Free* natürlich alle Variablen dieses Formulars verloren, daher ist noch eine weitere Möglichkeit des Ressourcensparens interessant:

Ressourcen sparen durch Trennung vom Fenster-Handle *R44*

Um vollen Nutzen aus der Unabhängigkeit der VCL-Komponenten von den Windows-Fenstern zu ziehen, können Sie ein Formular zur Laufzeit bewusst vom Windows-Fenster lösen, ohne das Formularobjekt mit all seinen Daten zu verlieren. Wie schon erläutert, genügt es dazu nicht, das Fenster einfach zu verstecken.

Statt dessen können Sie das Windows-Fenster mit der Methode *DestroyHandle* bewusst löschen und mit *CreateHandle* gezielt wieder erstellen. Beide Methoden haben den Zugriffsschutz *protected*, so dass Sie beispielsweise die *DestroyHandle*-Methode eines Formulars nicht von einem anderen Formular aus aufrufen können. Sie können jedoch für ein Formular zwei öffentliche Methoden zur Ressourcen-Freigabe und -Wiederbelegung schreiben, die Sie auch von außen aufrufen können.

Die folgenden beiden Methoden sind dem Beispielprogramm *ResTest* von der CD-ROM entnommen:

```
procedure TConsumerForm.FreeResources;
begin
  DestroyHandle;
  Visible := False; { notwendig, damit Visible:=True in der
                      nächsten Methode funktioniert. }
end;

procedure TConsumerForm.RestoreResources;
begin
  CreateHandle;
  Visible := True;
end;
```

FreeRessources gibt das Handle des Windows-Fensters frei, was bewirkt, dass das Formular sofort vom Bildschirm verschwindet, ohne dass dabei das Formularobjekt selbst gelöscht wird. Die Methode *RestoreRessources* stellt den alten Zustand wieder her und erstellt das Windows-Fenster und alle enthaltenen Steuerelemente erneut.

Hinweis: Sie sollten bei einem Formular, das Sie mit *DestroyHandle* freigeben, genau überprüfen, dass die Inhalte der Komponenten durch die Zerstörung des Fenster-Handle nicht verloren gehen. Zwar werden Ihre Formularvariablen nicht beeinflusst, wenn aber eine Komponente Teile ihres Inhalts, die von Windows verwaltet werden, nicht in eigenen Datenbereichen puffert, gehen diese durch den Aufruf von *DestroyHandle* verloren.

Das Beispielprogramm *ResTest* (Abbildung 3.6) demonstriert die Wirkung dieser Vorgehensweise: Wenn Sie dort den Schalter *Ressourcen freigeben* drückten, wurden dereinst unter Windows 3.1 ca. 10% der Ressourcen freigegeben; die Ressourcen-Anzeige von Windows 98 meldete im Test immer noch einen Gewinn von 8%.

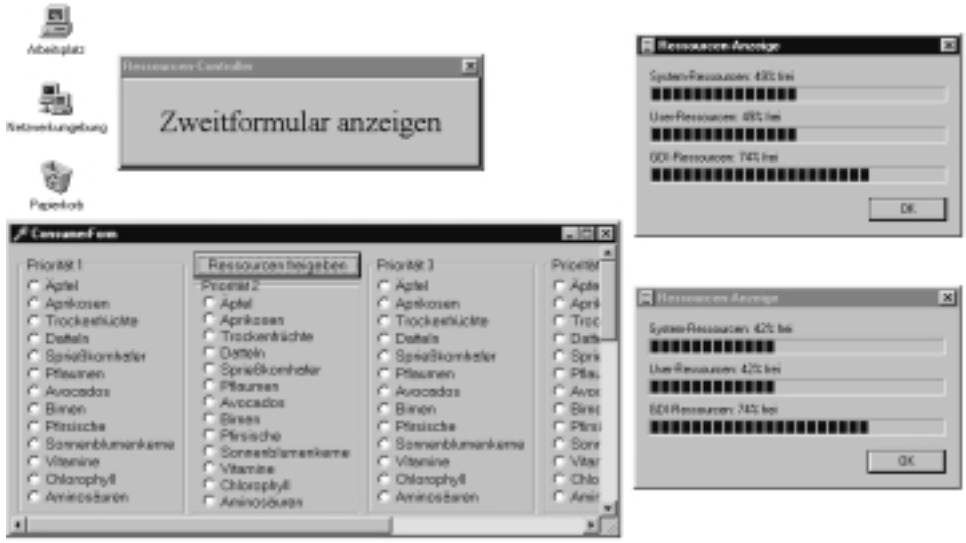


Abbildung 3.6: Ein Programm zur Demonstration der Methode DestroyHandle

Sie können in *ResTest* auch versuchen, das ressourcenintensive Fenster nur zu schließen. Da *OnClose* nicht überschrieben wurde, wird das Fenster dann nur versteckt – dementsprechend werden Sie in diesem Fall keine Veränderung der freien Ressourcen feststellen können.

3.5 Aufbau von Formularen und Verwendung von Dialogen

Delphi stellt Ihnen einige leistungsfähige Funktionen zur Verfügung, mit denen Sie auch komplexe Formulare übersichtlich gestalten und einfach programmieren können. Dieses Kapitel beschreibt sowohl den Entwurf von Steuerelement-Gruppen und mehrseitigen Formularen als auch deren Handhabung im Programmcode. Diese Techniken werden zwar meistens in Dialogen angewendet, Sie können sie aber auch bei jedem normalen Formular benutzen. Der letzte Abschnitt dieses Kapitels (3.5.7) behandelt schließlich ein reines Dialog-Thema: die Standarddialog-Komponenten, die Ihnen in vielen Fällen den Entwurf eigener Dialog-Formulare ganz ersparen.

3.5.1 Steuerelemente in Gruppen

Zwar können Sie Steuerelemente auf einfache Weise optisch gruppieren, indem Sie sie beispielsweise mit einer *TBevel*-Komponente umranden; eine erheblich leistungsfähigere Methode der Gruppierung besteht jedoch im Aufbau einer Eltern-Kind-Hierarchie der Komponenten. Bei dieser werden die Kindkomponenten einer Eltern-

komponente als Steuerelement-*Gruppe* angesehen. Diese Art der Gruppierung nutzt nicht nur dem Anwender, der ein wahrscheinlich übersichtlicheres Formular präsentiert bekommt, sondern hilft auch bei der Handhabung des Formulars und oft auch bei dessen Programmierung (siehe beispielsweise Steuerelement-Arrays in Kapitel 3.5.2).

Gruppieren von Steuerelementen zur Entwurfszeit *R I*

Um eine Gruppe von Steuerelementen zu erzeugen, benötigen Sie zunächst eine *Container-Komponente*, die andere Steuerelemente enthalten kann. Als Elternkomponenten kommen die folgenden Komponenten in Frage:

- in allen Delphi-Versionen die Komponenten *TPanel*, *TNotebook*, *TTabbedNotebook*, *TScrollBox*, *TGroupBox*;
- in 32-Bit-Delphi zusätzlich die Komponenten *TPageControl bzw. TTabSheet* (eine Komponente, die Sie nur innerhalb einer *TPageControl*-Komponente erzeugen können) und *TDBCtrlGrid*, das jedoch nur wenige datenbankspezifische Komponententypen enthalten kann;
- im Falle von Symbolleisten bietet es sich außerdem an, statt *TPanel* die Komponente *TToolbar* zu verwenden (siehe Kapitel 5.2.1 und 5.2.2).

Um einer dieser Komponenten Steuerelemente hinzuzufügen, haben Sie die folgenden Möglichkeiten:

- Wenn Sie ein Steuerelement direkt mit der Maus auf der Fläche der Elternkomponente einzeichnen, erkennt Delphi, dass das Steuerelement in diese Komponente eingefügt werden soll.
- Um Steuerelemente aus der Zwischenablage in die Gruppe einzufügen, markieren Sie zuerst die Elternkomponente und wählen dann BEARBEITEN | EINFÜGEN.
- Auch wenn Sie eine Komponente mit einem Doppelklick einfügen möchten, müssen Sie vorher die Elternkomponente auswählen, in die die neue Komponente eingefügt werden soll.

Wenn Sie Komponenten nachträglich verschieben, bleiben diese immer innerhalb ihrer bisherigen Elternkomponente und werden an deren Rand abgeschnitten. Insbesondere ist es nicht möglich, ein Steuerelement, das sich auf der freien Formularfläche befindet, in eine Gruppe aufzunehmen, indem Sie es einfach auf die Fläche der Gruppe schieben. Dadurch liegt das Steuerelement zwar auf (oder unter) der Gruppe, gehört aber nicht dazu. Um Steuerelemente in eine andere Gruppe zu verschieben, müssen Sie sie ausschneiden (BEARBEITEN | AUSSCHNEIDEN) und dann wie oben erwähnt in die gewünschte, vorher markierte Gruppe einfügen.

Oft ist es auch sinnvoll, Gruppen zu schachteln. So können Sie ein Formular beispielsweise durch *TPanel*-Komponenten, die mit *alClient* und *alTop* ausgerichtet sind, in

einen größenveränderlichen und einen festen Bereich unterteilen. Innerhalb dieser beiden Fensterhälften, die bereits ihrerseits Gruppen sind, können Sie dann weitere Untergruppen von Schaltern und anderen Steuerelementen unterbringen.

Gruppeneigenschaften

Steuerelement-Gruppen weisen die folgenden Eigenschaften auf, die Sie allesamt auch schon zur Entwurfszeit überprüfen können:

- Die Gruppenkomponente ist das Elternfenster der enthaltenen Elemente (das Formular bleibt weiterhin Besitzer und Ereignisempfänger). Die Kindelemente werden also am Rand des Gruppenfensters abgeschnitten, falls sie über diesen hinausragen.
- Wenn Sie das Gruppenfenster beim Entwurf oder zur Laufzeit verschieben, bewegen sich alle enthaltenen Elemente mit.
- Dementsprechend löschen Sie durch das Entfernen des Gruppenfensters auch alle darin enthaltenen Komponenten.
- Jede Gruppe hat eine eigene Tabulatorreihenfolge für die tastaturgesteuerte Navigation durch das Formular. Über das lokale Menü der Gruppenelemente und der Gruppenkomponente selbst gelangen Sie mit TABULATORREIHENFOLGE... in die Dialogbox, in der Sie die Reihenfolge der Gruppenelemente ändern können. Eine Gruppenebene höher (also in der Elternkomponente der Gruppe, normalerweise also im Formular) taucht die gesamte Gruppe nur als ein Punkt der Tabulatorreihenfolge auf. Die Tabulatorreihenfolge des Formulars wird also durch Gruppen erheblich übersichtlicher.
- Falls die Gruppe Radioschalter enthält, ist zu beachten: Innerhalb jeder Gruppe kann immer nur ein Radioschalter markiert sein.

Um zur Entwurfszeit einen genauen Überblick über die Gruppen bzw. über die gesamte Eltern-Kind-Hierarchie Ihres Formulars zu erhalten, wählen Sie ANSICHT ALS TEXT aus dem lokalen Menü des Formulars. Sie erhalten so eine durch Einrückung hierarchisch gegliederte Textdarstellung des Formulars mit all seinen Komponenten und Properties (siehe auch Kapitel 3.5.6, dort werden wir diese Darstellung editieren).

Gruppen im Programmcode

Wie sich die Eltern-Kind-Hierarchie im Programmcode ausdrückt und wie Sie sie zur Laufzeit verändern können, beschreibt Kapitel 3.2.2. Auch die Themen der folgenden beiden Abschnitte sind eng mit Steuerelement-Gruppen verknüpft: Wenn eine Ereignisbearbeitungsmethode mit mehreren Komponenten verknüpft werden soll (Kapitel

3.5.2), ist es oft auch sinnvoll, diese Komponenten in einer Gruppe zusammenzufassen. Darüber hinaus lassen sich die Komponenten einer Gruppe besonders einfach über Arrays ansprechen (Kapitel 3.5.3).

3.5.2 Gleichzeitige Behandlung mehrerer Komponenten

In Kapitel 1.8.6 wurde bereits gezeigt, wie sich mehrere Ereignisse einer oder mehrerer Komponenten in einer Methode bearbeiten lassen; allerdings wurde dort der Parameter *Sender* noch nicht benötigt. Dies ändert sich mit dem Beispielprogramm *StyleBtn* (Abbildung 3.4 auf Seite 363), in dem Sie den Rahmenstil und die Rahmen-Icons des Formulars (*TForm*-Properties *BorderStyle* und *BorderIcons*) zur Programmlaufzeit interaktiv verändern können und in dem zwei Schaltergruppen von jeweils einer Methode behandelt werden.

An dieser Stelle geht es um die erste dieser Gruppen: Unter der Überschrift *BorderStyle* finden Sie in einem *TPanel* sechs Radioschalter, die die möglichen Stile für die Formularumrandung angeben, sowie einen Markierungsschalter *AutoUpdate* und den Befehlsschalter *Update*, wobei es für die Funktion der letzteren beiden Elemente keine Rolle spielt, dass sie ebenfalls zur Gruppe gehören.

Solange der Schalter *AutoUpdate* markiert ist, soll das Formular die Umrandung immer sofort wechseln, sobald der Benutzer einen anderen *BorderStyle*-Schalter anwählt. Dafür müssen wir das *OnClick*-Ereignis dieser Schalter bearbeiten. So würde es für den Schalter *bsDialog* beispielsweise genügen, die folgende Methode zu schreiben:

```
procedure TBorderStyleForm.RadioButton1Click(Sender: TObject);
begin
  BorderStyle := bsDialog;
end;
```

Entsprechend würden dann die Methoden für die anderen Schalter aussehen. Wartungsfreundlicher, übersichtlicher und kleiner wird das Programm jedoch, wenn Sie in solchen Fällen alle Schalter mit einer einzigen Methode behandeln. Um zwischen den verschiedenen Schaltern unterscheiden zu können, erhält die Methode im Parameter *Sender* einen Hinweis (um nicht zu sagen einen Zeiger) auf den Schalter, auf den sich das Ereignis bezieht.

Um für alle Schalter am schnellsten eine gemeinsame Methode zu erstellen, wählen Sie im Entwurfsmodus alle Schalter gleichzeitig aus (Shift+Mausklicks). Der Objektinspektor zeigt daraufhin deren gemeinsame Properties an und erlaubt, dass Sie einzelne Properties für alle gewählten Schalter gleichzeitig verändern. Doppelklicken Sie also neben das *OnClick*-Ereignis, um den Rumpf der gemeinsamen Methode zu erhalten. Wenn der erste Schalter etwa den Namen *RadioButton1* hat, gibt der Objektinspektor den automatisch erzeugten Methoden für alle Schalter den Namen *RadioButton1Click*:

```
procedure TBorderStyleForm.RadioButton1Click(Sender: TObject);
begin
end;
```

Unterscheidung der Schalter in einer Methode *R12*

Die Schalter sind zwar schon durch verschiedene Beschriftungen, verschiedene Koordinaten und verschiedene Namen eindeutig unterscheidbar, wesentlich effektiver ist es aber, jedem Schalter darüber hinaus eine eindeutige Kennziffer zu geben. Hierfür ist das Allzweck-Property *Tag* bestens geeignet. Die Schalter der *BorderStyle*-Gruppe erhalten die *Tag*-Werte 0 bis 5, die genau den ordinalen Zahlenwerten der Stilkonstanten entsprechen, die mit den Schaltern eingeschaltet werden sollen. So ist z.B. der ordinale Wert von *bsSizeable* gleich zwei, umgekehrt ergibt die Typenumwandlung *TFormBorderStyle(2)* den Wert *bsSizeable* (siehe ordinale Typen in Kapitel 2.4.1).

Wir können nun also innerhalb der Methode für alle Schalter wie folgt vom *Tag*-Property des gedrückten Schalters auf den zu setzenden *BorderStyle* schließen:

```
BorderStyle := TFormBorderStyle(RadioButton.Tag);
```

Offen ist allerdings noch, um welchen *RadioButton* es sich handelt. Hier kommt nun der Parameter *Sender* ins Spiel, den diese Methode (wie jede andere Ereignisbearbeitungsmethode auch) erhält. Er gibt an, bei welcher Komponente das Ereignis aufgetreten ist. Bevor wir jedoch auf das *Tag*-Property von *Sender* zugreifen können, müssen wir dem Compiler mitteilen, dass der Parameter *Sender* nicht nur ein *TObject* ist, sondern eine Komponente, die ein *Tag*-Property enthält, also ein *TRadioButton* oder mindestens ein Objekt des Typs *TComponent*. Dies erreichen wir mit dem Ausdruck *Sender as TRadioButton*::

```
procedure TBorderStyleForm.RadioBsDialogClick(Sender: TObject);
begin
  { Funktion nur ausführen, wenn "AutoUpdate" markiert ist: }
  if AutoUpdate.Checked then
    with Sender as TRadioButton do
      BorderStyle := TFormBorderStyle(Tag);
end;
```

Klicken Sie zur Laufzeit beispielsweise auf den Schalter *bsSizeable*, erhält die obige Methode im Parameter *Sender* die Komponente *RadioBsSizeable*, deren Property *Tag* wir auf 2 gesetzt haben. Die Typenumwandlung *TFormBorderStyle(Tag)* führt zum Rahmenstil *bsSizeable*, der dem Formular-Property *BorderStyle* zugewiesen wird. Das Formular ändert zur Laufzeit sofort sein Aussehen.

Die Umwandlung von *Sender* in *TRadioButton* setzt natürlich voraus, dass Sie diese Methode nur mit Ereignissen verknüpfen, die auch von einem *TRadioButton* stammen.

Eine *with*-Anweisung wie die oben gezeigte kann übrigens zu Verwechslungen führen: Wenn *TRadioButton* nicht nur das Property *Tag*, sondern auch einen Bezeichner namens *BorderStyle* besäße, würde die obige Anweisung nicht den *BorderStyle* des Formulars, sondern den des Radioschalters verändern. Ohne die *with*-Anweisung ergäbe sich wie folgt ein eindeutiges Bild:

```
BorderStyle := TFormBorderStyle((Sender as TRadioButton).Tag);
```

Das Kapitel 3.5.3 beschreibt die Methode, die die manuelle Aktualisierung des Rahmenstils durchführt (falls der Schalter *AutoUpdate* nicht markiert ist).

Eine Variation

Die ersten vier Schalter der Gruppe *BorderIcons* können auf eine ähnliche Weise behandelt werden. Auch in dieser Schaltergruppe soll jeder Klick auf einen Schalter sofort zu einem Ergebnis führen. Die vier ersten Markierungsschalter entsprechen je einem Flag, das in der Menge *TBorderIcons* enthalten sein kann (zu diesen *TForm*-Properties siehe Kapitel 3.4.2).

Die Methode für die Markierungsschalter geht in zweierlei Hinsicht einen anderen (übersichtlicheren, aber nicht unbedingt besseren) Weg als die zuletzt gezeigte Methode für die Radioschalter:

- Sie speichert den in eine *TCheckBox* umgewandelten Parameter *Sender* in der lokalen Variable *Button*, über die sie auf alle Felder des Markierungsschalters explizit zugreifen kann.
- Sie verwendet das *Tag*-Property der Schalter nur zur Unterscheidung und nicht auch noch zur Typenumwandlung der Art *Flag := TBorderIcon(Button.Tag)* (für eine solche Typenumwandlung müssten die *Tag*-Properties auch zuerst in eine Reihenfolge gebracht werden, die der von *TBorderIcon* entspricht).

Schließlich bietet sich bei dieser Methode eine gute Gelegenheit, den »grauen Zustand« der Markierungsschalter auch einmal in diesem Buch zu testen:

```
procedure TBorderStyleForm.CheckBox1Click(Sender: TObject);
var
  Flag: TBorderIcons;
  Button: TCheckBox;
  NewSet: TBorderIcons;
begin
  Button := Sender as TCheckBox;
  case Button.Tag of
    1: Flag := [biMaximize];
    2: Flag := [biMinimize];
    3: Flag := [biSystemMenu];
    4: Flag := [biHelp];
  end;
```

```
    if Button.Checked
      then BorderIcons := BorderIcons+Flag
      else BorderIcons := BorderIcons-Flag;
    Alle.State := cbGrayed; (* nach Wahl eines einzelnen Icons sind
        wahrscheinlich weder Alle gewählt, noch Alle nicht gewählt.
        daher wird der Schalter Alle grau dargestellt. *)
  end;
```

Ist die Checkbox *Sender* markiert, fügt die Methode dem Mengen-Property *BorderIcons* die neue einelementige Menge *Flag* hinzu, ansonsten zieht sie sie davon ab. Da die Funktionen *Exclude* und *Include* (siehe *Mengen*, Kapitel 2.4.5) nicht mit Mengen-Properties arbeiten können (sie erwarten einen Variablenparameter), muss das einzelne Flag auf die oben gezeigte Weise in die Properties aufgenommen werden.

Der letzte Schalter der Gruppe mit der Aufschrift *Alle* benötigt eine Sonderbehandlung, die im folgenden Kapitel vorgestellt wird.

3.5.3 Steuerelemente in Arrays

Wenn mehrere Komponenten nicht nur einen ähnlichen Zweck, sondern auch den gleichen Typ haben, bietet es sich an, sie nicht nur in einer Gruppe zusammenzufassen, sondern auch in einem Array unterzubringen. Dies hat beispielsweise den Vorteil, dass Sie alle Elemente in einer einfachen Schleife ansprechen können.

Wir beschäftigen uns an dieser Stelle weiter mit dem Beispielprogramm *StyleBtn*, das bereits in Abbildung 3.4 auf Seite 363 gezeigt wurde.

Selbst gemachte Arrays *R17*

Durch die *Tag*-Properties erhielten die Schalter des Programms bereits in Kapitel 3.5.2 eine numerische Reihenfolge; es ist dadurch aber noch nicht möglich, sie in einer Schleife anzusprechen. Wenn Sie eine Schleife schreiben wollen (zwei Beispiele folgen), können Sie in der Formulardeklaration ein Array für die Schalter reservieren:

```
type
  TBorderStyleForm = class(TForm)
  ...
  private
    StyleButtons: array[TFormBorderStyle] of TRadioButton;
  ...
  end;
```

Durch diese Array-Deklaration erhalten Sie ein vierelementiges Array mit den Indizes des Typs *TFormBorderStyle* (*bsNone* bis *bsSizeToolWin*). Jeder Schalter soll an der Array-Position gespeichert werden, die seinem *Tag* und dem entsprechenden Rahmenstil entspricht. Selbst definierte Arrays müssen natürlich auch selbst gefüllt werden, eine günstige Gelegenheit dazu bietet das *OnCreate*-Event des Formulars:

```
procedure TBorderStyleForm.FormCreate(Sender: TObject);
begin
  StyleButtons[bsNone] := RadioBsNone;
  StyleButtons[bsSingle] := RadioBsSingle;
  StyleButtons[bsSizeable] := RadioBsSizeable;
  ...
```

Auch hier nutzen wir wieder die Tatsache, dass die Variablen *RadioBs...* intern Zeiger sind, denn durch das Kopieren dieser Variablen in ein Array erzeugen wir keine Doppelgänger für diese Schalter, sondern setzen lediglich ein Array von Verweisen auf die Schalterobjekte.

Nun zu den Anwendungen der Arrays: Als Erstes soll die bisherige automatische Anpassung des Rahmenstils bei jeder Veränderung der Radioschalter durch den Befehlsschalter *Update* ersetzt werden, den der Benutzer drücken muss, um die Änderungen zu bestätigen und sichtbar werden zu lassen. Die Methode *ManualUpdateClick* (verknüpft mit dem *OnClick*-Ereignis dieses Befehlsschalters) muss also herausfinden, welcher Schalter gedrückt ist, und dementsprechend den Rahmenstil ändern. Hierzu könnte man in den *OnClick*-Ereignissen der verschiedenen Schalter jeweils das *Tag*-Property des neu gesetzten Schalters in einer Variablen zwischenspeichern. Die Methode *ManualUpdateClick* soll jedoch völlig eigenständig sein und nur das Schalterarray verwenden:

```
procedure TBorderStyleForm.ManualUpdateClick(Sender: TObject);
var
  i: TFormBorderStyle;
begin
  { mit Low und High werden unabhängig von der Delphi-Version
    alle TFormBorderStyle-Werte durchlaufen: }
  for i := Low(TFormBorderStyle) to High(TFormBorderStyle) do
    if StyleButtons[i].Checked then begin
      BorderStyle := i;
      exit;
    end;
end;
```

Diese Methode durchläuft alle Schalter in einer Schleife und ändert den Stil *BorderStyle*, wenn sie den einzigen gesetzten Schalter (dessen *Checked*-Property gesetzt ist) gefunden hat. Danach überspringt sie die Überprüfung der restlichen Schalter durch die Anweisung *exit*.

Hinweis: Dieses Beispiel soll die Verwendung eines Komponenten-Arrays demonstrieren. Für Radioschalter sollten Sie jedoch die Komponente *TRadioGroup* verwenden, die noch einfacher verwendbar ist als ein Array von Radioschaltern. Statt der fünf Zeilen wäre in der obigen Methode nur eine Zeile notwendig, wenn die einzelnen Optionen in einer *TRadioGroup* untergebracht worden wären:

```
BorderStyle := TFormBorderStyle(RadioGroup.ItemIndex);
```

Fertig vorgegebene Arrays *R18*

Für ein zweites Array-Beispiel wenden wir uns nun der zweiten Gruppe des Programms *StyleBtn* zu: Es handelt sich dieses Mal um eine *TGroupBox*-Komponente, die fünf Markierungsschalter enthält. Wir werden nun eine weitere Vereinfachung durchführen, die allerdings auch mit zusätzlichen (kalkulierbaren) Gefahren verbunden ist. Die vier ersten Markierungsschalter sollen über den Schalter *Alle* gleichzeitig umgestellt werden können. Für das *OnClick*-Ereignis von *Alle* ist die folgende Methode zuständig:

```
procedure TBorderStyleForm.AlleClick(Sender: TObject);
var
  i: Byte;
begin
  for i := 0 to 4 do
    with GroupBox1.Controls[i] as TCheckBox do
      Checked := Alle.Checked;
end;
```

Diese Methode nutzt die Tatsache, dass alle Markierungsschalter als Kindfenster der *GroupBox1* in deren Array-Property *Controls* liegen (zum *Controls*-Array siehe Kapitel 3.2.2). Wenn die Gruppe nur diese fünf Schalter enthält, ist es auch nicht problematisch, die Indizes derselben herauszufinden – es müssen die Indizes 0 bis 4 sein. Die oben gezeigte Schleife setzt *alle* Schalter auf den Markierungsstatus von *Alle*, also auch *Alle* selbst. Wenn Sie jedoch nicht alle Elemente einer Gruppe auf einmal behandeln wollen, so müssen Sie den Index der einzelnen Elemente kennen. In diesem Beispiel lässt sich nicht ohne weiteres sagen, dass der Schalter *Alle* den Index 4 hat, nur weil er optisch an vierter Stelle untergebracht ist.

Die Indizes des Controls-Arrays *R8*

Es ist nicht möglich, die Indizes im Formular-Designer gezielt zu überprüfen: In der ERSTELLUNGSREIHENFOLGE tauchen die visuellen Komponenten nicht auf und die TABULATORREIHENFOLGE hat nichts mit den Indizes zu tun. Tatsächlich entsprechen die Indizes genau der Reihenfolge, in der die Steuerelemente in die Gruppe eingefügt wurden. Schneiden Sie beispielsweise das zweite von vier Elementen in die Zwischenablage aus und fügen es wieder ein, so wird es am Ende des *Controls*-Arrays wieder

eingefügt, wo es bei vier Elementen den Index 3 erhält. Gleiches bewirkt übrigens der lokale Formular-Menüpunkt NACH VORNE SETZEN, während NACH HINTEN SETZEN dem Element den Index 0 zuweist.

Um die Indizes aller Elemente festzustellen, können Sie das Formular in der Textdarstellung untersuchen (ANSICHT ALS TEXT im lokalen Menü des Formulars wählen). Die Reihenfolge der Objektdeklarationen in der Textdarstellung entspricht auch der Reihenfolge der Steuerelemente im *Controls*-Array. Durch Verschieben einzelner Komponenten können Sie in der Textform sogar die Indizes der Komponenten verändern (*Vorsicht:* Eine Verschiebung der falschen Textteile kann zur Beschädigung der `dfm`-Datei führen!).

Falls Sie das *Controls*-Array an einem Index lesen wollen, an dem sich gar kein Kindfenster / Steuerelement befindet, oder wenn Sie ein Element des Arrays in einen inkompatiblen Typen umwandeln, löst die VCL eine Exception aus, die das gerade bearbeitete Ereignis abbricht, die die Anwendung aber ansonsten sicher weiterlaufen lässt.

Hinweis: Die Reihenfolge der Komponenten im *Controls*-Array eines Formulars können Sie auch aus dem im Anhang vorgestellten Formular-Explorer ersehen.

3.5.4 Mehrseitige Dialoge

Zu den Standardkomponenten gehören auch solche, mit denen Sie Ihr Formular in mehrere Seiten unterteilen können. Ein mehrseitiger Dialog kann eine ganze Palette von einseitigen Dialogen ersetzen. Der große Vorteil liegt darin, dass zum Wechseln zwischen zwei Seiten nur ein (Maus-)Tastendruck notwendig ist, während zum Wechseln zwischen verschiedenen (modalen) Dialogen zuerst der eine geschlossen und dann der nächste wieder geöffnet werden muss.

Die Komponentenauswahl

Die Komponenten zur Gestaltung mehrseitiger Dialoge lassen sich unterteilen in alte Komponenten, die noch von 16-Bit-Delphi stammen und unter anderem natürlich zur Abwärts-Kompatibilität weiter existieren, und in die beiden folgenden neueren Komponenten für die mit Windows 95 eingeführten standardmäßigen mehrseitigen Dialoge:

- *TPageControl* bietet den mit Abstand komfortabelsten Zugang zu mehrseitigen Dialogen, denn sie lässt Sie die *TTabSheet*-Komponenten, die die einzelnen Seiten darstellen, schon zur Entwurfszeit als einzelne Komponenten behandeln. Im Gegensatz zum alten *TTabbedNotebook* können Sie die Seiten schon im Entwurfsmodus ganz normal mit der Maus umschlagen.

- *TTabControl* ist grafisch genauso gestaltet wie *TPageControl*, ein Register ist also vorhanden, allerdings fehlt die Unterstützung mehrerer Seiten. Damit bietet sich diese Komponente für Fälle an, in denen auch ohne Seitenunterteilung zwischen verschiedenen Dingen gewählt werden soll, z.B. zwischen verschiedenen Ansichten einer Grafik, die immer in dieselbe Komponente eingezeichnet wird.

Die beiden schon von Delphi 1 bekannten Formen mehrseitiger Dialoge lassen sich wie folgt realisieren:

- Durch gleichzeitiges Verwenden je einer Instanz der Klassen *TTabSet* und *TNotebook* erhalten Sie Dialoge wie die mehrseitigen Dialoge der IDE von Delphi 1. *TNotebook* ist nur zur Aufteilung der Elemente auf verschiedene Seiten notwendig und nicht fest an *TTabSet* gebunden. Statt letzterer Komponente können Sie also auch Listen, Schalter und beliebige andere Elemente einsetzen, um zwischen den Seiten zu wechseln. Diese Flexibilität erhalten Sie unter Win32 auch durch die *TTabControl*-Komponente.
- Die Komponente *TTabbedNotebook* vereinigt die Funktion von *TNotebook* und *TTabSet* unter einer anderen optischen Gestaltung, wie Sie in Abbildung 3.7 erkennen können. Die Register dieser Komponente werden in mehreren Zeilen angeordnet, wenn sie nicht alle nebeneinander passen. Dadurch bleiben auch ohne Scrollen durch das Register alle Seiten direkt zugänglich.

Anhand des Beispielprogramms *NBDemo* (Abbildung 3.7) werden wir beide Methoden ausprobieren.

Verwendung der TPageControl-Komponente *R23*

In der Tat ist *TPageControl* als Komponente so einfach handhabbar, dass wir uns hier nur kurz mit ihr zu beschäftigen brauchen. Um einen mehrseitigen Dialog mit Hilfe dieser Komponente zu entwerfen, können Sie die folgenden Schritte ausführen:

- Im einfachsten Fall fertigen Sie eine Kopie des *Registerdialogs* der Objektablage an (DATEI | NEU, Seite *Formulare*, Auswählen des Radioschalters *Kopieren* und des Icons *Registerdialog*).

  Sie erhalten dann nicht nur drei Demonstrationsseiten und die drei üblichen Schalter *OK*, *Abbrechen* und *Hilfe* in ihrer üblichen Position, sondern auch zwei unsichtbare *TPanel*-Komponenten, die das *PageControl* auch bei Größenänderung des Dialogs immer anpassen – und das mit einem stabilen Rand zu den Grenzen des Formulars. Alternativ dazu können Sie natürlich auch selbst eine *TPageControl*-Komponente in Ihr Formular einfügen, aus deren lokalem Menü Sie mit NEUE SEITE neue Seiten erstellen können.

Abbildung 3.7: Ansichten des Beispielprogramms

- Die einzelnen Seiten des *PageControls* stehen Ihnen nun in Form von *TTabSheet*-Komponenten zur Verfügung. Durch einen Klick ins Innere des *PageControls* wählen Sie jeweils das *TabSheet* für die aktuelle Seite aus. Über deren *Caption*-Property bestimmen Sie die Seitenbeschriftung des *PageControls*, über den *PageIndex* die Reihenfolge der Seiten, über das Property *Enabled* können Sie alle Elemente dieser Seite gleichzeitig aktivieren und deaktivieren. Löschen Sie ein *TabSheet* mit Hilfe von Entf, so entfernen Sie damit eine Seite des *PageControls*.
- Um die einzelnen Seiten mit Komponenten zu füllen, schlagen Sie sie einfach auf und zeichnen die Komponenten ein.

Erwähnenswert sind auch die Ereignisse von *TPageControl*: Es gibt nicht nur das übliche Ereignis *OnChange*, das Sie nach einer Seitenanwahl benachrichtigt, sondern auch ein Ereignis *OnChanging*, das Ihnen *vor* dem Umschlagen einer Seite erlaubt, weitere Maßnahmen zu treffen oder das Umschlagen gar zu verhindern (durch Setzen des Methodenparameters *AllowChange* auf *False*).

Das Beispielprogramm

Das Beispielprogramm, das Sie unter dem Namen *NBDemo* auf der CD-ROM finden, verwendet zu Demonstrationszwecken zwei verschiedene Möglichkeiten der Dialoggliederung in einem einzigen Formular (Abbildung 3.7): Auf den Seiten des äußeren *TPageControl* befindet sich eine (innere) *Notebook*-Komponente mit den fünf phantasievollen Seiten *Allgemeines*, *Details*, *Details-Editor*, *Details-Grafik* und *Drucker*. Dieses innere Notebook ist immer sichtbar, egal, welche Seite des äußeren *PageControls* gerade aufgeschlagen ist. (Damit es immer sichtbar ist, befindet sich die *Notebook*-Komponente *auf* dem *PageControl*, ist also keiner von dessen Seiten als Kindkomponente untergeordnet.)

Jede Seite des äußeren *PageControls* demonstriert eine eigene Variation der Seitenansteuerung für das innere *Notebook*. Durch diese Schachtelung werden der Entwurf und die Programmierung jedoch nicht erschwert.

Verwendung der Win32-unabhängigen Komponenten

Egal, ob Sie *TTabbedNotebook* oder eine Kombination aus *TNotebook* und anderen Komponenten verwenden, die grundsätzliche Vorgehensweise ist bei beiden Methoden sehr ähnlich:

- Fügen Sie eine *TNotebook* / *TTabbedNotebook*-Komponente nach Belieben in das Formular ein und stellen Sie eventuell ihr *Align*-Property auf *alClient*, um ein Aussehen wie im Beispielprogramm zu erhalten. Falls Sie bei der *alClient*-Einstellung einen automatischen Rand haben wollen, fügen Sie die Komponente nicht direkt in das Formular ein, sondern in eine *Panel*-Komponente, deren *Border*-Property Sie auf den gewünschten Rand setzen.

- Per Voreinstellung enthalten beide Notebook-Typen bereits eine erste Seite (die in einem *TNotebook* jedoch zunächst nicht sichtbar ist). Um deren Namen anzupassen und weitere Seiten hinzuzufügen, müssen Sie die Stringliste *Pages* editieren. Öffnen Sie dazu den Property-Editor für das Property *Pages* und geben Sie die Seitenbezeichnungen ein, unter Umständen kombiniert mit einem Hilfekontext. Im Falle eines *TabbedNotebooks* werden die Änderungen sichtbar, sobald Sie diesen Dialog schließen.

- Jetzt können Sie den Inhalt der einzelnen Seiten entwerfen, indem Sie eine der leeren Seiten aufschlagen und die dazugehörenden Komponenten darin platzieren. Zwar funktionieren die Registerzungen im Entwurfsmodus nicht zum Umschlagen der Seiten, dafür haben Sie aber gleich drei andere Möglichkeiten, die aktuelle Seite zu wechseln: indem Sie sie in der aufklappbaren Liste des Properties *ActivePage* auswählen, indem Sie die gewünschte Seitenzahl unter dem Property *PageIndex* eintragen oder indem Sie aus dem lokalen Menü der Notizbuch-Komponente die

Menüpunkte NÄCHSTE SEITE und VORHERIGE SEITE verwenden. Beide Notizbuch-Arten verstecken automatisch alle Elemente der aktuellen Seite, wenn Sie eine andere Seite aufschlagen. Insofern sind sie beide schon zur Entwurfszeit voll funktionsfähig, nur die Mausklicks auf die Registerzungen funktionieren noch nicht.

Im Falle von *TTabbedNotebook* sind Sie nun mit dem Entwurf fertig und müssen sich nicht weiter um die Verwaltung der Seiten kümmern. Ohne eine Zeile von Ihnen geschriebenen Codes können Sie die Seiten zur Programmlaufzeit mit der Maus umschlagen. Kapitel 3.5.5 beschäftigt sich mit den weiteren Maßnahmen, die im Falle von *TNotebook* erforderlich sind.

Hinweis: Im Gegensatz zu *TPageControl* blättern weder *TTabbedNotebook* noch *TTabSet* die Seiten um, wenn der Benutzer die dazu üblichen Tastenkombinationen Strg+Tab und Shift+Strg+Tab drückt. Um diese Funktion zu implementieren, müssen Sie die *KeyPreview*-Funktion des Formulars benutzen, wie am Ende von Kapitel 5.8.1 beschrieben (siehe auch Quelltext von *NBDemo*).

Programmierung und Properties

An der Programmierung der Steuerelemente auf den verschiedenen Seiten ändert sich nichts: Alle Komponenten sind im Programm immer ansprechbar. Allerdings ist etwas mehr Vorsicht geboten als bei einseitigen Dialogen: Wenn Sie zwei zueinander gehörende Kontrollelemente, zum Beispiel einen Schalter und eine Liste, die durch den Schalter verändert wird, auf verschiedenen Seiten unterbringen, beeinträchtigt das zwar nicht die Funktion des Programms, dafür aber den Gemütszustand des Anwenders.

Da eine *TTabbedNotebook*-Komponente die Seitenanwahl völlig automatisch steuert, verursacht sie überhaupt keinen Mehraufwand an Programmierung. Bei *TNotebook* werden pro Dialog ca. zwei zusätzliche Zeilen Code erforderlich, denen wir uns im nächsten Abschnitt zuwenden. Zuerst jedoch eine Zusammenfassung der Properties von *TNotebook* und *TTabbedNotebook*, die Informationen über die Seiten und die gewählte Seite enthalten, die meisten davon wurden bereits in der schrittweisen Aufbauanleitung genannt.

Property	Aufgabe
ActivePage	Name der aktuellen Seite. Dieses Property befreit Sie von der Aufgabe, sich die Seitenindizes merken zu müssen.
PageIndex	Index der gerade aufgeschlagenen Seite
Pages	*TStrings*-Liste der Seitennamen und zugehörigen Hilfekontextnummern, zur Entwurfszeit in einem eigenen Editor bearbeitet.

Property	Aufgabe
TabsPerRow	nur bei *TTabbedNotebook*: Anzahl der Registerzungen in einer Zeile (*TTabbedNotebook* gibt allen Registerzungen die gleiche Breite, ohne den Platzanspruch des Textes zu bedenken.)

Seitenübergreifende Elemente

Komponenten, die immer sichtbar sein sollen, unabhängig davon, welche Seite gerade aufgeschlagen ist, werden in vielen Anwendungen außerhalb der Seiten untergebracht, so z.B. die Schalter *Ok*, *Abbruch* und *Hilfe* in den Dialogen der Delphi-IDE. Sie können jedoch auch innerhalb der Seiten Komponenten unterbringen, die immer angezeigt werden.

Dafür müssen Sie sich eines einfachen Tricks bedienen: Fügen Sie diese Komponenten so ein, dass Delphi sie nicht der *PageControl* oder *Notebook*-Komponente unterordnet. Positionieren Sie sie also zuerst außerhalb jeglicher Komponenten, die Delphi als *Parent* missverstehen könnte, so dass das Formular zum *Parent* dieser Komponenten wird, und schieben Sie die Komponenten erst danach auf die Seitenfläche. (Wenn das gesamte Formular durch die Seiten-Komponente belegt wird, ist das so einfach nicht möglich. Wählen Sie dann das Formular aus der aufklappbaren Liste direkt unter der Objektinspektor-Titelzeile aus und fügen Sie die Komponente mit einem Doppelklick in die Komponentenpalette in das Formular ein.)

Klassen, die wie z.B. *TBevel* von *TGraphicControl* abstammen, können Sie leider nicht auf verschiedenen Seiten gleichzeitig anzeigen, da diese immer unterhalb aller *TWinControl*-Steuerelemente und somit auch unter den PageControls und Notebooks liegen und daher unsichtbar werden, wenn sie keine Kindelemente dieser Komponenten sind.

Ressourcenbedarf

Der Vorteil der gleichzeitigen Ansprechbarkeit der Komponenten aller Seiten bringt auch einen Nachteil mit sich, denn Windows belegt für jedes Fenster internen Speicher, der in der Ressourcen-Anzeige (bzw. im Programm-Manager) und verschiedenen anderen Utilities als »Systemressourcen« angegeben wird. Jedes Objekt einer Ableitung von *TWinControl* belegt auf diese Weise System-Ressourcen (Ressourcen des User-Moduls), manche Steuerelemente können eventuell auch nach Ressourcen des GDI-Moduls verlangen.

Der springende Punkt dabei ist, dass eine Komponente auch dann Ressourcen verbraucht, wenn sie nicht sichtbar ist, also sich auf einer nicht aufgeschlagenen Seite befindet. (Bei Verwendung von *TNotebook* beginnt der Ressourcenverbrauch allerdings erst dann, wenn die Komponente zum ersten Mal sichtbar wird. Wenn sie danach wieder umgeblättert wird, bleiben die Ressourcen belegt.) Da unter Windows 95/98 wie

noch zu Zeiten von 16-Bit-Windows nur zweimal ganze 64 KByte der Systemressourcen zur Verfügung stehen (je 64 KByte Ressourcen des User- und des GDI-Moduls), kann auch der größtmöglich ausgebaute Speicher nicht verhindern, dass es hierbei zu Engpässen kommt.

Die Delphi-IDE, die ja selbst auch mit der VCL programmiert ist, zeigt jedoch in ihren mehrseitigen Dialogen, dass trotzdem einige Seiten zustande kommen können. Wenn Sie den Dialog der Umgebungsoptionen von Delphi 1 öffnen und jede Seite einmal aufschlagen, verringern sich unter Windows 3.1 die freien User-Ressourcen um ca. 25%. Tests mit Delphi 4 ergaben unter Windows 98 einen Ressourcenverbrauch von 18% (unter Delphi 5 wurden einige Seiten aus dem Umgebungsdialog ausgelagert, weshalb ein wiederholter Test nicht so interessant erscheint). Das zeigt zwar, dass entweder der Umgebungsdialog von Delphi oder Windows 98 oder beide sparsamer mit den Ressourcen umgehen, trotzdem würde auch Windows 98 kaum fünf solcher Dialoge gleichzeitig verkraften.

Damit auch in Ihren Delphi-Anwendungen die Ressourcen eines Dialogs bei dessen Ende freigegeben werden, müssen Sie eine der in Kapitel 3.4.4 beschriebenen Maßnahmen ergreifen, sonst macht die VCL den Dialog beim Schließen nur unsichtbar. In diesem Fall ereilt die Ressourcen des Dialogs das gleiche Schicksal wie die Ressourcen der weggeblätterten Seiten: Sie bleiben weiterhin in der Hand der Komponenten, obwohl sie am Bildschirm nicht sichtbar sind.

Hinweis: Sogar 32-Bit-Anwendungen unter Windows 95 leiden also unter der alten 64 KByte-Grenze. Sie erhalten jedoch gegenüber 16-Bit-Anwendungen einen Vorteil bezüglich der Ressourcenverwaltung: Nach Beendigung einer 32-Bit-Anwendung gibt Windows die von der Anwendung noch nicht freigegebenen Ressourcen automatisch frei, so dass die knappen Ressourcen wenigstens nicht durch fehlerhafte Programme dauerhaft verloren gehen können. In 32-Bit-Delphi kann der integrierte Debugger daher die laufende Anwendung jederzeit abbrechen, ohne dass er wie Delphi 1 eine Warnung vor einem Ressourcenverlust anzeigt.

3.5.5 Maximale Flexibilität mit TNotebook

Erheblich flexibler als *TPageControl* und *TTabbedNotebook*, aber etwas arbeitsaufwändiger ist die Komponente *TNotebook*, die ebenfalls mehrere Seiten enthalten kann, aber völlig unsichtbar ist und zur Laufzeit nur dann zwischen den Seiten umschaltet, wenn Sie sie im Programm dazu auffordern. Um dem Benutzer die Anwahl der Seiten zu ermöglichen, müssen Sie weitere Steuerelemente bereitstellen.

Im Beispielprogramm *NBDemo* wird die *TNotebook*-Komponente (die sich innerhalb der *TPageControl*- bzw. *TTabbedNotebook*-Komponente befindet) durch ein *TBevel*-Element abgegrenzt, da man ansonsten den Rand ihrer Seiten nicht erkennen könnte.

Seitenauswahl für TNotebook

Während Sie die Seiten einer *TNotebook*-Komponente zur Entwurfszeit (wie in Kapitel 3.5.4, Abschnitt *Verwendung der Win32-unabhängigen Komponenten* beschrieben) auf drei verschiedene Arten umschlagen können, müssen Sie für die Laufzeit erst noch ein geeignetes Steuerelement hinzufügen und dieses zum Umschlagen der Seiten programmieren. Am häufigsten wird hierzu die Komponente *TTabSet* verwendet, deren Beschriftung ebenfalls in einer Stringliste (*Tabs*) gespeichert ist. Auch diese Stringliste können Sie leicht schon beim Entwurf angeben; da Sie aber die Beschriftung wahrscheinlich bereits in *Notebook.Pages* abgegeben haben, ist es einfacher, wenn Sie sie von dort zur Laufzeit in das *TabSet* kopieren (ganz ohne *TNotebook.Pages* kommen Sie sowieso nicht aus, da Sie sonst nur ein einseitiges Notebook erhalten).

Da *TTabSet.Tabs* und *TNotebook.Pages* kompatibel miteinander sind (beide haben den abstrakten Typ *TStrings*), genügt dazu eine einzige Zeile, die gut in die Methode für das *OnCreate*-Ereignis des Formulars passt:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  TabSet.Tabs := Notebook.Pages;
end;
```

Doch damit nicht genug: Zur vollständigen Funktion dieser Mehrseiten-Einheit ist noch eine zweite Zeile notwendig. Jedes Mal, wenn der Benutzer auf das Register klickt, setzt *TTabSet* das Property *TabIndex* auf den Index der neuen Seite und erzeugt ein *OnClick*-Event. Dieses können Sie nun zum Anlass nehmen, dem *Notebook* die neue Seitenzahl mitzuteilen:

```
procedure TForm1.TabSet1Click(Sender: TObject);
begin
  NoteBook.PageIndex := TabSet.TabIndex;
end;
```

Mit diesen beiden Methoden funktioniert der aus zwei Teilen zusammengesetzte mehrseitige Dialog bereits perfekt (bis auf die in Kapitel 3.5.4 bereits erwähnte fehlende Tastatursteuerung per Strg+Tab bzw. Shift+Strg+Tab). Auch in den anderen Seitenanwahlverfahren, die im nächsten Abschnitt beschrieben werden, sind zwei Schritte ähnlich den obigen beiden Methoden notwendig:

- Die Namen der *Notebook*-Seiten müssen auf einem zweiten Steuerelement sichtbar werden, meistens werden sie dazu bei der *OnCreate*-Initialisierung einfach aus *Notebook.Pages* kopiert.
- Wenn der Benutzer auf das Seitenauswahl-Steuerelement klickt, müssen Sie den Seitenindex des Notebooks anpassen.

Alternative Seitenauswahlverfahren *R24*

Statt des Registers sind auch andere Methoden zur Seitenansteuerung möglich, z.B. über eine normale Liste oder eine Liste mit expandierbaren Zweigen, die bei großer Seitenzahl noch übersichtlicher ist als ein einfaches Register (solche expandierbaren Listen verwendet beispielsweise Borland C++ 5.0).

Das Beispielprogramm demonstriert drei weitere Verfahren zur Auswahl der Seite: über eine Liste, eine *TOutline*-Komponente[9] und eine Schalteransammlung (siehe Abbildung 3.7). Die Einträge der Liste und des *Outlines* können Sie so schnell über eine Zuweisung von *Notebook.Pages* kopieren wie im letzten Abschnitt die Beschriftungen des *TabSets*. Die folgende *FormCreate*-Methode sorgt mit *ChangeLevelBy*-Aufrufen zusätzlich dafür, dass die Elemente des Outlines in einer hierarchischen Struktur angeordnet werden:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { die Seitennamen in drei verschiedene Komponenten übertragen: }
  TabSet.Tabs := Notebook.Pages;
  ListBox.Items := Notebook.Pages;
  Outline.Lines := Notebook.Pages;
  { Strukturierung des Outlines: }
  Outline.Items[2].ChangeLevelBy(1);
  Outline.Items[3].ChangeLevelBy(1);
  Outline.Items[3].ChangeLevelBy(1);
  Outline.Items[4].ChangeLevelBy(1);
end;
```

Die Beschriftung der Schalter könnte zwar auch automatisch in *FormCreate* stattfinden, wurde aber in diesem Beispiel schon zur Entwurfszeit fest in den *Caption*-Properties verankert.

Für jede der drei neuen Seitenanwahl-Methoden müssen wir nun wieder jeweils eine einzige weitere Programmzeile schreiben:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  NoteBook.PageIndex := Listbox.ItemIndex;
end;

procedure TForm1.Outline1Click(Sender: TObject);
begin
  { TOutline beginnt bei 1 zu zählen: }
  NoteBook.PageIndex := Outline.SelectedItem-1;
end;
```

9 *TOutline* ist veraltet. Damit dieses Beispielprogramm unter allen Delphi-Versionen funktioniert, verwendet es dennoch nicht die moderne *TTreeView*-Komponente.

```
procedure TForm1.AllgemeinesClick(Sender: TObject);
begin
  NoteBook.PageIndex := (Sender as TButton).Tag;
end;
```

Die Liste und das Outline geben mit *ItemIndex* bzw. *SelectedItem* einen Index an, der mit *Notebook.PageIndex* »kompatibel« ist, also genügen die oben gezeigten Zuweisungen. Bei den Schaltern müssen wir etwas nachhelfen, indem wir ihnen durch Verändern des Properties *Tag* zu einer Kennung verhelfen (auch das ist im Beispielprogramm bereits beim Entwurf des Formulars geschehen). Danach können Sie das *OnClick*-Ereignis aller Schalter mit einer einzigen Methode (hier *AllgemeinesClick*) verknüpfen.

Damit haben die Kapitel 3.5.4 und 3.5.5 nun alle fünf Methoden des Beispielformulars vollständig gezeigt. Der größte Teil der Arbeit beim Entwurf eines mehrseitigen Dialogs findet also, wie man es auch erwarten darf, visuell statt.

3.5.6 Formulardateien im Textmodus editieren

So komfortabel der Entwurf von Formularen in Delphi ist – es gibt doch einige Arbeitsabläufe, die visuell nicht oder nur sehr schwierig durchzuführen sind, etwa durch teilweisen Neuaufbau des Formulars oder umfangreiche Zwischenablageoperationen. Zu diesen Arbeitsabläufen gehören z.B.:

- Auswechseln einer Komponente, die andere Komponenten enthält, wobei diese enthaltenen Komponenten weiter bestehen sollen. Dies wird visuell besonders hart, wenn ein ganzes Notebook durch eine andere mehrseitige Komponente ausgetauscht wird und alle Komponenten von den Notebook-Seiten auf Seiten der neuen Komponente wechseln sollen.
- Austausch der Basisklasse des Formulars für die Formularvererbung,
- Einfügen und Löschen von Komponenten aus der Eltern-Kind-Hierarchie (z.B. Einfügen einer neuen Panel-Komponente als *Parent* für eine Ansammlung anderer Komponenten oder Entfernen eines nicht benötigten Panels),
- Suchen und Ersetzen von Property-Werten.

Für solche Aufgaben gibt es in Delphi die Textdarstellung eines Formulars, die ja in den letzten Abschnitten bereits angesprochen wurde. Zum Austausch der Basisklasse bei der Formularvererbung finden Sie in Kapitel 3.7.1 eine detaillierte Anleitung. An dieser Stelle geht es um den erstgenannten Punkt, also quasi um das Ändern der Klasse einer Komponente.

Austausch der Komponentenklassen *R7*

Als Beispiel soll ein mehrseitiger Dialog im *TTabbedNotebook*-Stil in einen mit *TPageControl* arbeitenden Dialog umgewandelt werden (vorstellbar wäre dieser Fall z.B. beim Portieren einer Delphi-1-Anwendung). Der visuelle Weg kommt hier kaum in Frage, denn versuchen Sie einmal, alle im Notizbuch befindlichen Komponenten im Entwurfsformular auszutauschen, ohne alle Elemente auf den einzelnen Seiten mühsam vom alten in das neue Mehrseiten-Steuerelement zu verschieben.

Um das Formular als Textdatei editieren zu können, wählen Sie in seinem lokalen Menü den Punkt ANSICHT ALS TEXT. (Seit Delphi 5 können Sie die DFM-Datei auch in einem beliebigen Texteditor bearbeiten, wenn Sie in Delphi die Option zur Speicherung im Textformat gewählt haben[10].

Das Formular des Beispielprogramms *NBDemo* in der Version, die die *TNotebook*-Komponente verwendet, sieht wie folgt aus (der Übersicht halber wurden mehrere Seiten von Property- und Unterkomponentenlisten weggelassen):

```
object Form1: TForm1
  Left = 202
  Top = 118
  BorderStyle = bsDialog
  Caption = 'Notebook-Demo: Seitenauswahl über...'
  ...
  object TabbedNotebook1: TTabbedNotebook
    Left = 0
    Top = 0
    Width = 419
    Height = 269
    Align = alClient
    TabFont.Charset = DEFAULT_CHARSET
    ...
    object TTabPage
      ...
      Caption = '... Tabs'
      object Bevel2: TBevel
        ...
```

Es genügen zwei kleine Änderungen, um das TabbedNotebook zu einem PageControl-Notizbuch zu machen:

- Ändern Sie die Typenangabe *TTabbedNotebook* in *TPageControl*.
- Ersetzen Sie mit der Suchen&Ersetzen-Funktion des Editors alle Vorkommnisse von *TTabPage* durch *TTabSheet*. Die Komponentenklasse *TTabPage* stellt die Seiten der *TTabbedNotebook*-Komponente dar und ist zur Entwurfszeit nicht ansprechbar.

10 Lokales Menü des Formulars: TEXT-DFM bzw. NEUE FORMULARE ALS TEXT in den Umgebungsoptionen unter PRÄFERENZEN.

Die Änderungen sind im folgenden Listing kursiv hervorgehoben, wobei die Zeile *object TTabSheet* in der vollständigen Datei natürlich mehrere Male vorkommt – einmal pro Seite.

```
object Form1: TForm1
  ...
  object TabbedNotebook1: TPageControl
    Left = 0
    Top = 0
    ...
    object TTabSheet
      ...
      Caption = '... Tabs'
      object Bevel2: TBevel
        ...
```

Sie können natürlich nur dann den Typ einer Komponente einfach so ändern, wenn der neue Typ alle Properties besitzt, die schon im alten vorhanden waren. Beispielsweise verfügen sowohl *TTabPage* als auch *TTabSheet* über das im obigen Listing erwähnte *Caption*-Property. Allerdings besitzt *TTabbedNotebook* einige Properties, die *TPageControl* nicht kennt, so etwa das im ersten Listing zu sehende Property *TabFont*. Wenn also nur wie vorgeschlagen die beiden Klassen *TTabbedNotebook* und *TTabSheet* ersetzt werden, erhalten wir zunächst eine unkorrekte Formulardefinition.

Trotzdem können wir in diesem Beispiel sofort ANSICHT ALS FORMULAR aus dem lokalen Menü des Editors wählen, um das Formular, dessen »Fundament« gerade ausgetauscht wurde, anzusehen. Delphi stellt nun fest, dass die Formulardatei einige nicht vorhandene Properties enthält (Abbildung 3.8), weigert sich jedoch nicht, das Formular zu laden, sondern bietet Ihnen an, die nicht vorhandenen Properties zu ignorieren. Konkret sind das die *TTabbedNotebook*-Properties *PageIndex*, *TabsPerRow*, *TabFont.Color*, *TabFont.Height*, *TabFont.Name* und *TabFont.Style*. *Ignorieren* bedeutet, dass die Werte dieser Properties einfach verloren gehen.

Manchmal verfügen Komponenten zwar über die gleichen Einstellmöglichkeiten, bieten diese aber in verschiedenen Properties an: So wird beispielsweise die Registerbeschriftung des PageControls nicht in einem *TabFont*-Property, sondern über das Property *Font* eingestellt. (Der Verlust des *TabFont*-Properties hätte in diesem Fall vermieden werden können, indem wir schon in der Textdarstellung *TabFont* durch *Font* ersetzt hätten).

Wenn Sie mehrmals den Schalter IGNORIEREN oder einmal den Schalter ALLE IGNORIEREN drücken, nimmt Delphi alle noch notwendigen Änderungen automatisch vor. Falls dadurch wichtige Properties verloren gehen, können Sie dies später im Objektinspektor korrigieren. So wollen Sie vielleicht auch den *TTabSheet*-Komponenten, deren *TTabPage*-Vorläufer noch unbenannt gewesen sind, Namen geben, um sie im Programm direkt ansprechen zu können.

Abbildung 3.8: Eine segensreiche Automatik von Delphi hat sich hier als Fehlermeldung getarnt.

Auf der CD finden Sie zwei Versionen von *NBDemo*: *NBDemo1* verwendet *TTabbedNotebook*, *NBDemo2* benutzt *TPageControl*. Der Quelltext beider Versionen ist identisch, die endgültige Version des Beispielprogramms wurde mit *TPageControl* unter 32-Bit-Delphi entwickelt. Anschließend wurde die Formulardatei wie beschrieben im Texteditor geändert (allerdings in umgekehrter Richtung: *TPageControl* wurde ersetzt durch *TTabbedNotebook*) und in das ansonsten identische Projekt *NBDemo1* übernommen.

Hinweis: Auf ähnliche Weise können Sie natürlich auch zwischen *TPageControls* und *TNotebook* wechseln. Statt *TTabPage* müssen Sie hier nur *TPage* durch *TTabSheet* ersetzen.

3.5.7 Verwenden der Standarddialoge

Da viele Routineaufgaben wie das Auswählen von Dateien, Schriften, Farben und Druckern in sehr vielen Anwendungen vorkommen, verfügt Windows über einige Standarddialoge, die jede Anwendung benutzen kann und die damit zu einer Vereinheitlichung führen können (Abbildung 3.9). In Delphi sind diese Dialoge zu leicht handhabbaren Komponenten geworden. Im Gegensatz zu den Steuerelementen sehen Sie diese Dialoge zur Entwurfszeit jedoch nicht so, wie sie zur Laufzeit auftreten. Lediglich ein kleines Icon weist zur Entwurfszeit auf das Vorhandensein eines Standarddialogs hin.

Standarddialog-Übersicht

In Delphi sind die Klassen der Standarddialoge von der Klasse *TCommonDialog* abgeleitet: *TColorDialog*, *TFontDialog*, *TPrinterSetupDialog*, *TPrintDialog*, *TFindDialog/TReplaceDialog*, *TOpenDialog/TSaveDialog*. Bei den beiden zuletzt genannten Klassenpaaren ist die erste jeweils Basisklasse für die zweite, die sich aber kaum von der ersten unterscheidet. Außerdem gibt es noch die beiden speziellen *TOpen/Save*-Dialoge *TOpenPictureDialog* und *TSavePictureDialog*.

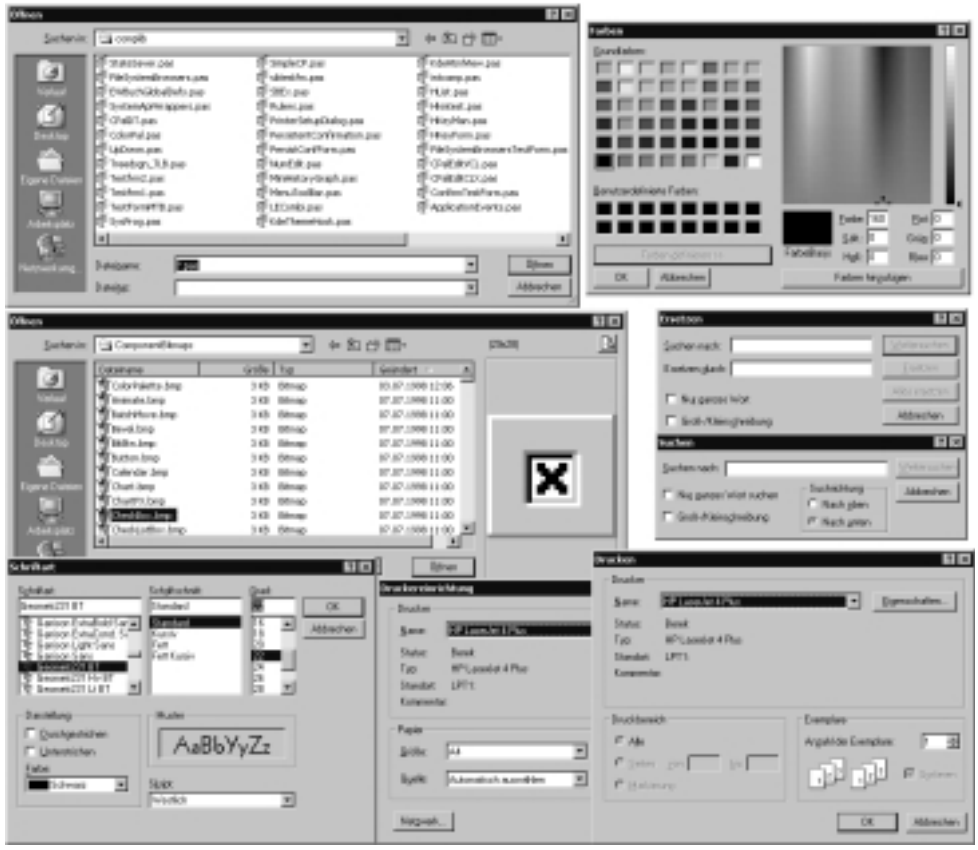


Abbildung 3.9: Die Standarddialoge von Delphi (ohne die Varianten zum Speichern von Dateien); in Delphi 6 wird nun auch die ab Windows 2000 implementierte Ordner-Liste auf der linken Seite der Dateiwahldialoge aktiviert.

Grundsätzliche Verwendung

Da die Dialog-Komponente, die zur Entwurfszeit immerhin noch als Icon dargestellt wird, zur Laufzeit überhaupt nicht mehr sichtbar ist, müssen Sie also ein weiteres Steuerelement wie z.B. einen Schalter oder einen Menüpunkt vorsehen, über den der Benutzer den Dialog aufrufen kann. In der Methode, die das von diesem Element ausgelöste Befehlsereignis (meistens *OnClick*) bearbeitet, können Sie den Code schreiben, der den Dialog aufruft. Das Minimum besteht hier in der reinen Ausführung des Dialogs, für die Sie wieder einen der typischen Einzeiler benötigen:

```
procedure TForm1.Dateispeichernunter1Click(Sender: TObject);
begin
  SaveDialog1.Execute;
end;
```

Die *Execute*-Methode initialisiert den Standarddialog nach den Werten der Properties, die Sie im Objektinspektor angegeben haben, und zeigt den Dialog an. Wenn der Benutzer ihn mit einem bestätigenden Schalter schließt, speichert *Execute* die Eingaben des Benutzers wieder in den Properties, bevor sie zu Ihrem Programm zurückkehrt.

Meistens gesellen sich zum Aufruf von *Execute* weitere Anweisungen: Vor dem Aufruf von *Execute* können Sie noch weitere Properties der Dialog-Komponente setzen, und nach der Ausführung werden Sie womöglich daran interessiert sein, was der Benutzer eingegeben hat. Außerdem müssen Sie unterscheiden, ob der Benutzer den Dialog per *OK*-Schalter oder mit einem Abbruch beendet hat (das Schließen über das Systemmenü zählt als Abbruch). Dies erfahren Sie vom booleschen Ergebnis, das *Execute* zurückliefert. Der erweiterte Ablauf eines Standarddialog-Aufrufs sieht damit wie folgt aus:

```
procedure OnDialogAufruf(...);
begin
  { optionale Initialisierung von Dialog-Properties }
  ...
  if StandardDialog.Execute = True then begin
     { Ergebnisverwertung, z.B. }
     LoadFile(StandardDialog.FileName);
  end;
  { else: Benutzer hat den Dialog abgebrochen }
end;
```

Eine wichtige Frage muss noch beantwortet werden: Angenommen, der Benutzer hat nun eine Einstellung vorgenommen, die sich katastrophal auf sein Computersystem auswirken würde, und bricht den Dialog mit *Abbruch* ab. Ihr Programm fragt aber nicht das Ergebnis von *Execute* ab und tut so, als hätte der Benutzer *OK* gedrückt. Was passiert dann?

Auch hier sorgen die Komponenten von Delphi für mehr Sicherheit, denn die Properties der Standarddialoge sind völlig von den Werten des tatsächlichen Dialogs getrennt: Wenn der Benutzer *Abbruch* drückt, liest die Dialogkomponente die Eingaben gar nicht aus dem Dialog aus, so dass sich die Werte der Properties durch den *Execute*-Aufruf nicht verändern. Waren also vor dem Dialogaufruf sinnvolle Werte in den Properties, so sind diese auch danach noch vorhanden; es passiert also nichts Schlimmes, um die eben gestellte Frage zu beantworten.

Hinweis: Standarddialoge belegen nur dann Windows-Ressourcen, wenn der Dialog aktiv ist. Der Aufruf eines Dialogs mit *Execute* entspricht also einer dynamischen Konstruktion eines Formulars, dem Aufruf von *ShowModal* und der anschließenden Freigabe des Formulars mit *Free*.

Dateiauswahl-Dialoge

Bei den Dialogen zur Dateiauswahl gibt es unterschiedliche Dialoge zum Speichern und zum Laden von Dateien. Neben minimalen Unterschieden im Äußeren liegen die Differenzen hauptsächlich in den Überprüfungen, die die Dialoge automatisch durchführen können. Wenn Sie z.B. eine nicht existierende Datei eingeben, kann der *Datei-öffnen*-Dialog fragen, ob eine neue Datei erstellt werden soll. Der *Datei-speichern-unter*-Dialog kann Sie zur Sicherheit fragen, ob die Datei überschrieben werden soll, wenn Sie eine schon existierende Datei angeben.

Trotz dieser Unterschiede sind jedoch die Properties der Komponentenklassen *TOpenDialog* und *TSaveDialog* absolut identisch, je nach Typ haben allerdings einige Optionen im Property *Options* keine Wirkung.

Property	Typ	Bedeutung
DefaultExt	String	Endung, die nach Abschluss des Dialogs automatisch an *FileName* angehängt wird, wenn der Benutzer keine Endung angegeben hat.
FileName	String	Pfad und Name der Datei, wird zur Initialisierung des Dialogs und zur Rückgabe des Werts verwendet.
Filter	String	Dieser String enthält die vorgegebenen Filter des Dialogs als eine Folge von sich abwechselnden Filternamen und Dateimasken. Drücken Sie zur Entwurfszeit auf den Schalter neben dem Property-Wert, um einen komfortablen Editor für diese Liste zu erhalten.
FilterIndex	Integer	Index in das *Filter*-Property, der den beim Öffnen des Dialogs verwendeten Filter angibt
Files	TStrings	Liste der ausgewählten Dateien (siehe Option *ofAllowMultiSel*)
InitialDir	String	Hier können Sie ein Verzeichnis angeben, das die Dialogbox als Erstes anzeigt bzw. in das sie wechselt (siehe auch Option *ofNoChangeDir*).
Options	TOpenOptions	siehe unten unter *Ausgewählte Optionen*
OptionsEx	TOpenOptionsEx	neu in Delphi 6, enthält bisher nur die Option *ofExNoPlacesBar*, mit der Sie das Feld mit den großen Ordnersymbolen am linken Rand des Dialogs (siehe Abbildung) abschalten können.
Title	String	Titel des Dialogfensters, der den Standardtitel ersetzen soll

Ausgewählte Optionen

Die vielleicht beliebtesten Optionen, die Sie als Menge im Property *Options* angeben können, sind:

- *ofPathMustExist* ist in den meisten Fällen empfehlenswert: Falls der Benutzer einen neuen Dateinamen angibt, so muss dieser wenigstens einen bestehenden Pfad beinhalten.

- *ofFileMustExist* empfiehlt sich beim Öffnen von Dateien, denn es erlaubt nur die Auswahl bestehender Dateien. Verzichten Sie auf diese Option und wollen Sie notfalls eine neue Datei erstellen, so bietet sich eine automatische Sicherheitsabfrage über die Option *ofCreatePrompt* an (diese Option ist nur beim Öffnen-Dialog gültig, allerdings nicht portabel, da in der CLX nicht enthalten).
- *ofOverwritePrompt* ist normalerweise für den Speichern-Dialog unbedingt zu empfehlen: Falls die ausgewählte Datei existiert, macht der Dialog eine Sicherheitsabfrage. Wenn Sie diese Option verwenden und das Ergebnis der *Execute*-Methode *True* ist, können Sie davon ausgehen, dass der Anwender sich entweder zustimmend zu dieser Sicherheitsabfrage geäußert oder eine neue Datei angegeben hat (diese Option ist nur beim Speichern-Dialog gültig).

Die zahlreichen weiteren Optionen betreffen unter anderem die Mehrfachauswahl, den Umgang mit schreibgeschützten Dateien, die Eingabe unzulässiger Zeichen, den Verzeichniswechsel und die Rückgabe von Verknüpfungen in LNK-Dateien. Eine komplette Auflistung dieser Optionen unterbleibt aus Platzgründen.

Beispiel für Dateidialog-Einstellungen

Bei den Dateiauswahl-Dialogen des TreeDesigners (Kapitel 5) wurden nur jeweils drei Properties geändert. So sieht beispielsweise der *Datei-öffnen*-Dialog in der Formulardatei wie folgt aus:

```
object OpenDialog1: TOpenDialog
  DefaultExt = 'TVF'
  Filter = 'Tree Viewer Files|*.TVF'
  Options = [ofPathMustExist, ofFileMustExist]
  Left = 152
  Top = 141
end
```

Die *Filter*-Einstellung sorgt (in Zusammenhang mit dem voreingestellten *FilterIndex* von »1«) dafür, dass der Dialog gleich die Dateien *.TVF anzeigt. *DefaultExt* stellt sicher, dass der Dialog die Endung automatisch anhängt, wenn der Benutzer vergessen sollte, sie einzugeben. Auch die beiden in den Optionen angegebenen Flags ersparen dem TreeDesigner einige Überprüfungen. Im *Speichern-unter*-Dialog wurden diese beiden Flags durch die Option *ofOverwritePrompt* ersetzt, ansonsten stimmen die wesentlichen Properties beider Dialoge überein.

Dialoge zur Bildauswahl

Wenn Ihr Dialog dazu dienen soll, eine Bilddatei zu wählen, deren Format auch Delphis *TPicture*-Komponente erkennen kann, können Sie statt *TOpenDialog* und *TSaveDialog* einen der Dialoge *TOpenPictureDialog* und *TSavePictureDialog* verwenden. Diese Dialoge weisen einen zusätzlichen Vorschaubereich auf, in dem der Inhalt der gerade

ausgewählten Datei automatisch angezeigt wird. Die Verwendung dieser Dialoge stimmt mit der der normalen Dialoge völlig überein. Wenn Sie die Unit *JPEG* in Ihr Projekt einbinden, funktioniert *TOpenPictureDialog* auch mit Dateien im JPEG-Format.

Der Farbauswahl-Dialog

Aufgabe des *TColorDialogs* ist es, einen Wert des Typs *TColor* zum Editieren bereitzustellen. Sie können die anfangs angezeigte Farbe im Property *Color* bestimmen, aus dem Sie auch die Ergebnisfarbe auslesen können. Eine Liste von vordefinierten Farben, die der Farbdialog im Feld *Selbst definierte Farben* anzeigt, können Sie als String im Property *CustomColors* angeben. Ein Beispiel zum Format dieses Strings finden Sie im Zusammenhang mit der Farbpaletten-Komponente in Kapitel 6.6.3.

Die Optionen sind diesmal eher dünn gesät, zwei der drei in allen Delphi-Versionen möglichen Flags verdanken ihr Vorhandensein der Tatsache, dass der Windows-Dialog aufklappbar ist, wobei der aufklappbare Teil eine Werkstatt zum Definieren eigener Farben enthält (siehe Abbildung 3.9).

Option	**Bedeutung**
cdFullOpen	bewirkt, dass der Dialog schon nach dem Öffnen im aufgeklappten Zustand angezeigt wird.
cdPreventFullOpen	nimmt dem Benutzer die Möglichkeit, den Dialog aufzuklappen, indem der entsprechende Schalter deaktiviert wird.
cdShowHelp	bewirkt im Zustand *True*, dass ein Hilfeschalter im Dialog angezeigt wird.

Mit zwei weiteren Optionen können Sie beeinflussen, welche Farben erlaubt sind. Diese Optionen wirken sich im Echtfarbmodus nicht aus, da in diesem immer alle Farben zugelassen sind. Bei 16 oder 256 Farben können Sie jedoch durch Einschalten der Option *cdSolidColors* bewirken, dass der Dialog nur die Farben zurückliefert, die im aktuellen Grafikmodus ohne Mischverfahren (Dithering) dargestellt werden können.

Der Dialog zur Schriftauswahl

Im Schriftauswahl-Dialog dreht sich alles um das Property *Font*, das den Typ *TFont* besitzt. Sie bedienen dieses Property wie die Properties *Color* und *FileName* der bisher gesehenen Dialoge. Die Vielzahl der unter Windows verwendeten Schrifttypen beschert dem *TFontDialog* eine reichhaltige Auswahl an Optionen, mit denen Sie die Arten der anzuzeigenden Schriften wählen. Diese und andere Optionen finden Sie im Property *Options*. Ob der Dialog Schriften für den Drucker, den Bildschirm oder beide anzeigen soll, bestimmen Sie im Property *Device*. Schließlich können Sie in den Properties *MaxFontSize* und *MinFontSize* einen Bereich der Schriftgrößen eingeben, den der Benutzer einhalten soll (hier benötigen Sie auch die Option *fdLimitSize*).

In vielen Fällen verrichtet der Schriftauswahl-Dialog mit den voreingestellten Properties gute Dienste, so z.B. auch im ersten Beispielprogramm aus Kapitel 1, daher sei für eine detaillierte Beschreibung der Properties und Optionen auf die Online-Hilfe verwiesen.

Dialoge für das Drucken

Das Zusammenspiel der beiden Standarddialoge für das Drucken läuft so ab wie in den meisten Windows-Anwendungen, in denen es die Menüpunkte DRUCKEN... und DRUCKEREINRICHTUNG... gibt. Der erste Menüpunkt führt zu einem Dialog, in dem Sie einstellen können, was zu drucken ist (siehe auch Abbildung 3.9), der zweite Menüpunkt ruft einen Dialog zur Einstellung des Druckertreibers auf. Mit einem Schalter können Sie außerdem vom *Drucken*-Dialog aus den Einstellungs-Dialog aufrufen, weshalb ein eigener Menüpunkt DRUCKEREINRICHTUNG nicht unbedingt erforderlich ist (die Delphi-IDE verzichtet beispielsweise darauf, verwendet aber auch nicht den Standarddialog zum Drucken).

Die VCL stellt die beiden Dialoge in den Klassen *TPrinterSetupDialog* für die allgemeine Druckereinrichtung und *TPrintDialog* für die Optionen des speziellen Ausdrucks zur Verfügung. *TPrinterSetupDialog* enthält keine neuen öffentlichen Properties oder Methoden und lässt sich daher in der Ausführung nicht beeinflussen. Der *Drucken*-Dialog enthält jedoch einige Felder, die *TPrintDialog* dementsprechend als Properties zur Verfügung stellt:

Property	**Beschreibung**
Collate	entspricht dem Markierungsfeld *Kopien sortieren*.
Copies	Anzahl der Kopien der gesamten Druckausgabe.
FromPage	Seite, mit der der Druck beginnen soll.
PrintToFile	entspricht dem Zustand des Markierungsschalters *Ausdruck in Datei*, der nur bei eingeschalteter Option *poPrintToFile* angezeigt wird.
PrintRange	*prAllPages*, wenn alle Seiten gedruckt werden sollen. Bei *prSelection* soll nur der markierte Bereich und bei *prPageNums* der Bereich *FromPage* bis *ToPage* gedruckt werden.
ToPage	letzte Seite, die gedruckt werden soll
MinPage MaxPage	Diese Properties geben einen vorgegebenen Bereich an, in den die Eingaben *FromPage* und *ToPage* des Benutzers fallen sollen.
Options	Hier kontrollieren Sie, welche Elemente des Dialogs angezeigt bzw. deaktiviert werden (*poPrintToFile*, *poHelp*, *poDisablePrintToFile*), welche der Radioschalter *Markierung* und *Seiten* aktiviert werden (*poPageNums*, *poSelection*) und ob der Dialog den Benutzer warnen soll, wenn kein Drucker installiert ist (*poWarning*).

Während jedoch der Druckereinrichtungs-Dialog etwas Handfestes tut, indem er einen Druckertreiber auswählt und dessen Optionen einstellt, stellt Ihnen der Drucken-Dialog lediglich eine unverbindliche Eingabemaske zur Verfügung. Damit die Eingaben des Benutzers auch im Ausdruck berücksichtigt werden, müssen Sie diese aus der Dialogkomponente auslesen und in Ihrer Druckfunktion verwenden (siehe z.B. Kapitel 5.6.4). Für die Berücksichtigung der Optionen im Einrichtungsdialog ist automatisch gesorgt.

Dialoge für die Textsuche

Aus technischer Sicht sind die beiden Dialoge zum Suchen bzw. zum Suchen und Ersetzen interessant, denn *TReplaceDialog* ist von *TFindDialog* abgeleitet. Der *Ersetzen*-Dialog ist eine Erweiterung des *Suchen*-Dialogs, so dass die Klasse *TReplaceDialog* einiges von *TFindDialog* erben kann.

Ein weiter Aspekt, der bei keinem anderen Standarddialog zu finden ist, ist die Ereignissteuerung: Beide Dialoge können Ereignisse erzeugen, zu denen Sie Bearbeitungsmethoden schreiben müssen, wenn Sie das übliche Verhalten der Dialoge sicherstellen wollen: Drückt der Benutzer den Schalter *Suchen* oder *Ersetzen*, so soll jeweils das nächste Vorkommnis des Suchtextes gesucht oder ersetzt werden. Den Code zum Suchen müssen Sie selbst zur Verfügung stellen, und zwar müssen Sie ihn mit den *OnFind*- bzw. *OnReplace*-Ereignissen verknüpfen. Diese Ereignisse werden aufgerufen, während der Dialog läuft, also noch bevor der Aufruf der *Execute*-Methode beendet ist.

Welchen Text der Benutzer suchen und ersetzen will, erfahren Sie aus den Dialog-Properties *FindText* und *ReplaceText*; die Optionen, die er eingestellt hat (Groß- und Kleinschreibung, ganze Wörter etc.), befinden sich wieder im Property *Options*, das noch viele weitere Elemente enthält, die alle von Ihrer Suchmethode berücksichtigt werden könnten und die hier aufgrund ihrer selbst erklärenden Namen nur kurz zur Übersicht aufgelistet werden sollen:

```
TFindOption = (frDown, frFindNext, frHideMatchCase, frHideWholeWord,
    frHideUpDown, frMatchCase, frDisableMatchCase, frDisableUpDown,
    frDisableWholeWord, frReplace, frReplaceAll, frWholeWord,
    frShowHelp);
```

3.6 Komplexere Steuerelemente

Dieses Kapitel behandelt in Ergänzung zu den Kapiteln 1.9 (Überblick und Vorstellung einfacher Komponenten) und 3.3 (Gemeinsamkeiten aller Komponenten) die folgenden Themen:

- die Editierkomponenten *TEdit*, *TMemo* und *TRichEdit* (Kapitel 3.6.1)

- die Darstellung einer Menge von Objekten in tabellarischen und mit Icons versehenen Ansichten mit Hilfe von *TListView* in Kapitel 3.6.2 (in diesem Zusammenhang wird auch die allgemein verwendbare Bilderlisten-Komponente *TImageList* vorgestellt)
- die Anzeige hierarchischer Strukturen mit *TTreeView* (Kapitel 3.6.3).

3.6.1 Editierfelder, Memos und RTF-Felder

Auf der *Standard*-Seite der Komponentenpalette finden Sie zwei verschiedene Arten von Editierfeldern: einzeilige *Edit*-Komponenten und mehrzeilige *Memos*. Die bei beiden vorhandenen Eigenschaften werden in der gemeinsamen Basisklasse *TCustomEdit* definiert. Alle Eigenschaften, die im Folgenden für die Klasse *TEdit* besprochen werden, stammen aus *TCustomEdit* und gelten folglich sowohl für *TEdit* als auch für *TMemo*. *TMemo* enthält einige weitere Properties, die in einem eigenen Abschnitt zusammengefasst werden. Auch die Komponente *TRichEdit* stammt von *TCustomEdit* ab; ihr ist das Ende dieses Kapitels gewidmet.

Textinhalt

Das Wichtigste an einem Editierfeld jeglicher Art ist sicher der Text selbst. Über das von *TControl* geerbte und überschriebene Property *Text* erhalten Sie Zugriff auf diesen Text; Memofelder geben Ihnen über das Property *Lines* außerdem Zugriff auf einzelne Zeilen.

Markierter Text

Die vielleicht zweitwichtigste Information, die Sie einer jeden Editierkomponente entnehmen können, ist, welcher Text selektiert ist. Dabei kann grundsätzlich nur ein durchgängiger Bereich des Textes markiert sein, der ab einem bestimmten Zeichen beginnt und eine bestimmte Länge hat. Der Index des ersten markierten Zeichens steht dabei in dem Property *SelStart*, die Zahl der markierten Zeichen finden Sie im Property *SelLength*. Das letzte markierte Zeichen hat also den Index *SelStart+SelLength-1*.

Wenn Sie jedoch den selektierten Text auslesen wollen, brauchen Sie nicht mit diesen Indizes im Textpuffer der Komponente herumzusuchen, denn das Property *SelText* gibt direkten Zugriff auf den ausgewählten Text. Die wichtigen *TEdit*-Elemente für den selektierten Text zusammengefasst sind:

Elementart	Name	Typ bzw. Parameter	Bedeutung
property	SelStart	Integer	Index des ersten markierten Zeichens; falls nichts markiert ist: Index der Cursorposition
property	SelLength	Integer	Zahl der markierten Zeichen.

Elementart	Name	Typ bzw. Parameter	Bedeutung
property	SelText	String	enthält den markierten Text.
procedure	SelectAll	–	selektiert den gesamten Text.

Eine Besonderheit von *SelStart* ist, dass dieses Property auch dann sinnvolle Werte enthält, wenn kein Text markiert ist, wenn also *SelLength* Null ist. *SelStart* enthält in diesem Fall den Index des Zeichens, das sich gerade an der Position der Eingabemarkierung befindet.

Editierfeld-Properties

Weitere von *TEdit* deklarierte Properties sind:

Property	Typ	Bedeutung
AutoSelect	Boolean	Wenn *True*, wird der gesamte Text selektiert, sobald der Benutzer mit der Tabulatortaste zu diesem Editierfeld springt (so, dass er den Text sofort überschreiben kann).
AutoSize	Boolean	automatische Größenanpassung des Editierfelds an die Schriftgröße (und nicht an den Text selbst, wie bei *TLabel*)
CharCase	(ecNormal, ecUpperCase, ecLowerCase)	bestimmt, ob die Zeichen automatisch in Groß- oder Kleinbuchstaben umgewandelt werden.
HideSelection	Boolean	Wenn *True*, wird die Markierung des Textes nur angezeigt, wenn das Editierfeld den Tastaturfokus hat.
MaxLength	Integer	gibt die maximale Länge des Textes an; bei Null gilt eine systembedingte Längenbeschränkung.
Modified	Boolean	*True*, wenn Text geändert wurde, nachdem die Methode *ClearModify* das letzte Mal aufgerufen wurde.
PasswordChar	Char	Solange dieses Property das Nullzeichen (»#0«) enthält, funktioniert das Editierfeld normal. Geben Sie hier ein anderes Zeichen an, zeigt das Editierfeld für jedes eingegebene Zeichen nur dieses *PasswordChar* an, speichert aber in *Text* den richtigen Text, z.B. ein Passwort.
ReadOnly	Boolean	verhindert Veränderungen am Text, wenn *True*

Bei einzeiligen Editierfeldern der Klasse *TEdit* benötigen Sie häufig nur das Property *Text*, das den Feldinhalt speichert, und manchmal das Ereignis *OnChange*, um sofort auf jede Änderung reagieren zu können.

Bearbeitungs- und Zwischenablage-Methoden

Ein weiterer Funktionsbereich von *TEdit* beschäftigt sich mit der Zwischenablage und anderen Funktionen, die häufig in einem BEARBEITEN-Menü zur Verfügung stehen. Hierfür sind ausschließlich Methoden zuständig:

Methode	Bedeutung
Clear	löscht den gesamten Text.
ClearSelection	löscht den ausgewählten Text.
CopyToClipboard	kopiert den Text in die Zwischenablage und wird normalerweise in einer Methode aufgerufen, die das *OnClick*-Ereignis eines BEARBEITEN | KOPIEREN-Menüpunkts bearbeitet.
CutToClipboard	entspricht dem Menüpunkt AUSSCHNEIDEN (=*CopyToClipBoard+ClearSelection*).
PasteFromClipboard	entspricht dem Menüpunkt EINFÜGEN – fügt den Inhalt der Zwischenablage ein und markiert den eingefügten Text (bisher markierter Text wird überschrieben).

Kooperation mit einem TUpDown

Für die Eingabe von Zahlenwerten können Sie ein *Edit*-Feld auch mit einer Komponente der Klasse *TUpDown* verbinden. Ein UpDown-Element besteht aus zwei kleinen Schaltern, die zum Erhöhen und Verringern eines numerischen Wertes dienen. Diese Schalter sind meistens untereinander angeordnet, können aber auch horizontal nebeneinander liegen (Property *Orientation*).

Zum Verknüpfen der beiden Komponenten setzen Sie das *Associate*-Property des UpDown-Elements auf das Editierfeld, das UpDown wird dann spätestens zur Laufzeit rechts oder links des Editierfeldes positioniert (je nach Einstellung von *AlignButton*). Nach der Verknüpfung funktioniert das Komponenten-Duo wie folgt:

- Im UpDown-Property *Position* können Sie jederzeit den Zahlenwert des Editierfelds auslesen, auch wenn der Anwender diesen ohne Hilfe des UpDowns eingegeben hat.
- Bevor das UpDown-Element den Text des Editierfelds verändert, erzeugt es ein *OnChanging*-Ereignis, bei dem Sie die Änderung noch verhindern können (über den Variablenparameter *AllowChange* der Bearbeitungsmethode).
- Kommt es zu einer Änderung durch das UpDown-Element, so ändert sich zwar das *Position*-Property des letzteren, ein *OnChange*-Ereignis wird jedoch nur für das Editierfeld ausgelöst.
- UpDown- und Editierfeld »teilen« sich den Tastaturfokus: Wenn Sie [↑] oder [↓] im Editierfeld drücken, betätigen Sie damit die Schalter des UpDowns (falls Sie dessen Property *ArrowKeys* nicht abgeschaltet haben). Ein einzelnes UpDown-Element ist mit der Tastatur nicht ansteuerbar.

Die neuen Properties von *TUpDown* legen einen maximalen Bereich (*Min*, *Max*) und eine Schrittweite (*Increment*) fest, über das Property *Thousands* können Sie die voreingestellte Unterteilung der Zahlen in Drei-Ziffern-Blöcke abstellen.

TMemo

Während sich die Klasse *TEdit* darauf beschränkt, Properties von *TCustomEdit* zu veröffentlichen, ohne neue Funktionen einzuführen, erbt *TMemo* einige weitere Properties von der »Zwischenklasse« *TCustomMemo*. Diese Klasse ist besonders deswegen interessant, weil sie ihre Elemente auch an *TRichEdit* weitervererbt. Die folgende Tabelle fasst die neuen Properties zusammen:

Property	**Typ/Werte**	**Bedeutung**
Lines	TStrings	Alternative zum Property *Text*, die Zugriff auf einzelne Zeilen des Textes bietet.
Alignment	taLeftJustify, taRightJustify, taCenter	gibt die Ausrichtung der Zeilen innerhalb des Memos an.
ScrollBars	ssHorizontal, ssVertical, ssBoth, ssNone	gibt der Komponente eine horizontale oder vertikale Bildlaufleiste oder es schaltet beide Bildlaufleisten ab (*ssNone*).
WantReturns	Boolean	Solange dieses Property bei seiner Voreinstellung *True* bleibt, bearbeitet und »verschluckt« das Editierfeld die Eingabetaste. Bei *False* geht die Eingabetaste an den Dialog, der dann einen Klick auf den Standard-Befehlsschalter simuliert. Anstelle der Eingabetaste können Sie dann im Editierfeld die Tastenkombination Strg + ↵ verwenden.
WantTabs	Boolean	wie *WantReturns*, bezieht sich jedoch auf die Tabulatortaste. Außerdem gibt es innerhalb des Editierfelds keinen Ersatz für Tab, wenn *WantTabs* auf *False* gesetzt ist.
WordWrap	Boolean	Bei *True* nimmt das Editierfeld automatisch einen Zeilenumbruch am Ende der Zeile vor.

RTF-Editierfelder: TRichEdit

Die Komponente *TRichEdit* basiert auf dem mit Windows 95 eingeführten RTF-Editierfeld. *RTF* steht für *Rich Text Format*, ein Dateiformat, mit dem nicht nur einfacher Text, sondern auch zahlreiche Textattribute gespeichert werden können. So können Sie beispielsweise Textabschnitte fett, kursiv und unterstrichen darstellen, verschiedene Schriftarten und -größen wählen und Absatzformatierungen vornehmen. Die Besonderheit des RTF-Formats liegen darin, dass auch diese Formatierungen als normaler Text gespeichert werden. RTF-Dateien können also mit jedem Texteditor gelesen werden, der ASCII- bzw. ANSI-Dateien erkennt, sind dann aber kaum mehr lesbar, da z.B.

der Text »Dies ist ein Absatz mit einem Schriftartwechsel« in einer Unmenge von Formatanweisungen versteckt wird:

```
\deflang1031\pard\plain\f0\fs16\cf0 Dies ist ein Absatz mit einem
\plain\f3\fs34\cf0 Schriftartwechsel\plain\f0\fs16\cf0 .
```

Der Nachteil von *TRichEdit* ist, dass Sie zusätzlich zu einer Komponente dieses Typs zahlreiche weitere Komponenten benötigen, mit denen die Formatierungen veranlasst werden können, und dass Sie für diese einigen Code schreiben müssen, um das RichEdit-Feld zu steuern. Die Aufgabenteilung zwischen *TRichEdit* und Ihrem Code sieht wie folgt aus:

- *TRichEdit* verwaltet die Daten im RTF-Format. Dazu gehört, sie in Dateien zu schreiben, von dort zu lesen und am Bildschirm anzuzeigen. Neben den Textdaten speichert *TRichEdit* die aktuellen Textattribute. Neu eingegebener Text wird automatisch mit diesen Attributen versehen.
- Um die Möglichkeiten von *TRichEdit* auszunutzen, müssen Sie dieser Komponente in Ihrem Code mitteilen, welche Textattribute als aktuelles Format verwendet werden sollen. Auch wenn der Text gespeichert, geladen oder nach der Eingabe umformatiert werden soll, müssen Sie dem RichEdit-Feld die entsprechenden Anweisungen geben.

Um den gerade markierten Text zu formatieren, setzen Sie das Property *SelAttributes*; das Format für neu eingegebenen Text befindet sich entsprechend im Property *DefAttributes*. Beide Properties haben den Typ *TTextAttributes* und können somit einen Schriftnamen, eine Schriftgröße und verschiedene andere Attribute speichern, darunter sogar ein Flag, das den Textabschnitt vor Veränderungen schützt (*TTextAttributes.Protected*).

Typische Formatierungsanweisungen im Quelltext sehen damit wie folgt aus:

```
// den als Nächstes eingegebenen Text fett (fsBold) darstellen:
RichEdit.DefAttributes.Style := RichEdit.DefAttributes.Style + [fsBold];
// den gerade markierten Text kursiv (fsItalic) formatieren:
with RichEdit.SelAttributes do
  Style := Style + [fsItalic];
// feststellen, ob Text markiert ist. Wenn ja -> diesen über
// SelAttributes formatieren, wenn nicht -> das Format für neu
// eingegebenen Text setzen (DefAttributes):
if RichEdit.SelLength > 0 then // Text markiert?
  CurrText := RichEdit.SelAttributes
else CurrText := RichEdit.DefAttributes;
CurrText.Style := ... // Attribute setzen (z.B. wie oben)
```

Neben diesen vielen neuen Möglichkeiten erbt *TRichEdit* auch eine Reihe von Methoden und Properties, die Ihnen wahrscheinlich schon von *TMemo* bekannt sind, denn *TRichEdit* ist wie *TMemo* von *TCustomMemo* abgeleitet.

Wie schon angedeutet, benötigen Sie recht viel Code, um alle Möglichkeiten von *TRichEdit* für den Anwender verfügbar zu machen. Für die wichtigsten Operationen genügt jedoch bereits die Verwendung der erwähnten Properties *DefAttributes* und *SelAttributes*, der von *TCustomEdit* geerbten Methoden *CopyToClipboard*, *CutToClipboard* und *PasteFromClipboard* für die üblichen Zwischenablageoperationen sowie der *TStrings*-Methoden *SaveToFile* und *LoadToFile*, mit denen Sie den Text speichern und laden können (z.B. *RichEdit.Lines.LoadFromFile(Dateiname)*).

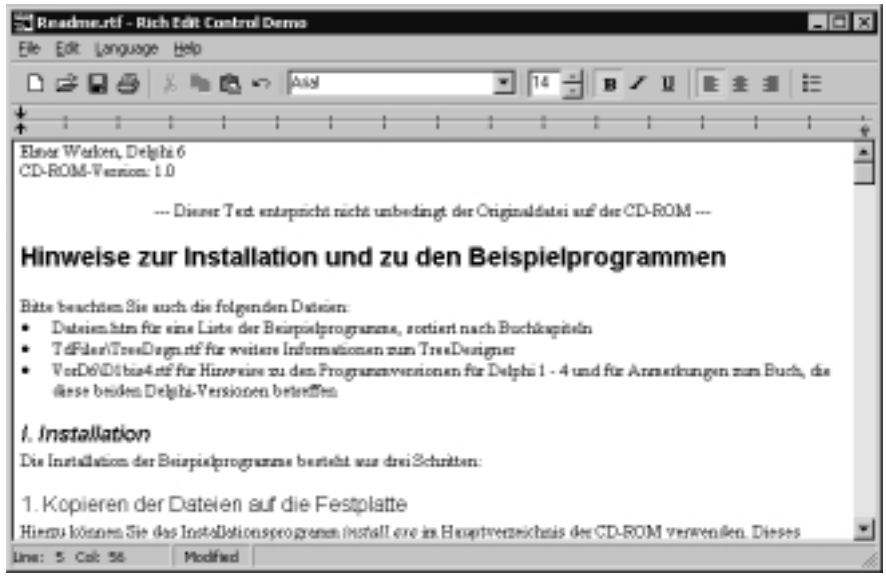


Abbildung 3.10: Ein mit Delphi mitgeliefertes Beispielprogramm liefert ein Paradebeispiel für die Benutzung der Komponente TRichEdit

Ein Beispielprogramm erübrigt sich an dieser Stelle, denn zu Delphis Lieferumfang gehört mit `Delphi-Verzeichnis\demos\richedit\richedit.bpg` bereits ein Demoprogramm, welches all diese Standardoperationen und noch einige mehr über ein Menü und eine Toolbar zur Verfügung stellt (siehe Abbildung 3.10). Es enthält sogar ein Lineal, an dem Sie die Absatzeinzüge einstellen können (interessanterweise setzt sich dieses aus einem *TPanel* und drei *TLabel*-Komponenten zusammen).

Wenn Sie also beispielsweise wissen wollen, wie Sie einen Absatz rechtsbündig darstellen können, laden Sie einfach dieses Projekt, doppelklicken zur Entwurfszeit auf den entsprechenden *TSpeedButton* der Toolbar (der zweite von rechts) und finden den gesuchten Beispiel-Code, der sich in diesem Fall des bisher noch nicht erwähnten *TRichEdit*-Properties *Paragraph* bedient.

3.6.2 Listenansichten (ListViews) und Bilderlisten

In Kapitel 1.9.5 wurde bereits anhand von *Wecker3e* der praktische Einsatz eines ListViews demonstriert. Im vorliegenden Kapitel finden Sie weitere Details zu dieser Komponente, wie etwa zu Zustandsbildern, Overlay-Bildern, Drag&Drop, Editieren der Beschriftung und zum dynamischen Aufbau der Bilderlisten. All diesen Themen dient das spezielle Beispielprogramm *DynamicListView*.

Nachdem die verschiedenen Ansichten eines ListViews und die zugehörigen Werte des *ViewStyle*-Properties (*vsIcon*, *vsSmallIcon*, *vsList* und *vsReport*) bereits in Kapitel 1.9.5 erläutert wurden, befassen wir uns als Nächstes mit den Bilderlisten.

Die Verknüpfung zwischen TListView und Bilderlisten

Ein ListView enthält selbst keine Bilder, sondern nur eine Liste von Einträgen und drei Verweise auf Bilderlisten. Solche Listen sind darauf ausgelegt, die als Bitmaps vorliegenden Bilder speicher- und zeitsparend zu verwalten.

Bilderlisten werden als eigenständige *TImageList*-Komponenten in einem Formular untergebracht. Zu jedem Eintrag des ListViews gehören verschiedene Indizes, die jeweils auf ein Bild der Bilderlisten verweisen. Sie können diese Indizes wie folgt setzen:

- zur Entwurfszeit im Editor des Properties *TListView.Items*
- zur Laufzeit über die Properties der *TListItem*-Objekte, die im Property *Items* enthalten sind (*Items* ist ein Array-Property).

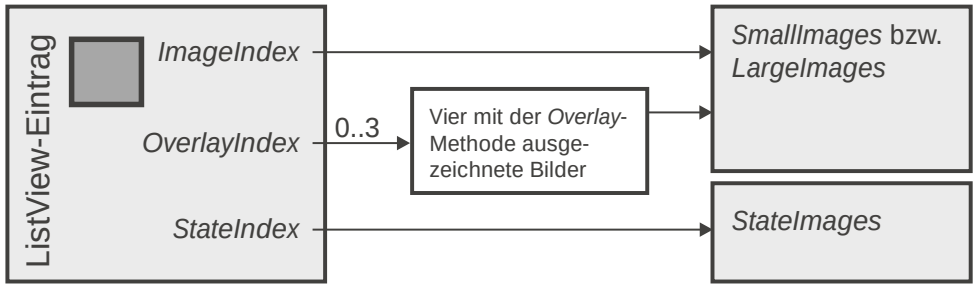


Abbildung 3.11: Die Verknüpfungen zwischen ListView-Element und Bilderlisten

Abbildung 3.11 stellt diese Zusammenhänge dar: Von den beiden Bilderlisten *SmallImages* und *LargeImages* ist immer nur eine Liste aktiv. *LargeImages* wird nur im Modus *vsIcon* verwendet, in den drei anderen Modi verwendet das ListView statt dessen die

Liste *SmallImages*. Unabhängig vom Anzeigemodus kommt als Zweites noch die Liste *StateImages* hinzu. Jeder ListView-Eintrag enthält drei Verweise auf Bilder dieser Listen:

- Das Property *ImageIndex* (ListView Einträge-Editor: Feld *Bild-Index*) bezeichnet gewissermaßen ein »Haupt-Icon« für den Eintrag, das Icon, das diesen Eintrag repräsentieren soll. Je nach Anzeigemodus stammt dieses Icon aus verschiedenen Bilderlisten (*SmallImages* oder *LargeImages*); der Index ist jedoch für beide Bilderlisten derselbe.
- *StateIndex* (ListView-Einträge-Editor: *Status-Index*) verweist auf ein Bild, das optional den Zustand des Eintrags anzeigen kann. Das *StateIndex*-Bild wird links neben dem »Haupt-Icon« dargestellt. Alle möglichen Zustände werden in einer eigenen Liste gespeichert, und zwar in *StateImages* (theoretisch können Sie *StateImages*, *LargeImages* und *SmallImages* auch auf dieselbe Liste weisen lassen). Per Voreinstellung hat *StateIndex* den Wert -1, was bedeutet, dass kein zusätzliches Zustands-Icon angezeigt wird.
- Den Index *OverlayIndex* können Sie nur zur Laufzeit ansprechen. Ein Overlay-Bild wird nicht wie das Statusbild neben dem Haupt-Icon gezeichnet, sondern von diesem überlagert. Normalerweise sind große Teile des Overlay-Bildes transparent, so dass das darunter liegende Icon noch erkennbar bleibt. Ein Beispiel für Overlay-Bilder gibt die Windows-Shell: Objekte, die eine Verknüpfung zu einem anderen Objekt darstellen, verwenden dasselbe Icon wie das Original-Objekt und überlagern dieses mit einem kleinen Pfeilsymbol.

  OverlayIndex bezieht sich zwar auf die Bilder derselben Bilderliste wie *ImageIndex*, ist jedoch kein direkter Index zu dieser Liste. Statt dessen gibt er eines von vier Bildern an, die zuvor mit der Methode *Overlay* extra als Overlay-Bild ausgezeichnet wurden.

Hinweis: Über eine »visuelle Formularverknüpfung« genannte Funktion können Sie von mehreren Formularen auf dieselbe Bilderliste zugreifen. Das Formular, das die Bilderliste enthält, muss dazu in der *uses*-Anweisung der Unit aufgenommen werden, die auf die Bilderliste zugreifen will. Der Objektinspektor zeigt dann für die *ImageList*-Properties in dieser Unit auch die Bilderlisten des anderen Formulars an. Auch die in Kapitel 7.2.1 besprochenen Datenmodule eignen sich zur Aufnahme von Bilderlisten, die in mehreren Formularen benötigt werden.

TImageList zur Entwurfszeit

Wenn Sie ein Exemplar der Klasse *TImageList* in das Formular einfügen, müssen Sie die Größe der Bilder in den Properties *Height* und *Width* einstellen, bevor Sie die Bilder hinzufügen, denn ein nachträgliches Ändern von *Height* und *Width* führt zur Löschung

aller bisher eingefügten Bilder. Eine Verwendung von verschieden großen Bildern ist nicht möglich, da die Bilderlisten zur effektiven Verwaltung großer Bilderzahlen auf Gleichförmigkeit aufbauen.

Nach der Größeneinstellung können Sie aus dem Kontextmenü der *TImageList*-Komponente den *Bildlisten-Editor* aufrufen (der Objektinspektor bietet keinen Zugang zu diesem Editor) und, wenn auch auf eine etwas umständliche Weise, einzelne BMP-Dateien in die Bilderliste laden.

Auch die Verbindung zum ListView können Sie schon zur Entwurfszeit knüpfen, indem Sie die *TListView*-Properties *LargeImages* und *SmallImages*, eventuell auch *StateImages*, auf die Bilderlisten setzen.

TListView zur Entwurfszeit

Sie können eine *TListView*-Komponente schon vor dem Programmstart dialoggesteuert mit Inhalt füllen, allerdings sind die hierzu verwendeten Dialoge sehr umständlich in der Handhabung und in der Praxis werden die ListView-Inhalte sowieso meistens erst zur Laufzeit aufgebaut. Um einen Überblick über die *TListView*-Strukturen zu gewinnen, eignet sich diese Vorgehensweise aber allemal: Rufen Sie im Objektinspektor den Editor zum Property *Items* auf. Sie erhalten ein Dialogfenster, in dem Sie zu jedem Eintrag neben dem obligatorischen Text auch schon zwei der erwähnten Bilder-Indizes eintragen können.

Der Schalter NEUER UNTEREINTRAG dient dazu, den Text festzulegen, der im Ansichtsmodus *vsReport* in den weiteren Spalten dargestellt wird. Damit die Komponente in diesem Modus überhaupt über mehrere Spalten verfügt, müssen Sie jedoch zuerst im Property *Columns* diese Spalten definieren. Der Property-Editor zu *TListView.Columns* wurde schon in Kapitel 1.9.5 verwendet, um die Spalten für die Alarmereignisse des dortigen Beispielprogramms zu definieren.

TListItems und TListItem

TListView richtet sich wie alle anderen komplexeren Komponenten der VCL nach dem Prinzip, dass die Daten, die in der Komponente dargestellt werden, in einer eigenen Klasse zusammengefasst werden. Was also die Klasse *TStrings* für das *Items*-Property der Komponente *TListBox* ist, ist die Klasse *TListItems* für *TListView.Items*.

Da *TListItems* große Ähnlichkeit zu *TStringList* aufweist, ist das *Items*-Property von *TListView* im Quelltext fast noch einfacher zu verwenden als im Property-Editor. Im Wesentlichen besteht es aus einem Array von *TListItem*-Objekten, von denen jedes einzelne einen Listeneintrag repräsentiert und unter anderem den Beschriftungstext sowie die Nummer des Bildes in den Bilderlisten enthält. Die einzelnen *TListItem*-Objekte sind über das indizierte Property *TListItems.Item* ansprechbar, das als Stan-

dard-Array-Property definiert ist, so dass Sie ein Listenelement auf zwei verschiedene Arten ansprechen können. Zum Auswählen des Elements an der Position *Index* können Sie beispielsweise schreiben:

```
  ListView.Items.Item[Index].Selected := True;
// und einfacher:
  ListView.Items[Index].Selected := True;
```

Beispielprogramm DynamicListView

Anhand des Beispielprogramms *DynamicListView* (Abbildung 3.12) wenden wir uns nun der Funktion des ListViews zur Laufzeit zu. Die wichtigste Funktion besteht normalerweise darin, die Einträge dynamisch hinzuzufügen. Da schon Kapitel 1.9 gezeigt hat, wie Sie Einträge dynamisch hinzufügen können, beginnen wir hier zuerst mit ein paar Funktionen, die zwar sehr einfach sind, aber nicht unwichtig, denn der Benutzer ist ja von den ListViews des Windows-Explorers bereits einiges gewöhnt:

- Um dem Benutzer eine Mehrfachauswahl zu ermöglichen, schalten Sie das Property *MultiSelect* auf *True.*
- Damit die Hervorhebung der ausgewählten Einträge nicht unsichtbar gemacht wird, wenn das ListView gerade nicht aktiv ist (d. h. wenn es den Tastaturfokus verliert), wurde *HideSelection* auf *False* gesetzt.
- Für Kontextmenüs stellt *ListView* wie die meisten anderen Komponenten das Property *PopupMenu* zur Verfügung. Auch das Beispielprogramm verfügt über ein solches Popup-Menü (zu weiteren Informationen über Kontextmenüs siehe Kapitel 5.2.6, Abschnitt *Optionen in Popup-Menüs*).
- Kaum der Rede wert ist auch, wie Sie den Benutzer zur Laufzeit zwischen den einzelnen Ansichten umschalten lassen können. Hier genügen vier Schalter, die über ihr *Tag*-Property unterschieden werden können, und eine Methode, die mit den *OnClick*-Ereignissen aller vier Schalter verknüpft wird (dabei wird das *Tag*-Property nach dem in Kapitel 3.5.2 ausführlich erläuterten Prinzip verwendet):

```
procedure TMainForm.SetVSIconClick(Sender: TObject);
begin
  LV.ViewStyle := TViewStyle((Sender as TToolButton).Tag);
end;
```

Editieren

Das Editieren der Eintragsbeschriftungen ist bereits standardmäßig wie im Explorer möglich (über F2 oder einfaches Anklicken einer bereits gewählten Beschriftung). Über das Property *ReadOnly* können Sie diese Funktion auch abschalten. Ist die Editierfunktion angeschaltet, erzeugt *TListView* vor jedem Editieren ein *OnEditing*-Ereignis, in dem Sie das Editieren noch unterbinden können.

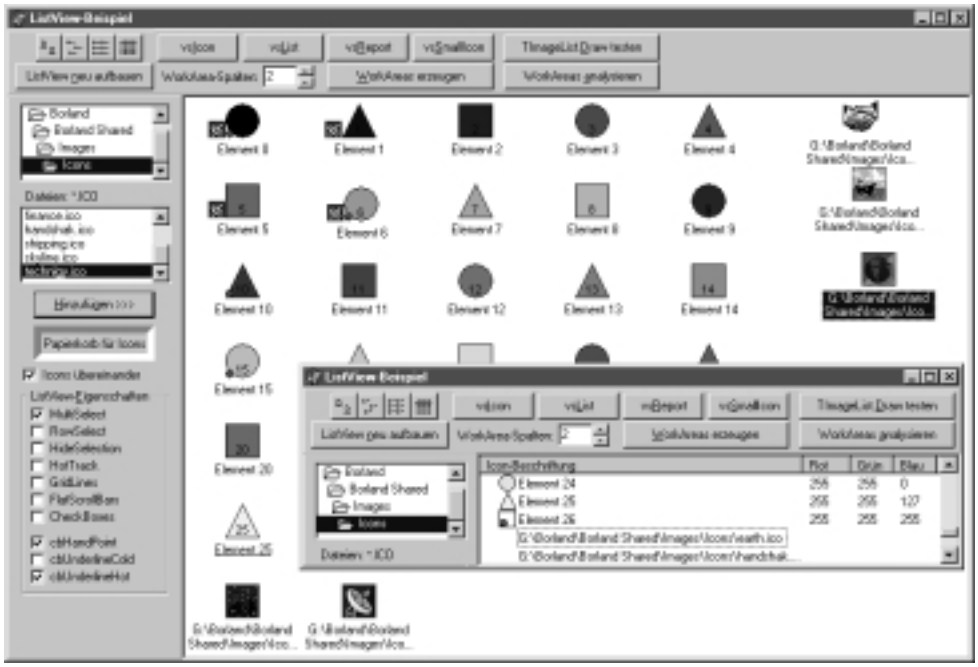


Abbildung 3.12: Dieses Programm implementiert zahlreiche ListView-Funktionen und baut den ListView-Inhalt dynamisch auf

Wenn der Benutzer die Änderung der Beschriftung ohne Abbruch abgeschlossen hat, erhalten Sie ein *OnEdited*-Ereignis, das Ihnen den editierten Eintrag und die neue Beschriftung liefert und bei dem Sie die Änderung möglicherweise an eine andere Stelle übertragen müssen. Der Windows-Explorer z.B. muss nach dem Editieren einer Dateibeschriftung die zugehörige Datei umbenennen. Im Beispielprogramm *DynamicListView* spielt es keine Rolle, ob der Benutzer die Beschriftung der Einträge ändert. Im Beispielprogramm von Kapitel 1.9.5 würde es eine Rolle spielen: Das Programm müsste dann eventuell die Daten des Alarmereignisses ändern. Da es hierfür aber einen eigenen Dialog vorsieht, verwendet es ein *ReadOnly*-ListView.

Behandeln der gewählten Einträge am Beispiel des Löschens

Im Gegensatz zum Editieren der Beschriftung können Sie sich für das Löschen von Einträgen nicht auf eine Automatik des ListViews verlassen. Das Beispielprogramm demonstriert anhand des Löschens, wie Sie alle im ListView markierten Einträge abfragen und bearbeiten. Dabei kommt das schon erwähnte *Items*-Array zum Einsatz:

```
procedure TMainForm.DeleteSelection;
var
  i: Integer;
```

```
begin
  i := 0;
  while i < ListView.Items.Count do begin // alle Einträge durchlaufen
    if ListView.Items[i].Selected then  // wenn markiert
      ListView.Items.Delete(i) // löschen
    else inc(i); // (Index nicht erhöhen, wenn Eintrag gelöscht wurde)
  end;
end;
```

Die gezeigte Methode wird im Beispielprogramm bei zwei Gelegenheiten aufgerufen: Wenn der Benutzer Einträge des ListViews per Drag&Drop in den als »Papierkorb« bezeichneten Fensterbereich zieht (dies findet im Rahmen der Drag&Drop-Funktionalität statt) und wenn er die Löschtaste drückt. Letzteres wird beim Ereignis *OnKeyDown* des ListViews abgefragt:

```
procedure TMainForm.ListViewKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key = Key_Delete then
    DeleteListViewSelection;
end;
```

Der fokussierte Eintrag

Wenn Sie in einem ListView die Mehrfachauswahl zulassen, gilt es zwischen einem selektierten und einem *fokussierten* Eintrag zu unterscheiden. Während mehrere Einträge selektiert sein können, ist immer nur ein Eintrag fokussiert, und zwar derjenige, der zuletzt angeklickt wurde. Zwar hat jeder Eintrag der *Items*-Liste auch ein *Focused*-Property; statt wie oben in einer Schleife können Sie den fokussierten Eintrag jedoch ganz schnell über das *FocusItem*-Property des ListViews herausfinden. Die folgende Anweisung beispielsweise löscht nur den fokussierten Eintrag. Auch sie benötigt einen Index als Parameter für *Delete*, den sie vom *Index*-Property des fokussierten Eintrages erfährt:

```
ListView.Items.Delete(ListView.ItemFocused.Index);
```

Verschieben der Icons in vsIcon- und vsSmallIcon-Darstellung *R55*

Obwohl die Ansichtsmodi *vsIcon* und *vsSmallIcon* theoretisch erlauben, dass Sie die Icons frei innerhalb des ListViews verschieben, ist diese Funktion nicht fest eingebaut. Zwar ist es keine funktional besonders wichtige Funktion, sie lässt sich aber sehr einfach über die Drag&Drop-Ereignisse implementieren: Es genügt, das *DragMode*-Property des ListViews auf *True* zu setzen und die ListView-Ereignisse *OnMouseDown*, *OnDragOver* und *OnDragDrop* zu bearbeiten. Da Drag&Drop bereits in Kapitel 5.8.3 ausführlich behandelt wird, sind hier nur die wesentlichen Anweisungen für das Beispielprogramm abgedruckt.

Wir benötigen hier die Methode *SetPosition* der Listeneinträge. Sie verschiebt den Eintrag an die angegebene Position (alternativ können Sie auch die Properties *Left* und *Top* des Eintrags setzen):

```
Dragged.SetPosition(Point(x-XOfs, y-YOfs));
```

Dabei sind *x* und *y* die Koordinaten der Maus, die vom *OnDragDrop*-Ereignis übergeben werden. *XOfs* und *YOfs* geben den Abstand an, den die Maus beim Anklicken des Elements zur linken oberen Ecke des Elements hatte; sie müssen beim *OnMouseDown*-Ereignis gespeichert werden. Durch das Abziehen dieser Werte von *x* und *y* wird die linke obere Ecke beim Loslassen der Maustaste nicht direkt unter die Maus platziert, sondern behält den gleichen Abstand zum Mauszeiger bei.

Während des Ziehvorgangs verwendet das Beispielprogramm eine weitere Methode des ListViews: *GetItemAt* stellt fest, welcher Eintrag sich an einer bestimmten Position befindet. So können Sie in der *OnDragOver*-Methode beispielsweise verhindern, dass ein Eintrag auf einen anderen Eintrag geschoben wird:

```
procedure TMainForm.ListViewDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  ...
  Accept := (ListView.GetItemAt(x, y) = nil);
```

Zustands-Bilder

Wie schon erwähnt, verfügt jedes *TListItem*-Objekt über ein *StateIndex*-Property, das einen Index in die Zustands-Bilderliste darstellt. Der voreingestellte Wert von -1 besagt, dass der Eintrag nicht mit einem Zustandsbild versehen wird. *DynamicListView* verwendet eine Zustands-Bilderliste mit zwei Bildern. Indem Sie auf einen Eintrag doppelklicken, schalten Sie den Eintrag jeweils in seinen nächsten Zustand:

```
procedure TMainForm.LVDblClick(Sender: TObject);
begin
  ListView.ItemFocused.StateIndex := // Wechseln zwischen -1, 0 und 1
    ListView.ItemFocused.StateIndex+2 mod 3 -1;
end;
```

Overlay-Bilder

Für die Verwendung von Overlay-Bildern ist ein wenig mehr Aufwand erforderlich, denn, wie schon erwähnt, gibt es keine eigene Bilderliste für die Overlay-Bilder. Das Beispielprogramm verwendet ein einziges Overlay-Bild, das es bei der Erzeugung des Formulars wie folgt als Overlay-Bild registriert.

```
ImageList.Overlay(Index, 0);
```

Dabei ist *Index* der Index dieses Bildes in der Bilderliste (*SmallImages* oder *LargeImages*, je nach Ansichtsmodus), der zweite Parameter von 0 besagt, dass dieses Bild in Zukunft unabhängig von seiner Position in der Bilderliste als Overlay-Bild Nummer 0 angesprochen werden kann.

Um ein Overlay-Bild mit einem Eintrag zu überlagern, setzen Sie das *OverlayIndex*-Property dieses Eintrags, das per Voreinstellung den Wert -1 hat, auf den eben im zweiten Parameter festgelegten Index. Das Beispielprogramm stellt in einem dem ListViews zugeordneten Popup-Menü einen Befehl zur Verfügung, der den fokussierten Eintrag mit dem Overlay-Bild versieht:

```
if Assigned(LV.ItemFocused) then
  LV.ItemFocused.OverlayIndex := 0;
```

Anders als die »Haupt-Bitmaps« der ListView-Einträge, die trotz ihrer Transparenz andere Bitmaps vollständig zudecken, deckt ein Overlay-Bild die überlagerte Bitmap nicht zu. Das Ergebnis des Überlagerns ist in Abbildung 3.12 bei einigen Einträgen erkennbar: Der rote Punkt bei einigen ListView-Elementen links unten stammt von einer Bitmap, die sich das Programm »selbst gemalt« hat. Die Bitmap für diesen Punkt ist so groß wie jedes andere Bild der jeweiligen Bilderliste, besteht aber größtenteils aus einer weißen Fläche, wobei die Farbe Weiß als Platzhalter für die transparente Fläche verwendet wird. Wenn Sie eine Bilderliste zur Entwurfszeit erstellen, können Sie diese »transparente Farbe« im Bilderlisteneditor für jedes Bild separat festlegen. *DynamicListView* legt diese Farbe zur Laufzeit beim Aufruf der Methode *AddMasked* fest, mit der es jede selbst gezeichnete Bitmap zur Bilderliste hinzufügt (siehe den in Kürze folgenden Abschnitt »Bilderlisten zur Laufzeit aufbauen« .

Einträge dynamisch hinzufügen

Um zur Laufzeit einen Eintrag zu einem ListView hinzuzufügen, verwenden Sie die Methode *Items.Add*, die ein neues *TListItem*-Objekt erzeugt, dieses an die Liste anhängt und einen Verweis auf dieses Objekt als Funktionsergebnis zurückliefert. Wie schon erwähnt, hat dieses Objekt die Klasse *TListItem*. Im Anschluss an den Aufruf von *Add* legen Sie die verschiedenen Daten des Eintrags fest, indem Sie die Properties des *TListItems* setzen. *DynamicListView* erzeugt beispielsweise alle Einträge in einer Schleife, wobei es für jeden Eintrag ein anderes Bild verwendet (anders ausgedrückt erzeugt es für jedes Bild der Bilderliste einen Eintrag):

```
const
  ElementCount = 27;
var
  NewItem: TListItem;
  iItem: Integer;
begin
  ListView.Items.Clear;
```

```
for iItem := 0 to ElementCount-1 do begin
  NewItem := ListView.Items.Add;
  // Eintrag beschriften mit dem Text "Eintrag [Nummer]":
  NewItem.Caption := 'Element '+(IntToStr(iItem));
  NewItem.ImageIndex := iItem; // mit Bild verknüpfen
  // die weiteren Spalten erzeugen (das Beispielprogramm
  // verwendet einen anderen Text):
  NewItem.SubItems.Add('Text für erste Spalte');
  NewItem.SubItems.Add('Text für zweite Spalte');
  ...
end;
```

Nachdem sie wie oben gezeigt mit *Add* hinzugefügt wurden, lassen sich die Textspalten des *SubItems*-Properties übrigens ebenso leicht ansprechen wie die Haupteinträge des ListViews, denn bei *SubItems* handelt es sich einfach um ein Stringlistenobjekt der weit verbreiteten *TStrings*-Klasse:

```
ListView.Items[ElementIndex].SubItems[SpaltenIndex] := 'Text';
```

Die Bilderliste des Beispielprogramms

Wie Abbildung 3.12 schon vermuten lässt, verwendet das Beispielprogramm DynamicListView keine zur Entwurfszeit festgelegte Bilderliste mit mühsam selbst gezeichneten Icons, sondern es zeichnet sich diese Icons einfach selbst. Es gestattet Ihnen zur Laufzeit außerdem, aus einer Dateiliste Icon-Dateien auszuwählen, die Sie mit dem Schalter HINZUFÜGEN in das ListView aufnehmen können. Dabei macht es sich die Dienste der Klasse *TImageList* zunutze, der wir uns als Nächstes zuwenden werden.

Hinweis: Dieses Vorgehen ist nicht zu verwechseln mit den besitzergezeichneten ListViews (*OwnerDraw*-Property = *True*), bei denen Sie ähnlich wie im Falle der besitzergezeichneten Listbox in Kapitel 4.5.4 die Einträge direkt auf den Bildschirm ausgeben, wenn Sie wollen *ohne* eine Bilderliste.

Bilderlisten zur Laufzeit aufbauen *R45*

Die Klasse *TImageList* verfügt trotz ihres recht einfachen Zwecks – der effizienten Verwaltung großer Mengen von Bildern – über eine umfangreiche Schnittstelle, von der hier nur die Methoden und Properties herausgegriffen werden sollen, die für den dynamischen Aufbau der Liste zur Laufzeit erforderlich sind.

TImageList besitzt drei mal drei Methoden, um neue Bilder zur Liste hinzuzufügen: Sie beginnen mit einem der Worte *Add...*, *Insert...* und *Replace...* Während die *Add*-Methoden das Bild immer ans Ende der Liste anhängen und als Ergebnis den Index zurückliefern, den das angehängte Bild erhalten hat, geben Sie bei den beiden anderen Arten von Methoden in einem zusätzlichen Parameter an, an welcher Position das Bild in die Liste eingefügt (*Insert...*) bzw. welches alte Bild durch das neue Bild ersetzt werden soll (*Replace...*).

Die weitere Unterscheidung der Einfügemethoden verläuft nach der Art, wie die *Maske* des Bildes angegeben wird. Diese Maske gibt an, welche Teile des Bildes transparent sind, den Hintergrund also nicht abdecken. Ein transparentes Pixel nimmt immer die Hintergrundfarbe des ListViews an, unabhängig davon, welche das ist. (Liegt unter diesem transparenten Pixel beispielsweise das Icon eines anderen Eintrags, so wird dieser mit der Hintergrundfarbe übermalt. Sie können das beim Verschieben von Icons im Beispielprogramm ausprobieren.)

Die folgende Aufzählung nennt der Einfachheit halber jeweils nur die mit »*Add*« beginnende Methode:

- *Add(Image, Mask: TBitmap)* fügt ein Bild ein, das in Form eines *TBitmap*-Objekts vorliegt (siehe Kapitel 4.5.2). Die Maske des Bildes wird in Form eines zweiten *TBitmap*-Parameters festgelegt. Masken-Bitmaps verwenden pro Pixel nur ein Bit, wobei gesetzte Bits ein transparentes Pixel ausweisen.
- *AddMasked(Image: TBitmap; MaskColor: TColor)* bewirkt, dass die Bilderliste die Maske selbst berechnet. Sie markiert in der Maske jedes Pixel als transparent, das in *Image* die in *MaskColor* angegebene Farbe aufweist.
- *AddIcon(Icon: TIcon)* übernimmt das durch den Icon-Parameter angegebene Bild und benötigt keinen Parameter für die Maske, da eine solche bereits Teil eines jeden Icons ist.

Zu den weiteren Methoden von *TImageList* gehören auch *Delete* und *Move*, mit denen Sie Bilder auch wieder löschen und verschieben können. Das Beispielprogramm macht hiervon jedoch keinen Gebrauch.

Der folgende Code-Auszug zeigt, wie *DynamicListView* die Liste der großen Bilder mit Hilfe eines einzigen *TBitmap*-Objekts erzeugt, das wiederholt an die Methode *AddMasked* übergeben wird. (Die Anweisungen, die die Grafik zeichnen, wurden dabei weggelassen, ebenso die äußere Schleife, die den gezeigten Programmabschnitt noch einmal für die Liste der kleinen Bilder wiederholt.)

```
BM := TBitmap.Create;
try
  LargeImageList.Clear;
  BM.Height := 32;    // Erzeugen der großen Bitmaps
  BM.Width := BM.Height;
  for iBitmap := 0 to ElementCount-1 do begin
    with BM.Canvas do begin
      ... Zeichnen der Bitmap, siehe CD-ROM ...
    end;
    LargeImageList.AddMasked(BM, clWhite)
  end;
finally
  BM.Free;
end;
```

Während dieser Code bei Programmstart ausgeführt wird (*OnCreate*-Ereignis des Formulars), um einen festen Satz an Bildern zu erhalten, können auch noch nachträglich – z.B. zusammen mit einem neuen Eintrag für das ListView – Bilder zu einer Bilderliste hinzugefügt werden.

Wenn Sie in der Dateiliste des Beispielprogramms eine Icon-Datei markieren und den Schalter HINZUFÜGEN drücken, lädt das Programm das gewählte Icon und zeigt es im ListView als neues Element an. Dazu muss es das Icon zunächst mit *AddIcon* zur Bilderliste hinzufügen und dann einen passenden Eintrag im ListView (mit dem Icon-Dateinamen als Beschriftung) erzeugen. Der von *AddIcon* zurückgelieferte Bildindex wird im *ImageIndex*-Property des ListView-Eintrages gespeichert, damit das Bild auch am Bildschirm sichtbar wird:

```
procedure TMainForm.InsertFile;
var
  Index: Integer;
  Item: TListItem;
  Icon: TIcon;
begin
  FileListBox.FileName;
  Icon := TIcon.Create;
  try
    Icon.LoadFromFile(FileListBox.FileName);
    Index := RTLargeImageList.AddIcon(Icon);
  finally
    Icon.Free;
  end;
  Item := LV.Items.Add; // neuen Eintrag erzeugen
  Item.Caption := FileListBox.FileName; // Beschriftung = Dateiname
  Item.ImageIndex := Index; // Eintrag mit dem Icon verbinden
end;
```

Jüngere Fähigkeiten von TListView

Die Neuerungen in der Common Control-Bibliothek erschienen nicht, wie früher bei neuen Steuerelementen üblich, im Gleichschritt mit neuen Betriebssystem-Versionen, sondern im schnellen Rhythmus der neuen Internet-Explorer-Versionen. Dies ist unter anderem der Grund dafür, dass sich in den letzten Delphi-Versionen einige Erweiterungen in der *TListView*-Komponente angesammelt haben, die in der ersten *TListView*-Version von Delphi 2 noch nicht enthalten waren.

Im Beispielprogramm können Sie am linken Rand des Hauptfensters einige noch nicht genannte boolesche Properties des ListViews an- und ausschalten:

- *RowSelect* bewirkt, dass im *vsReport*-Modus (Tabellenmodus) immer eine gesamte Zeile markiert wird (in den Anfangstagen der ListView-Steuerelemente wurde immer nur der Name des Eintrages markiert).

- *HotTrack* bedeutet, dass der Benutzer nicht klicken muss, um ein Element zu selektieren, sondern dass die Auswahl automatisch dem Mauszeiger folgt. Sollten Sie mit der Reaktionszeit dieser Verfolgung unzufrieden sein, können Sie diese auch noch im Property *HoverTime* millisekundengenau anpassen.
- *GridLines* sorgt im Modus *vsReport* für die Anzeige von Tabellenlinien zwischen den Einträgen und Spalten.
- *FlatScrollBars* schaltet den Stil der Scrollbars um.
- *CheckBoxes* bewirkt die Anzeige von Checkboxen neben jedem Eintrag. Um den Markierungsstatus eines Eintrags zu erfahren, lesen Sie das Property *TListItem.Checked*.
- Die letzten drei Optionen des Beispielprogramms entsprechen den möglichen Elementen des Mengenproperties *HotTrackStyles*. Bei eingeschaltetem *cbHandPoint* verändert sich der Mauszeiger zu einer Hand, wenn er sich über einem wählbaren Element befindet. *cbUnderlineHot* bewirkt, dass Einträge unter der Maus außerdem unterstrichen dargestellt werden (also wie ein Hyperlink in einem Web-Browser) und *cbUnderlineCold* weitet diese Unterstreichung auch auf alle anderen Elemente aus (in einer anderen Farbe).

RowSelect, *HotTrack*, *GridLines* und *CheckBoxes* sind ab Delphi 3, die *FlatScrollbars* und die drei *cb*...-Optionen ab Delphi 4 verfügbar.

Hinweis: Sind die *cb*...-Flags alle eingeschaltet, erhalten Sie ein ListView, dessen Einträge optisch die Hyperlinks eines Web-Browsers nachahmen. In diesem Fall kann das Property *HotTrack* ruhig deaktiviert werden (es würde sonst dafür sorgen, dass zusätzlich zu der Hyperlink-artigen Hervorhebung noch die herkömmliche Auswahlmarkierung angezeigt wird). Das eigene Experiment mit dem Beispielprogramm ist in diesen Fragen sicher am aufschlussreichsten. Allerdings traten im Test nach einigem Umschalten der Flags kleinere Hervorhebungsfehler auf, die eine noch unvollkommene Implementierung der ListViews vermuten lassen (entweder in Windows oder in der VCL).

Weitere Möglichkeiten

Mit den bisher besprochenen Möglichkeiten können Sie bereits flexible und leistungsfähige ListViews programmieren. Da hier nicht alle Details von *TListView*, *TImageList*, *TListItems* und *TListItem* besprochen werden können, sei zum Abschluss auf weitere Themen verwiesen, die Sie zum Teil in den Beispielprogrammen der CD demonstriert finden:

- *Zeichnen von Bildern aus Bilderlisten*: Sie können die Bilder einer Bilderliste auch unabhängig von einem ListView in die durch ein *Canvas*-Objekt angegebene Zeichenfläche zeichnen. *TImageList* stellt dafür die Methoden *Draw* und *DrawOverlay* zur Verfügung. Im Beispielprogramm *DynamicListView* rufen Sie mit dem Schalter *Draw* ein zweites Fenster auf, das mit der Bitmap des gerade fokussierten ListView-Eintrages gefüllt wird.

- *Daten mit den Einträgen verknüpfen*: Der *ShellExplorer* aus Kapitel 8.4.3 zeigt, wie Sie die Einträge von List- und TreeViews mit weiteren Daten Ihrer Anwendung verknüpfen können. Prinzipiell geht das auf dieselbe Weise, wie Sie mit jedem String einer *TStrings*-Liste einen Zeiger auf beliebige Daten Ihrer Anwendung verbinden können.

- *Sortieren*: *TListView* verfügt über einen automatischen alphabetischen und über einen selbst definierbaren Sortiermodus. Weitere Informationen und ein Beispiel finden Sie in Kapitel 8.4.3.

- *Anordnen*: Mit der Methode *Arrange* weisen Sie das ListView in den Icon-Darstellungen an, die Icons in Reihen oder Spalten anzuordnen. Im Programm *DynamicListView* können Sie die Anordnen-Funktion aus dem Popup-Menü aufrufen.

- *Workareas*: Dies sind rechteckige »Arbeitsbereiche« innerhalb der ListView-Fläche, die durch eine gestrichelte Linie abgegrenzt und am unteren Ende mit einem Namen (*DisplayName*) versehen werden. Das ListView beschränkt sich dabei darauf, diese Flächen zu markieren und zu bestimmen, in welcher WorkArea sich ein bestimmter Eintrag des ListViews befindet. Das Programm *DynamicListView* gibt Ihnen Gelegenheit, mit diesen WorkAreas zu experimentieren (hierzu dienen die in Abbildung 3.12 zu sehenden Schalter WORKAREAS ERZEUGEN und WORKAREAS ANALYSIEREN). Da dem Autor jedoch auch zwei Jahre nach Entwicklung dieses Beispiels kein wirklich sinnvoll erscheinender Einsatzzweck für diese Funktion eingefallen ist, wurde die Erläuterung zu diesem Beispiel in die Programmdateien auf der CD verbannt (sollten Sie einen sinnvollen Einsatzzweck für die Workareas kennen, würde sich der Autor über eine Zuschrift freuen).

- *Virtuelle Inhalte*: Seit Delphi 4 erlaubt *TListView* die Darstellung virtueller Inhalte. Bei einem virtuellen ListView werden die Daten nicht in den *Items* gespeichert, sondern das ListView fragt immer, wenn es einen Eintrag zeichnen muss, über das Event *OnData* nach diesen Daten. Auf diese Weise können auch sehr große ListView-Inhalte effizient realisiert werden, weil alle Daten nur nach Bedarf geladen oder berechnet werden müssen. Ein Beispiel hierfür finden Sie im Lieferumfang von Delphi (Verzeichnis DEMO\VIRTUALLISTVIEW). In Delphi 6 wurde übrigens auch die einfache *TListBox* mit der Möglichkeit ausgestattet, virtuelle Inhalte anzuzeigen.

3.6.3 Baumanzeige-Komponenten (TreeViews)

Erfreulicherweise bestehen zwischen *TListView* und *TTreeView* große Ähnlichkeiten, so dass Sie, wenn Sie eine dieser Komponenten kennen, die andere nicht von Grund auf neu kennen lernen müssen. So laufen beispielsweise das Verknüpfen der Einträge mit Datenobjekten und das Editieren der Eintrags-Beschriftungen nach demselben Prinzip ab.

Weitere wichtige Ähnlichkeiten gibt es bei der Organisation der Bilderlisten. Abbildung 3.13 zeigt die Situation für TreeViews. Jeder TreeView-Eintrag verfügt über vier Indizes und damit über einen Index mehr als ein ListView-Eintrag. Dafür hat ein TreeView nur einen einzigen Anzeigemodus und benötigt so insgesamt nur zwei Bilderlisten. Die Unterscheidung zwischen *LargeImages* und *SmallImages* entfällt also zugunsten der Liste *Images*.

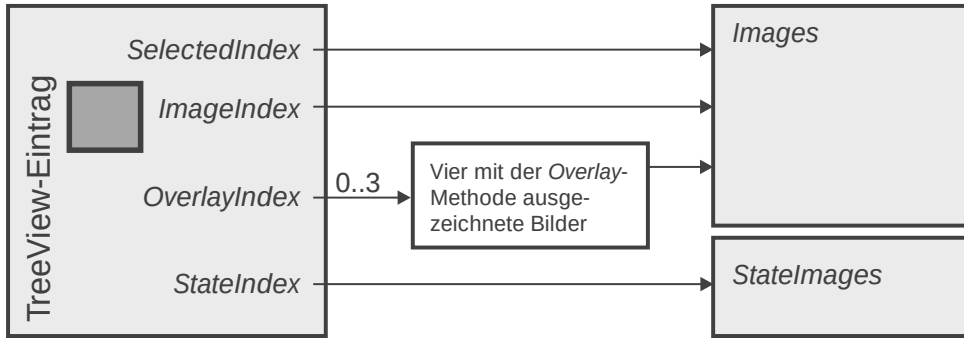


Abbildung 3.13: Die Verknüpfungen zwischen TreeView-Element und Bilderlisten

Die Properties *ImageIndex*, *StateIndex* und *OverlayIndex* haben dieselbe Bedeutung wie die gleichnamigen Properties eines ListViews. *ImageIndex* und *StateIndex* können Sie auch zur Entwurfszeit im Editor für das *Items*-Property des TreeViews einstellen. Zur Laufzeit können Sie die Properties ebenfalls über *TTreeView.Items* ansprechen, die Elemente von *Items* haben allerdings nicht den Typ *TListItem*, sondern den Typ *TTreeNode*.

Der zusätzliche Index *SelectedIndex* (Bezeichnung im TreeView Einträge-Editor: *Ausgewählt-Index*) bezeichnet ein Bild, das dargestellt werden soll, wenn der Eintrag selektiert ist. Dieses Bild tritt dann an die Stelle des mit *ImageIndex* angegebenen Bildes. Der Windows-Explorer stellt beispielsweise den im Verzeichnisbaum ausgewählten Ordner als geöffneten Ordner dar. Um den gewählten Eintrag zur Laufzeit abzufragen, verwenden Sie das Property *TTreeView.Selected*.

TTreeView zur Entwurfszeit

Wie schon bei *TListView* geben Sie auch bei einem *TTreeView* den Text der einzelnen Elemente und die Indizes für die Bilder im Editor für das Property *Items* an. Sogar der Schalter NEUER UNTEREINTRAG stimmt mit dem entsprechenden Editor der ListViews überein – mit dem Unterschied, dass Sie bei TreeView einen echten Untereintrag erhalten, bei dem Sie auch die Felder *Bild-Index*, *Ausgewählt-Index* und *Status-Index* editieren können.

Hinzufügen von Knoten *R56*

Das *Items*-Property von *TTreeView* ist um einiges komplizierter strukturiert als die Eintragsliste eines ListViews, da jedes einzelne Element der Klasse *TTreeNode* als Unterelemente nicht einfach eine Liste von Strings (wie *TListItem*), sondern wieder eine Liste von einzelnen *TTreeNode*-Objekten zugeordnet bekommt. So können Sie mit *TreeView.Items[0][4][0].Text* den Text des ersten Untereintrags des fünften Untereintrags der Wurzel ansprechen.

Um einem Eintrag einen neuen Untereintrag zu spendieren, dürfen Sie sich jedoch nicht an diesen Eintrag wenden (ein Aufruf wie *Items[0].Add* ist also nicht möglich), sondern Sie müssen wieder eine *Add*-Methode der Gesamtliste aufrufen (also *Items.Add...*). Die einfachste Methode davon, *Add(Sibling, S)*, erzeugt einen neuen Knoten mit der Beschriftung *S* und fügt ihn als Geschwister von *Sibling* ein (und zwar an letzter Stelle, falls es sich nicht um einen sortierten TreeView handelt.)

Außerdem gibt es sieben weitere *Add*-Methoden, die sich durch einen Namenszusatz auszeichnen, der aus den Elementen *Child*, *Object* und *First* kombiniert wird:

- Bei den vier *AddChild...*-Methoden geben Sie im ersten Parameter einen Eintrag an, der einen neuen Untereintrag erhalten soll. Bei den Methoden ohne das »Child« im Namen geben Sie statt dessen einen Eintrag der Baumebene an, in der der neue Eintrag eingefügt werden soll (also einen Geschwistereintrag wie bei der einfachen *Add*-Methode).
- Bei den vier *Add[Child]Object...*-Methoden geben Sie als Parameter eine Beschriftung für den neuen Eintrag und einen Zeiger auf zu verknüpfende Daten an (dieser kann daraufhin analog zu den mit einem ListView-Item verknüpften Daten mit *Node.Data* angesprochen werden). Die Methoden ohne das »Object« erwarten nur einen Beschriftungs-String.
- Die vier *Add[Child][Object]First*-Methoden fügen den Eintrag vor alle benachbarten Einträge an; ohne das »First« wird er hinter seinen Geschwistereinträgen eingeordnet.

In Delphi 6 wurde außerdem eine universelle neue *Add*-Methode mit dem Namen *AddNode* eingeführt. Sie erfüllt die Aufgaben aller bisherigen Methoden und ist trotzdem noch flexibler:

```
function AddNode(Node, Relative: TTreeNode; const S: string;
      Ptr: Pointer; Method: TNodeAttachMode): TTreeNode;
```

Im Parameter *Method* geben Sie eine der Konstanten *naAdd*, *naAddFirst*, *naAddChild*, *naAddChildFirst* und *naInsert* an. Die *Add*-Konstanten bewirken dabei, dass der neue Knoten so eingefügt wird wie mit der entsprechenden *Add*-Methode. *Relative* gibt entweder den Geschwister- oder Elternknoten des neuen Knotens an (je nachdem, ob die *Add*- oder *AddChild*-Variante verwendet wird). *S* gibt wie gehabt die Beschriftung für den Knoten an. *Ptr* gibt einen Zeiger auf die zu verknüpfenden Daten an wie die oben genannten *AddObject*...-Methoden.

Der Clou der *AddNode*-Methode ist, dass Sie den neuen Knoten selbst im Parameter *Node* angeben können. Wenn Sie hier *nil* angeben, erzeugt *AddNode* wie die acht anderen *Add*-Methoden den Knoten selbst, und zwar per Voreinstellung als Objekt der vordefinierten Klasse *TTreeNode*. Wenn *Sie* den Knoten erzeugen, haben Sie die Möglichkeit, eine selbst definierte, von *TTreeNode* abgeleitete Klasse zu wählen. Die selbst definierte Klasse kann beispielsweise dazu dienen, die zusätzlichen anwendungsspezifischen Daten mit jedem Knoten zu speichern, die sonst in *Node.Data* untergebracht werden würden. Bei Verwendung einer eigenen *TreeNode*-Klasse ist also die Verwendung von *Node.Data* nicht mehr erforderlich.

Hinweis: Auch ohne den Aufruf von *AddNode* können Sie eine TreeView-Komponente zur Verwendung selbst definierter Knotenklassen bringen, wenn Sie das *OnCreateNodeClass*-Ereignis bearbeiten. Dieses tritt immer dann auch, wenn Sie *nicht* die *AddNode*-Methode aufrufen oder im ersten Parameter von *AddNode* keinen vorbereiteten Knoten, sondern *nil* übergeben.

Beispiel für eine dynamisch erzeugte Baumansicht

Ein gutes Beispiel für dynamisch zur Laufzeit gefüllte TreeViews gibt der ToolsApi-Erforscher aus Anhang A. Die folgenden Listings gelten für den linken seiner beiden TreeViews (Abbildung 3.14). Zunächst wird zu Beginn des Programms (bzw. wenn das Fenster geöffnet wird) die folgende Methode aufgerufen, um die oberste Ebene des TreeViews zu füllen:

```
procedure TExpert1Form.InitTreeView1;
var
  TopNode, node: TTreeNode;
begin
  tv.Items.Clear;
  TopNode := tv.Items.AddObject(nil, 'ToolIntf.ToolServices', nil);
```

```
  node := tv.Items.AddChildObject(TopNode, 'Forms', TFormsData.Create);
  node.HasChildren := true;
  node.ImageIndex := 0;
  TFormsData(node.Data).Node := node;
  node := tv.Items.AddChildObject(TopNode, 'Units', TUnitsData.Create);
  node.HasChildren := true;
  node.ImageIndex := 0;
  TUnitsData(node.Data).Node := node;
  node := tv.Items.AddChildObject(TopNode, 'Module', TModulesData.Create);
  node.HasChildren := true;
  node.ImageIndex := 0;
  TopNode.Expand(false);
  TModulesData(node.Data).Node := node;
end;
```

Abbildung 3.14: Beim Öffnen des Fensters werden diese TreeViews jeweils nur bis eine Ebene unter dem Wurzelknoten initialisiert.

Die Klassen *TFormsData*, *TUnitsData* und *TModulesData* dienen dazu, zusätzliche Informationen mit den entsprechenden Knoten zu speichern; sie sind alle drei von der ebenfalls selbst definierten Klasse *TNodeData* abgeleitet. Im folgenden Beispiel werden wir uns auf den in der Abbildung gezeigten *Forms*-Knoten und damit auf die Klasse *TFormsData* beschränken:

```
type
  TNodeData = class
    Node: TTreeNode;
    Info: string;
    constructor Create;
    procedure CreateSubElements; virtual;
    procedure WriteInfo(m: TMemo); virtual;
  end;
  // Erste Stufe:
  TFormsData = class(TNodeData)
    procedure CreateSubElements; override;
  end;
```

Das Beispielprogramm verknüpft die vom TreeView erzeugten *TTreeNode*-Objekte und die eigenen *TNodeData*-Objekte in beiden Richtungen: Während *TTreeNode.Data* auf das zugehörige *TNodeData*-Objekt weist, führt *TNodeData.Node* in umgekehrter Rich-

tung zum zugehörigen *TTreeNode*. Es sei noch einmal darauf hingewiesen, dass es unter Delphi 6 nicht mehr nötig ist, die zusätzlichen Daten wie in diesem Beispiel die *TFormsData*, *TUnitsData* etc. mit dem Property *TTreeNode.Data* zu verknüpfen, sondern dass *TNodeData* auch direkt von *TTreeNode* abgeleitet werden könnte. *TTreeNode* und *TNodeData* würden dann eine Einheit bilden und die Verknüpfung zwischen beiden würde sich erübrigen.

Die Methode *TNodeData.CreateSubElements* dient im nächsten Abschnitt dazu, die Unterelemente des Eintrags bei Bedarf zu erzeugen.

Effizienter Aufbau des Baumes

Die wohl wichtigste Eigenschaft eines TreeViews ist, dass es einzelne Zweige des Baumes ein- und ausblenden kann. Normalerweise sind Baumstrukturen zu Beginn der Übersicht wegen nur zu einem kleinen Teil geöffnet oder ganz geschlossen (der Windows-Explorer öffnet zum Beispiel nach dem Start nur die Zweige, die zu einem bestimmten Verzeichnis führen).

Es wäre aber ungünstig, wenn zwar der Anwender von der Komplexität des Baumes verschont würde, das Programm den Baum aber immer komplett aufbauen müsste. Dies könnte z.B. für den Verzeichnisbaum einer Festplatte sehr lange dauern. Jeder Baumeintrag hat daher ein Property namens *HasChildren*, in dem Sie mit dem Wert *True* angeben können, dass der Eintrag Unterelemente hat, obwohl Sie diese noch nicht in die Eintragsliste eingefügt haben.

Sofern Sie das TreeView-Property *ShowButtons* bei seinem voreingestellten Wert *False* lassen, kennzeichnet der TreeView sowohl die Einträge, die bereits Kinder haben, als auch die Einträge, bei denen nur das Property *HasChildren* auf *True* gesetzt ist, mit einem Plus-Symbol. Wenn der Anwender auf dieses Symbol klickt, wird der Zweig expandiert und das Symbol verwandelt sich in ein Minus-Zeichen.

Beim Expandieren erzeugt *TTreeView* ein *OnExpanding*-Ereignis, das Ihnen die Gelegenheit gibt, die Kindelemente des Knotens hinzuzufügen (wenn Sie dann keine Kindelemente hinzufügen, wird das Plus-Symbol neben dem expandierten Knoten nicht durch das Minus-Symbol ersetzt, sondern ersatzlos entfernt).

Im oben begonnenen Beispiel wird *OnExpanding* wie folgt bearbeitet (der Parameter *Node* gibt den Knoten an, der expandiert werden soll, diese Expansion könnte in der Ereignisbearbeitung verhindert werden, indem der *AllowExpansion*-Parameter auf *False* gesetzt wird):

```
procedure TExpert1Form.tvExpanding(Sender: TObject; Node: TTreeNode;
  var AllowExpansion: Boolean);
begin
  if Node.GetFirstChild = nil then
    TNodeData(Node.Data).CreateSubElements;
end;
```

Zuerst wird überprüft, ob der Knoten bereits Kindelemente hat. Ist das nicht der Fall, wird die in *TNodeData* definierte *CreateSubElements*-Methode über das *Data*-Property des expandierten Knotens aufgerufen. Dies setzt natürlich voraus, dass das Programm das *Data*-Property für alle expandierbaren Knoten auf ein *TNodeData*-Objekt gesetzt hat (Sie können dies – zumindest für die Knoten der obersten Ebene – am obigen Listing der Methode *InitTreeView1* überprüfen, in der das *Data*-Property aller neu erzeugten Knoten entsprechend gesetzt wird).

Welche Kindelemente nun beim Expandieren hinzukommen, hängt vom expandierten Knoten ab. So soll etwa der *Forms*-Knoten des Beispiels die Formulare anzeigen, die im aktuell in der Delphi-IDE geladenen Projekt enthalten sind, der *Units*-Knoten alle Units des Projekts. Das Beispielprogramm definiert daher für jede Knotenart eine eigene Klasse, die von der oben gezeigten Klasse *TNodeData* abgeleitet ist und mit einer eigenen *CreateSubElements*-Methode aufwartet. Als Beispiel sei wie schon angekündigt die Klasse *TFormsData* für den *Forms*-Knoten herausgegriffen:

```
procedure TFormsData.CreateSubElements;
var
  ChildNode: TTreeNode;
  i: integer;
begin
  if ToolServices <> nil then begin
    if ToolServices.GetFormCount = 0 then Node.HasChildren := false;
    for i := 0 to ToolServices.GetFormCount-1 do begin
      ChildNode  :=  TTreeView(Node.TreeView).Items.
        AddChildObject(Node, ToolServices.GetFormName(i)+
                       '  [ToolServices.GetFormName('+IntToStr(i)+')]',
                       TFormData.Create);
      ChildNode.HasChildren := true;
      ChildNode.ImageIndex := 2;
      ChildNode.SelectedIndex := ChildNode.ImageIndex;
      TFormData(ChildNode.Data).Node := ChildNode;
    end;
  end;
end;
```

Hier könnte sich der Prozess der dynamischen Erweiterung nun rekursiv wiederholen: *TFormsData.CreateSubElements* erzeugt für jeden Formular-Knoten ein Objekt der Klasse *TFormData*, die wieder über eine eigene *CreateSubElements*-Methode verfügt. Im Beispiel fügt diese Methode jedoch keine neuen Knoten hinzu, so dass neben den von der obigen Methode neu erzeugten Knoten zwar ein Plus-Symbol angezeigt wird (weil *HasChildren* gesetzt wurde), dieses aber verschwindet, wenn Sie es zur Laufzeit des Programms anklicken. Sie finden im vollständigen Programm-Code auf der CD jedoch noch vier andere *CreateSubElements*-Methoden, die durchaus neue Knoten für diesen TreeView erzeugen (für den zweiten TreeView des Programms wiederholt sich das gleiche Spiel; für Abbildungen des kompletten Beispielformulars siehe Anhang A).

3.6.4 Der Mediaplayer

Dieses Kapitel demonstriert die Fähigkeiten der Komponente *TMediaPlayer* am Beispiel eines programmierbaren CD-Spielers mit automatischer Titelspeicherung (Abbildung 3.15).

Eine MediaPlayer-Komponente stellt nach außen hin ein kleines Schaltpult dar, mit dem der Benutzer die Wiedergabe oder die Aufnahme einer Multimedia-Datei steuern kann. Intern kapselt *TMediaPlayer* einen wichtigen Teil des MCI (Media Control Interface), eine der Multimedia-Schnittstellen von Windows, und stellt einige Properties zur Verfügung, mit denen Sie den über die Schalter bereitgestellten Funktionen weitere Zusatzfunktionen hinzufügen können. Mit einer *TMediaPlayer*-Komponente genügt bereits ein Handgriff (Setzen des Properties *FileName*, siehe Abschnitt *Dateien wiedergeben*), um Ihrer Anwendung die Multimedia-Funktionen zu verleihen, die in dieser Komponente bereits integriert sind (die über die Schalter bereitgestellten Funktionen wie *Pause* und *Play*). Mit der Programmierschnittstelle der Komponente können Sie jedoch auch selbst neue Funktionen schreiben (hierzu ist es noch nicht einmal erforderlich, dass die Schalter der Komponente zur Laufzeit sichtbar sind).

Medientypen

Die multiple Auswahl der Media-Typen spiegelt sich im Aufzählungstyp *TMPDeviceTypes* wider, dessen Werte Sie dem Property *DeviceType* des Mediaplayers zuweisen können:

```
TMPDeviceTypes =
  (dtAutoSelect, dtAVIVideo, dtCDAudio, dtDAT, dtDigitalVideo,
   dtMMMovie, dtOther, dtOverlay, dtScanner, dtSequencer, dtVCR,
   dtVideodisc, dtWaveAudio);
```

Die wohl am häufigsten verwendeten Typen sind *dtWaveAudio* (Klangdateien im WAV-Format), *dtAVIVideo* für Videodateien im AVI-Format und *dtCDAudio* für die Musikwiedergabefähigkeiten eines CD-ROM-Laufwerks.

Dateien wiedergeben *R166*

Die wohl einfachste Einsatzmöglichkeit des *Mediaplayers* ist, ihn als Abspielgerät für Multimedia-Dateien einzusetzen. Per Voreinstellung ist der Typ des Mediums bereits auf *dtAutoSelect* eingestellt, was bedeutet, dass der Mediaplayer den Typ anhand der Dateiendung feststellt. Sie müssen dann lediglich das Property *FileName* setzen, um dem Mediaplayer mitzuteilen, welche Datei er abspielen soll. Bei Videos können Sie im Property *Display* zusätzlich eine Komponente festlegen, auf deren Oberfläche das Video abgespielt werden soll, andernfalls wird ein eigenes Video-Fenster erstellt.

Um die in *FileName* angegebene Datei abzuspielen, können Sie ganz auf die Schalter der *TMediaPlayer*-Komponente vertrauen, über die der Benutzer das Abspielen der Datei nach Belieben starten, unterbrechen und beenden kann, oder Sie können diese Aktionen selbst ausführen, indem Sie die *TMediaPlayer*-Methoden *Play*, *Stop*, *Pause*, *Next*, *Previous*, *Step*, *Back*, *StartRecording* und *Eject* selbst aufrufen.

Weitere Details zu den Gerätetypen und den einfachen Abspielmethoden gibt Ihnen die Online-Hilfe, wir wenden uns hier einem etwas anspruchsvolleren praktischen Beispiel zu.

Das CD-Player-Beispielprogramm

Da eine CD normalerweise nicht nur einen Titel enthält, erfordert ein CD-Player üblicherweise einen höheren Programmieraufwand als das Abspielen einer Datei. Abbildung 3.15 zeigt den CD-Player (der auch auf der CD-ROM *CDPlayer* heißt), der neben dem *MediaPlayer*-Schaltpult als wichtigstes Element eine Tabelle für die auf der CD befindlichen Titel enthält. Eingebaut sind bereits die folgenden Funktionen:

- Nummerierung der Titel und Darstellung der Titellängen in der Tabelle
- Benennung der Titel und Abspeichern aller zu einer CD gehörenden Titel mitsamt einer Titelbezeichnung für die CD
- automatische Identifikation der CD anhand der Längen der ersten beiden Titel und automatisches Laden der gespeicherten Titelbezeichnungen
- Anzeige der aktuellen Position, der Titelnummer und der Titelbezeichnung
- direkte Anwahl eines Titels durch Doppelklicken auf den Eintrag der Tabelle
- Spielen einer Schleife zwischen zwei Positionsmarken, die mit dem Schalter *Loop* gesetzt werden
- Über den Schalter *Programm* gelangen Sie in den ebenfalls in Abbildung 3.15 sichtbaren Dialog, in dem Sie eine aus bis zu 999 Einzeltiteln bestehende Titelfolge programmieren können.
- Über zwei Buttons können Sie innerhalb eines Titels in verschieden großen Schritten vor- und rückwärts springen, wobei Sie die Schrittgröße aus einem Popup-Menü dieser beiden Schalter wählen.

Zu den teilweise einfach erweiterbaren Grenzen des CD-Players gehören:

- Das Programm kümmert sich nicht um das Einlegen von CDs: Wenn keine CD eingelegt ist, zeigt die Positionsanzeige ungültige Werte an, und bei einem CD-Wechsel müssen Sie explizit den Schalter *Neu lesen* drücken, damit die Anzeige des Programms aktualisiert wird.

- Der Programm-Modus ist nicht so komfortabel wie bei einem Hardware-CD-Spieler: Sie können das Programm nur stoppen, aber nicht anhalten oder innerhalb des Programms Titel überspringen. Um diese Funktionen zu implementieren, müssten die *OnClick*-Ereignisse der Tasten *Pause*, *Vor* und *Zurück* der Mediaplayer-Komponente bearbeitet werden.

Damit der CD-Player funktioniert, müssen Sie einen MCI-Treiber für CD-Audio unter Windows installiert haben.

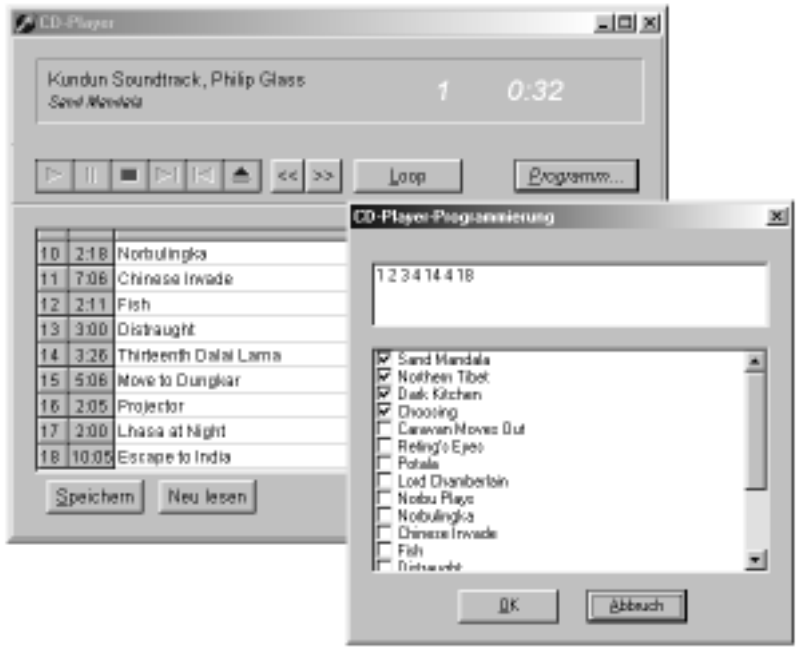


Abbildung 3.15: Die Mediaplayer-Komponente (bestehend aus den quadratischen Schaltern) spinnt ihre Fäden zum größten Teil unter dieser Oberfläche.

Für den CD-Player wurden zur Entwurfszeit bereits drei Properties angepasst:

- Da für CDs kein Dateiname angegeben werden kann, müssen Sie in *DeviceType* angeben, mit welchem Medium der Mediaplayer arbeiten soll. Für CD-Audio ist das der Wert *dtCDAudio*.
- Indem Sie *AutoOpen* auf *True* setzen, ersparen Sie sich einen manuellen Aufruf der Methode *Open*. Falls bei Programmstart keine CD eingelegt ist, kommt es so allerdings zu einer Exception.

- Die bei einem CD-Player nicht benötigten Schalter *btStep*, *btBack* und *btRecord* können Sie im Property *VisibleButtons* abschalten.

Zur Laufzeit verwendet *CDPlayer* einige weitere *TMediaPlayer*-Properties und ruft die Methoden *Play* und *Pause* selbst auf. Im Folgenden werden einige dieser Properties und Methoden erklärt, einige weitere Methoden des Beispielprogramms sind jedoch aus Platzgründen nur auf der CD-ROM zu finden.

Hinweis: In Delphi 4 funktioniert die *AutoOpen*-Option nur dann im Modus *dtAudioCD*, wenn Sie im *FileName*-Property schon zur Entwurfszeit das Laufwerk angeben, von dem die CD abgespielt werden soll (z.B. als »K:«). Dies ist natürlich ungünstig, wenn der CD-Spieler auf verschiedenen Computer-Systemen laufen soll, deren CD-ROM-Laufwerke verschiedene Laufwerksbuchstaben haben. Damit es auch mit Delphi 4 keine derartigen Probleme gibt, arbeitet das Projekt auf der CD zum Buch mit einem *AutoOpen* von *False* und sucht bei der Erzeugung des Formulars nach einem CD-ROM-Laufwerk, dessen Laufwerksbuchstaben es dann in das *FileName*-Property einträgt.

Informationen über die Titel

Von den Properties *Tracks* und *TrackLength* erfährt der CD-Player, wie viele Titel die CD hat und wie lang die einzelnen Titel sind. Die Methode *UpdateInfo* füllt die Tabelle mit diesen Daten der aktuellen CD. Zuerst passt sie die Zeilenzahl der *StringGrid*-Komponente an die *Track*-Zahl der CD an, dann füllt sie die ersten beiden Spalten mit Titelindex und Titeldauer:

```
procedure TForm1.UpdateInfo;
var
  i: Integer;
begin
  MediaPlayer1.TimeFormat := tfHMS;
  StringGrid1.RowCount := MediaPlayer1.Tracks+1;
  { die erste Zeile des StringGrids dient nur zur
    optischen Gestaltung }
  for i := 1 to MediaPlayer1.Tracks do begin
    StringGrid1.Cells[0, i] := Format('%3d', [i]);
    StringGrid1.Cells[1, i] :=
      LengthToString(MediaPlayer1.TrackLength[i]);
  end;
  LoadDescription;
end;
```

Zeitformate

Für die Titeldauer bemüht die obige Methode ein weiteres Property: In *TimeFormat* sagen Sie einer *TMediaPlayer*-Komponente, in welchem Format verschiedene Zeitangaben wie Start- und Endposition gemacht werden sollen. So gibt es beispielsweise Angaben in Millisekunden, in Bytes, in Video-Frames und verschiedenen weiteren Kombinationen. In jedem Fall besteht die Zeitangabe aus einem 4 Byte großen LongInt, dessen einzelne Bytes je nach Format eine andere Bedeutung haben.

In diesem CD-Player wird normalerweise das Format *tfTMSF* verwendet, bei dem das erste Byte die Titelnummer (*T* wie Track) und die beiden folgenden Bytes (*M* und *S*) die Minuten und Sekunden angeben. Bei den Längen der Titel interessiert sich der CD-Player nur für Minuten und Sekunden, allerdings befinden sich diese auch bei eingestelltem Format *tfTMSF* nicht in den beiden mittleren, sondern in den ersten beiden Bytes des LongInts.

Der folgende Code interpretiert den LongInt als vierteiligen Record, wandelt Minuten und Sekunden mit *EncodeTime* in eine Delphi-Zeitangabe um und formatiert diese mit *FormatDateTime* zu einer lesbaren Zeitangabe:

```
type
  TTimeF = record
    a, b, c, d: Byte;
  end;

function LengthToString(Time: LongInt): string;
begin
  Result := FormatDateTime('n:ss',
              EncodeTime(0, TTimeF(Time).a, TTimeF(Time).b, 0));
  { rechtsbündiges Auffüllen bis zu 5 Zeichen Länge: }
  while length(Result) < 5 do Result := ' '+Result;
end;
```

Positionssteuerung *R167*

Nun geht es darum, nach einem Doppelklick auf einen Titel diesen auch zu spielen. Dazu muss der CD-Player zuerst die Koordinaten des Doppelklicks herausfinden. Er merkt sich zu diesem Zweck bei jedem *OnMouseMove*-Event die neuesten Mauskoordinaten:

```
procedure TForm1.StringGrid1MouseMove(Sender: TObject;
                      Shift: TShiftState; X, Y: Integer);
{ Festhalten der "MousePos" für die OnDblClk-Bearbeitung: }
begin
  MousePos := Point(x, y);
end;
```

Gerüstet mit dieser Information und mit der Hilfe der *TStringGrid*-Methode *MouseToCell* kann die Methode für das *OnDblClick*-Ereignis nun die gewünschte Titelnummer (*Track*) herausfinden, es handelt sich einfach um die Nummer der angeklickten Zeile (die erste Zeile hat die Nummer 0).

```
const
  WhilePlaying: TButtonSet =
    [btPause, btStop, btNext, btPrev, btEject];
  WhilePausing: TButtonSet =
    [btPlay, btNext, btPrev, btEject];

procedure TForm1.StringGrid1DblClick(Sender: TObject);
var
  ColDummy, Track: LongInt;
begin
  StringGrid1.MouseToCell(MousePos.x, MousePos.y, ColDummy, Track);
  with MediaPlayer1 do begin
    StartPos := Track;
    Play;
    AutoEnable:=False;
    EnabledButtons := WhilePlaying;
  end;
end;
```

Um also den Titel mit der Nummer *Track* zu spielen, weisen Sie diese Nummer einfach dem *MediaPlayer*-Property *StartPos* zu (dies entspricht dem *tfTMSF*-Wert von *T=Track*, *M=0* und *S=0*) und starten die Wiedergabe mit *Play*. Die gezeigte Methode verwendet schließlich noch die beiden folgenden Properties:

- *EnabledButtons* gibt die Schalter des Mediaplayers an, die aktiv sein sollen. Da die Mediaplayer-Komponente die Aktivierung ihrer Schalter nur dann anpasst, wenn der Benutzer einen der Schalter drückt, muss das Beispielprogramm sie selbst anpassen, wenn der Mediaplayer über eine eigene Benutzerschnittstelle gesteuert wird. Beim Starten eines Titels per Doppelklick in die Titelliste sollte beispielsweise der *Stop*-Schalter aktiviert und der *Play*-Schalter deaktiviert werden.
- Allerdings bewirkt erst *AutoEnable*, dass *TMediaPlayer* den Wert von *EnabledButtons* überhaupt berücksichtigt. Ist *AutoEnable=False*, ignoriert der MediaPlayer das Property *EnabledButtons* einfach (er verändert es also auch nicht). Das Beispielprogramm schaltet *AutoEnable* bei jedem *OnClick*-Ereignis des MediaPlayers wieder ein.

Positionsanzeige

Um die aktuelle Position anzuzeigen (Titel / Minuten / Sekunden) verwendet *CDPlayer* eine Timer-Komponente, die jede Sekunde ein *OnTimer*-Ereignis auslöst. Die zugehörige Methode (eine der hier aus Platzgründen nicht abgedruckten Methoden) liest das

TMediaPlayer-Property *Position* aus, das im Format *tfTMSF* zu verstehen ist, dessen niedrigstwertiges Byte (»das erste von links«) also die Spurnummer enthält, gefolgt von Minuten und Sekunden:

```
function PosToString(Time: LongInt): string;
begin
  Result := IntToStr(TTimeF(Time).a);
  Result := Result+'       '+
    FormatDateTime('n:ss',
      EncodeTime(0, TTimeF(Time).b, TTimeF(Time).c, 0));
end;
```

Die Programmieren-Dialogbox

Abbildung 3.15 zeigt auch den Dialog des Beispielprogramms, in dem Sie eine Titelfolge als Programm für den CD-Player eingeben. Sie haben dazu die folgenden Möglichkeiten:

- Editieren Sie das obere Feld, in dem die Titelfolge als reine Zahlenfolge angegeben wird.
- Doppelklicken Sie auf einen Titel in der Liste, um den Titel im oberen Feld an der aktuellen Cursorposition einzufügen.
- Klicken Sie auf das Markierungsfeld, um einen Titel erstmalig in das Programm einzufügen (wenn das Markierungsfeld noch nicht gesetzt ist), oder um alle Vorkommnisse des Titels aus dem Programm zu löschen (wenn das Markierungsfeld bereits gewählt ist).

Die Markierung eines Titels in der CheckListBox bedeutet also, dass dieser Titel mindestens einmal im Programm vorkommt. Diese Verwendung der *TCheckListBox* ist daher ungewöhnlich, da Sie einen Titel durch Anklicken des Markierungsfeldes nur einmal zum Programm hinzufügen, aber durch Entfernen der Markierung gleich ein mehrfaches Vorkommen löschen können. Für diese Funktionalität genügt es, die Ereignisse *OnCheckClick* und *OnDblClk* der CheckListBox zu bearbeiten. Da diese Methoden im Beispielprogramm nichts mit *TMediaPlayer* zu tun haben, sondern hauptsächlich die Titelliste und den String des Editierfeldes zu verarbeiten haben, sei für Details auf den Quelltext auf der CD verwiesen.

Multimedia-Benachrichtigungen *R168*

Als Letztes soll das Konzept der Multimedia-Benachrichtigung anhand des Beispiels programmierter Titelfolgen vorgestellt werden. Da die Wiedergabe von Multimedia-Daten den Computer nicht blockieren, sondern quasi nebenbei ablaufen soll, kann das Programm beispielsweise beim Starten des ersten Titels aus einem Programm nicht warten, bis die Wiedergabe beendet ist. Die Wiedergabe läuft also alleine weiter, was

beim CD-Player dann ausreicht, wenn einfach die ganze CD am Stück abgespielt werden sollen. Befindet sich der CD-Spieler jedoch im Programm-Modus, muss unsere Anwendung wissen, wann das Abspielen eines Titels beendet ist, damit sie die aktuelle Spielposition der MediaPlayer-Komponente (Property *Position*) auf den nächsten programmierten Titel setzen kann.

Hierzu sind die Multimedia-Benachrichtigungen gedacht. Eine *TMediaPlayer*-Komponente stellt Ihnen diese im Event *OnNotify* zur Verfügung. Standardmäßig wird nach dem Abschluss von *Play* und *Record* eine solche Benachrichtigung erzeugt (falls Sie auch nach anderen Befehlen eine Benachrichtigung haben wollen, benötigen Sie zusätzlich das Property *Notify*). Hier ist zunächst der Code, der beim Starten des Titel-Programms aufgerufen wird:

```
// aus der OnClick-Bearbeitung des Schalters Programm...:
TheProgram := CDPProgDialog.ProgramInEdit;
ProgButton.Font.Style := ProgButton.Font.Style + [fsItalic];
if ProgramPos=-1 then begin // nur wenn nicht schon ein Programm läuft.
  ProgramPos := 0;
  with MediaPlayer1 do begin
    Stop;
    StartPos := TrackPosition[TheProgram[0]];
    EndPos := StartPos+((TrackLength[TheProgram[0]] and $0000FFFF) shl 8);
    // wie schon unter "Zeitformate" beschrieben, befinden sich die
    // Minuten und Sekunden in TrackLength in den beiden untersten Bytes.
    // mit "and $0000FFFF" werden die beiden obersten Bytes gelöscht
    // und mit shl 8 werden Minuten und Sekunden in die Mitte geschoben,
    // um mit dem Format tfTMSF kompatibel zu sein.
    EnabledButtons := WhilePlayingProgram; // = [btStop, btEject]
    AutoEnable := False;
    Play;
    OnNotify := NotifyNextProgramTrack;
  end;
end;
```

In *EndPos*, dem Gegenstück zu *StartPos*, wird angegeben, bei welcher Position die Wiedergabe automatisch stoppen soll. In diesem Fall ist es das Ende des ersten Titels im Programm (*TheProgram[0]*). Die Endposition ist also StartPosition + Länge des ersten Titels und muss auf etwas umständliche Weise berechnet werden.

EnabledButtons und *AutoEnable* wurden schon für den Start von Titeln per Doppelklick erläutert, so dass wir nun zur Formularmethode *NotifyNextProgramTrack* kommen können, die im obigen Listing mit dem *OnNotify*-Ereignis verknüpft wird. Ihre wichtigste Aufgabe ist es, den nächsten Titel des Programms zu starten.

NotifyNextProgramTrack erhöht den Programmindex um eins und beendet gegebenenfalls den Programm-Modus. Wenn noch ein weiterer Titel abzuspielen ist, wird dieser mit *StartPos/EndPos/Play* so programmiert, wie oben schon beim Start des Programms.

Außerdem fragt *NotifyNextProgramTrack* noch das Property *NotifyValue* ab, das Sie sich statt als *TMediaPlayer*-Property vielleicht besser als Parameter für die *OnNotify*-Methode vorstellen. Nur bei einem *NotifyValue* von *nvSuccessful* wurde der letzte *Play*-Befehl vollständig ausgeführt (d.h. bis zum Erreichen der *EndPos*). Wenn der Benutzer beispielsweise den Stop-Schalter drückt, wird der laufende *Play*-Befehl durch einen *Stop*-Befehl abgelöst und *NotifyValue* erhält den Wert *nvSuperseded*:

```
procedure TCDPMainForm.NotifyNextProgramTrack(Sender: TObject);
begin
  with Mediaplayer1 do
    if (MediaPlayer1.NotifyValue=nvSuccessful) then begin
      if ProgramPos<>-1 then begin
        inc(ProgramPos);
        if TheProgram[ProgramPos]=0 then begin
          Stop;
          ProgramStopped; // siehe hierzu Quelltext auf der CD
        end else begin
          StartPos := TrackPosition[TheProgram[ProgramPos]];
          EndPos := StartPos+((TrackLength[TheProgram[ProgramPos]]
            and $0000FFFF) shl 8); // Erläuterung siehe erster Codeauszug
          Play;
        end;
      end
    end else if (MediaPlayer1.NotifyValue=nvSuperseded) then begin
      ProgramStopped;
      MessageDlg('Programm wurde abgebrochen.', mtInformation, [mbOk], 0);
    end;
end;
```

Hinweise: Im Beispielprogramm wurde *OnNotify* nicht schon zur Entwurfszeit mit einer Methode verknüpft, da dieses Event auch noch für die Musikschleife verwendet wird. Der Übersicht halber verwendet das Programm dafür eine eigene, zweite Methode. Zur Laufzeit wird *OnNotify* also mal auf die oben gezeigte, mal auf die andere Methode gesetzt (siehe Code auf der CD).

Beachten Sie außerdem, dass Sie nach dem Starten von *Play* eine *nvSuperseded*-Benachrichtigung »versehentlich« auslösen können, wenn Sie bestimmte Properties oder Methoden von *TMediaPlayer* an anderen Stellen des Programmcodes ansprechen.

Ebenfalls mit Multimedia-Benachrichtigungen, aber erheblich einfacher realisieren lässt sich die Musikschleife des Beispielprogramms. Wenn der Anwender am Anfang und am Ende des Musikabschnitts, der wiederholt werden soll, den Schalter *Loop* drückt, merkt sich das Programm einfach die aktuelle Position, setzt *TMediaPlayer.Start-*

Pos und *EndPos* und lässt sich sodann von jedem Erreichen der *EndPos* benachrichtigen, um dann gleich eine Wiederholung veranlassen zu können. Siehe hierzu auf der CD die Methode *Button3Click*.

Der CD-Detektor

Viele der weiteren, hier nicht abgedruckten Methoden haben weniger mit der *TMediaPlayer*-Komponente zu tun. Sie speichern die Titelangaben des Benutzers in einer Datei, deren Name automatisch aus den Längen der ersten beiden Titel gebildet wird. Dazu werden Minuten und Sekunden hexadezimal hintereinander geschrieben, so dass der Dateiname genau acht Zeichen lang wird, mit Endung wird daraus z.B. »032F0338.CDD«:

```
with MediaPlayer1 do begin
  len1 := TrackLength[1];
  len2 := TrackLength[2];
  Dateiname := IntToHex(TTimeF(len1).a, 2)+
               IntToHex(TTimeF(len1).b, 2)+
               IntToHex(TTimeF(len2).a, 2)+
               IntToHex(TTimeF(len2).b, 2)+
               '.CDD';
end;
```

Bei Programmstart lädt der CD-Player automatisch die zur aktuellen CD passende Datei. Um nach einem CD-Wechsel die Beschriftung zu dieser CD zu laden, drücken Sie den Schalter *Neu lesen*. Änderungen an den Titeln werden nicht automatisch gespeichert, hierfür ist der Schalter *Speichern* gedacht.

3.7 Frames und verwandte Techniken

Trotz der extremen Beschleunigung, die die Softwareentwicklung beim Entwurf der Benutzerschnittstelle durch Delphi erfahren hat, bleibt noch enorm viel Spielraum für weitere Effizienzsteigerungen, wenn man von Delphis reinem Formular-Editor ausgeht. Zu häufig gibt es beim visuellen Entwurf wiederkehrende Aufgaben, die bei der x-ten Wiederholung sehr lästig werden können und die nach Vereinfachung verlangen: Manche Properties müssen immer auf dieselben Werte eingestellt werden (z.B. TPanel.Caption auf einen leeren String, damit die Panel-Oberfläche frei wird), einfache Formularbestandteile müssen immer aus denselben Komponenten zusammengebaut werden (so etwa die Kombination aus Label und Editierfeld) und in größeren Anwendungen mit vielen Formularen kann es vorkommen, dass viele Formulare sich ähneln und zum Teil aus den gleichen Komponenten bestehen. Zu den Komponenten und deren Properties können dann auch noch Ereignismethoden kommen, die ebenfalls immer gleich sind.

Ein Beispiel für eine mehrfach vorkommende Komponentengruppe wäre z.B. in einer Firmendatenverwaltung mit verschiedenen Personenlisten für Kunden und Mitarbeiter, die zum Teil die gleichen Tabellenspalten wie Name, Vorname, Straße usw. enthalten, zum Teil aber auch andere Felder aufweisen. Die Dialoge zum Ändern von Kunden- und Mitarbeiterdaten würden dann in den Eingabefeldern für die Adresse übereinstimmen. Ein anderes Beispiel wäre es, wenn eine Komponentengruppe zur Auswahl eines Verzeichnisses und einer Datei in vielen verschiedenen Formularen verwendet wird.

Delphi bietet eine ganze Reihe von Mechanismen an, mit der sich der Entwurf von Formularen mit wiederkehrenden und ähnlichen Bestandteilen vereinfachen lässt. Die Palette dieser Mechanismen nimmt fast mit jeder neuen Delphi-Version zu:

- Seit Delphi 1 lassen sich komplette Formulare und Projekte bequem wiederverwenden, wenn Sie eine Schablone davon in der Objektablage ablegen (siehe auch Kapitel 1.6.4).
- In Delphi 2 wurde diese Verwendung von Formularschablonen näher an die Prinzipien der OOP angenähert, indem ein Formular nicht unbedingt eine exakte Kopie einer Schablone sein muss, sondern von einem Schablonenformular erben kann.
- Delphi 3 brachte der Entwickler-Gemeinde eine Technik zur Wiederverwendung von Komponentengruppen über den Weg von Komponentenschablonen, die in der Komponentenpalette abgelegt werden können.
- Eine ganz allgemeine und mächtige Technologie zur Wiederverwendung von Komponentengruppen liegt natürlich in der Entwicklung neuer Komponentenklassen. Bereits seit der ersten Delphi-Version ist es möglich, mehrere Komponenten zu einer neuen Komponente zusammenzufassen (siehe Kapitel 6.5.4), allerdings findet diese Komponentenentwicklung nicht-visuell statt, fällt also hier ein wenig aus dem Rahmen.
- Seit Delphi 5 gibt es das Konzept der *Frames*, bei dem das Prinzip der Vererbung (in Delphi 2 schon bei Formularen bekannt) auf Komponentengruppen Anwendung findet.

Ziel dieses Kapitels ist es, die Vor- und Nachteile dieser Techniken an Beispielen zu erläutern und den Einsatz von Formularvererbung und Frames zu demonstrieren.

3.7.1 Formularvererbung

Die Formularvererbung ist ein Teil des Funktionsangebots der Objektablage und wird in Kapitel 1.6.4 im Einzelnen beschrieben. An dieser Stelle geht es um ein Beispiel, das in Kapitel 3.7.2 über die Frames fortgesetzt wird.

Als Beispiel soll eine Situation dienen, in der sich die Vererbung der Formulare aus der Vererbung von nicht-visuellen Klassen des Quelltextes herleiten lässt. Nehmen wir eine Anwendung an, in der es verschiedene Klassen von Grafikobjekten gibt, die von einer gemeinsamen Basisklasse abgeleitet sind. Die Basisklasse definiert die Eigenschaften, die allen Grafikobjekten gemein sein sollen, so etwa die Koordinaten des umgebenden Rechtecks und die Stift- und Pinselattribute zum Zeichnen des Objekts. Als spezielle davon abgeleitete Klasse könnte es z.B. eine Klasse geben, die einen Farbverlauf zwischen zwei beliebigen Farben darstellen sollen. Sie müsste die Daten der Basisklasse durch zwei Farbwerte ergänzen. Ein anderes Beispiel wären spezielle Grafikobjekt-Klassen für Diagrammsymbole, wie sie in CASE-Tools zur Darstellung eines Software-Entwurfs verwendet werden. Die Objekte für diese Symbole könnten den Daten der Basisklasse z.B. diverse Beschriftungen für verschiedene Stellen des Symbols hinzufügen, wie sie z.B. in UML-Diagrammen zu finden sind.

Was an dieser Stelle nicht interessiert, ist die Problematik der Wahl der günstigsten Klassenaufteilung einer Grafikanwendung (so wäre es vielleicht günstiger, Farbverläufe schon als Eigenschaft der grundlegenden Basisklasse zu implementieren, so dass *alle* speziellen Grafikobjekte über die Option eines Farbverlaufs verfügen würden). Thema dieses Kapitels soll es ausschließlich sein, eine gegebene Vererbungshierarchie auf den Formularentwurf zu erweitern.

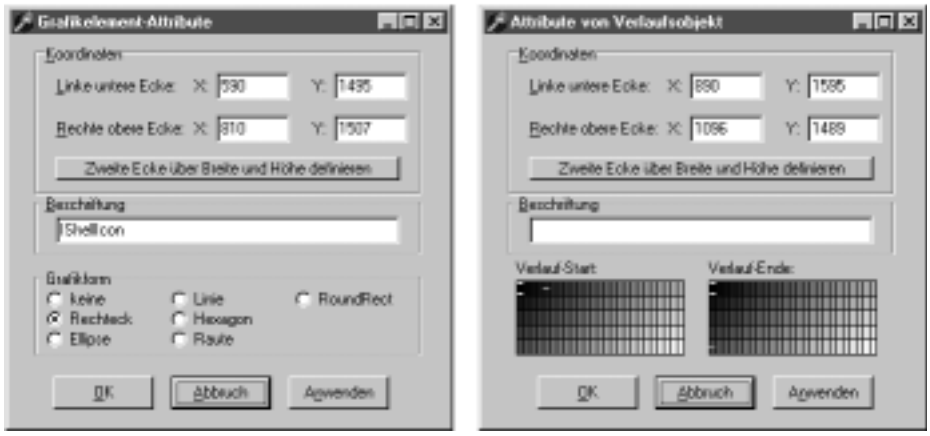


Abbildung 3.16: Das Ziel ist, diese beiden und eventuell viele weitere Grafikobjekt-Dialoge zu entwerfen und dabei die gemeinsamen Komponenten nur einmal an einer zentralen Stelle zu definieren.

Abbildung 3.16 zeigt zwei Dialoge, die im beschriebenen Beispiel verwendet werden könnten. Das Problem für den Formular-Entwickler besteht nun darin, die gemeinsamen Teile des Dialogs nicht jedes Mal neu entwerfen zu müssen. Im Fall einer Anwen-

dung mit nur zwei Dialogen kann man natürlich auf alle besonderen Funktionen von Delphi verzichten und die gemeinsamen Komponenten einfach über die Zwischenablage vom ersten in das zweite Formular kopieren.

Diese Lösung liegt jedoch nicht im Sinn der OOP und greift bei einer Anwendung, die später um viele weitere Grafikklassen und somit auch etliche neue Dialoge erweitert werden soll, etwas kurz. Probleme gibt es spätestens dann, wenn der Entwurf der gemeinsamen Dialogelemente geändert werden soll: Wenn diese Elemente erst einmal in alle Dialogformulare kopiert sind, muss auch jede Änderung an ihnen erneut in jedes einzelne Formular übertragen werden.

Delphis Objektablage bietet mit den Formularschablonen eine einfache Lösung für das vorliegende Problem. Abbildung 3.17 zeigt den Entwurf eines Formulars mit den gemeinsamen Komponenten der beiden Dialoge aus Abbildung 3.16 sowie den Dialog der Objektablage, den Sie mit DATEI | NEU aufrufen und über den Sie auf Grundlage des bestehenden Formulars ein neues Formular über die Formularvererbung erzeugen können. Die Formulare des aktuellen Projekts sind im DATEI | NEU- bzw. Objektablage-Dialog auf einer eigenen Seite mit dem Namen des Projekts angeordnet. Um die Vererbung zu nutzen, stellen Sie vor dem Drücken des Ok-Schalters sicher, dass im unteren Teil des Dialogs die Option VERERBEN gewählt ist.

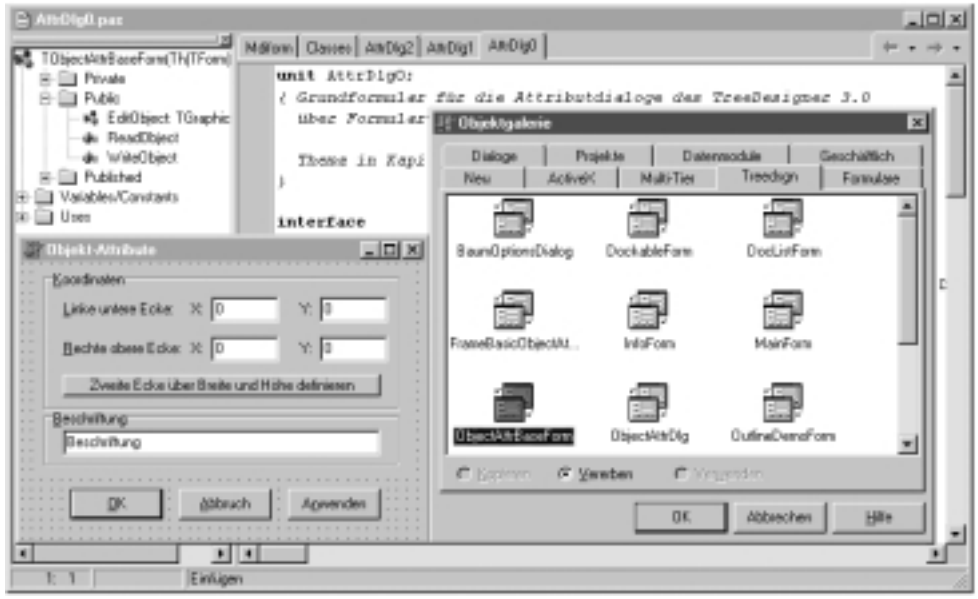


Abbildung 3.17: Eine Formularschablone mit den gemeinsamen Teilen des Dialogs; rechts das Erscheinen der Schablone in der Objektgalerie

Wenn wir auf diese Weise im Beispiel zwei »Abzüge« dieser Schablone anfertigen, kommen wir durch Hinzufügen der fehlenden Komponenten exakt zu den in Abbildung 3.16 gezeigten Dialogen.

Hinweise: Im beschriebenen Beispiel werden zur Laufzeit des Programms nur die beiden abgeleiteten Formulare als sichtbare Fenster auf dem Bildschirm benötigt. Standardmäßig würde jedoch auch ein Exemplar des Basisformulars erzeugt werden. Um diesen unnötigen Aufwand zu vermeiden, können Sie die automatische Erzeugung dieses Formulars in den Projektoptionen auf der Seite FORMULARE abschalten.

Zur Option VERERBEN und den drei anderen Optionen der Objektablage siehe Kapitel 1.6.4. Die Option KOPIEREN ist übrigens in diesem Beispiel gleichwertig mit dem erwähnten Kopieren der übereinstimmenden Komponenten über die Zwischenablage.

Formularvererbung im Entwurfsmodus

Für die Veränderung eines Formulars, das von einem anderen Formular erbt, gibt es zwei Ansatzpunkte: Sie können das Basisformular oder das erbende Formular editieren. Die Delphi-IDE kann mit beiden Situationen gut umgehen. Sie können die beiden Formulare zur Entwurfszeit nebeneinander legen, im Basisformular ein paar Komponenten verschieben und beobachten, wie Delphi automatisch für eine Anpassung der Komponenten im abgeleiteten Formular sorgt. Bei einer Änderung im abgeleiteten Formular tut sich dagegen im Basisformular nichts.

Ganz allgemein wird jede Änderung am Basisformular zur Entwurfszeit sofort in das abgeleitete Formular übertragen, sofern die geänderte Eigenschaft dort nicht *überschrieben* wurde. Das Überschreiben von geerbten Eigenschaften (damit sind sowohl Properties und Events des Formulars selbst als auch die seiner Komponenten gemeint) entspricht dem Überschreiben von Klassenelementen im OOP.

Um eine geerbte Eigenschaft zu überschreiben, passen Sie sie einfach im abgeleiteten Formular an. Vor einer Änderung einer Eigenschaft des Basisformulars prüft Delphi, ob die Eigenschaft im abgeleiteten Formular denselben Wert aufweist. Ist das der Fall, wird sie auch im abgeleiteten Formular geändert, ansonsten betrachtet Delphi die Eigenschaft als überschrieben und belässt sie im abgeleiteten Formular bei ihrem individuellen Wert.

Hinweis: Dies zeigt, wie Sie das Überschreiben einer geerbten Eigenschaft rückgängig machen können: Setzen Sie die Eigenschaft einfach wieder auf den Wert, den sie im Basisformular hat. Ein anderer Weg besteht darin, das abgeleitete Formular im Textmodus zu betrachten und alle Eigenschaften, die unverändert vom Basisformular übernommen werden sollen, zu löschen. Um einfach *alle* Einstellungen einer bestimmten Komponente auf die geerbten Werte zurückzusetzen, wählen Sie GEERBTE EINSTELLUNGEN WIEDERHERSTELLEN aus dem Kontextmenü der Komponente.

Überschreiben von Event-Handlern

Das Überschreiben von Ereignisbearbeitungsmethoden funktioniert prinzipiell genauso wie das Überschreiben von Property-Werten. Allerdings ist es im Falle der Events häufig wünschenswert, dass die geerbte Ereignisbehandlung nicht komplett ersetzt, sondern nur ergänzt wird. Delphi rechnet bereits damit und fügt in neue Ereignismethoden eines abgeleiteten Formulars automatisch die Anweisung *inherited* ein:

```
procedure TForm2.FormKeyPress(Sender: TObject; var Key: Char);
begin
  inherited;

end;
```

Inherited sorgt für den Aufruf einer eventuellen Bearbeitungsmethode im Basisformular. Es steht Ihnen jedoch frei, vor diesem Aufruf eigene Anweisungen einzufügen oder den Aufruf ganz zu löschen, wenn Sie die geerbte Ereignisbearbeitung ausschalten wollen.

In jedem Fall können Sie die geerbte Ereignisbearbeitung wiederherstellen, wenn Sie die Verknüpfung des Ereignisses im abgeleiteten Formular entfernen, also das entsprechende Ereignis-Feld im Objektinspektor löschen.

Vererbungsbeispiel im TreeDesigner

Der in Kapitel 5 vorgestellte TreeDesigner gibt ein Beispiel für die Formularvererbung, und zwar ist er die Quelle der in Abbildung 3.16 gezeigten Dialoge. Allerdings sind diese Dialoge anders realisiert, als es bisher beschrieben wurde. Die in beiden Dialogen übereinstimmenden Komponenten wurden nicht über die Formularvererbung, sondern über Frames realisiert (hierzu mehr in Kapitel 3.7.2). Das in Abbildung 3.17 gezeigte Basisformular heißt im TreeDesigner *TObjectAttrBaseForm* und ist in `AttrDlg0.pas` definiert. Die beiden in Abbildung 3.16 gezeigten Dialoge sind *TObjectAttrDlg* in `AttrDlg1.pas` und *TVerlaufObjAttrDlg* in `AttrDlg2.pas`.

Das Basisformular enthält auch die drei Schalter der Dialoge inklusive der dazu gehörenden Ereignismethoden. Da die Funktion dieser Methoden sehr eng mit der des Dialogformulars verknüpft ist und es kaum Sinn machen würde, diese drei Schalter in völlig andere (nicht von diesem Formular erbende) Formulare zu übernehmen, wurden sie auch nicht als Frame realisiert. Außerdem definiert das Basisformular noch den Rahmentyp (*BorderStyle = bsToolWindow*) und die Bildschirmposition (*Position = poMainFormCenter*) der beiden Dialoge

Hinweis: Um die Frames im TreeDesigner in Aktion zu sehen, laden Sie eine Grafikdatei in den TreeDesigner, klicken mit der rechten Maustaste auf ein Grafikelement und wählen den Menüpunkt ATTRIBUTE. Die Änderungen im Dialog werden wirksam, wenn Sie OK oder ANWENDEN drücken. Bei ANWENDEN bleibt der Dialog noch aktiv, so dass Sie weitere Änderungen vornehmen können, falls das bisherige Ergebnis Ihnen nicht gefällt.

Vererbung von Methoden

Neben der Vererbung von Komponenten und Ereignismethoden bietet es sich an, bei der Formularvererbung auch zusätzliche selbst definierte Methoden und Properties im Basisformular zu definieren, die vielleicht in allen abgeleiteten Formularen benötigt werden. Besonders flexibel wird die Vererbungsbeziehung zwischen zwei Formularen, wenn Sie virtuelle Methoden einsetzen.

Das TreeDesigner-Formular *TObjectAttrBaseForm* definiert beispielsweise ein Property, in dem das Grafikelement gespeichert wird, dessen Attribute im Dialog editiert werden:

```
property EditObject: TGraphicElement read GetEditObject write SetEditObject;
```

Hinzu kommen zwei virtuelle Methoden, die es immer dann aufruft, wenn der Inhalt der Steuerelemente auf das Grafikelement übertragen werden soll oder umgekehrt:

```
procedure TObjectAttrBaseForm.ReadObject; { virtuelle Methode }
begin // alle Steuerelemente mit den Daten von EditObject füllen
  BasicAttrFrame.CancelChanges;
end;

procedure TObjectAttrBaseForm.WriteObject; { virtuelle Methode }
begin // den Inhalt der Steuerelemente auf das EditObject anwenden
  BasicAttrFrame.ApplyChanges;
end;
```

Diese bedienen sich lediglich der Methoden *CancelChanges* und *ApplyChanges*, die in der Frame-Unit definiert und hier nicht weiter von Bedeutung sind. Interessant ist an dieser Stelle, dass die abgeleiteten Formulare diese Methoden überschreiben können. Zum Beispiel der Dialog für die Attribute eines »normalen« Grafikelements: Er enthält

ja neben dem geerbten Frame mit den Koordinaten des Grafikelements eine *RadioGroup*, mit der die Form des Elements eingestellt werden kann. In seinen Methoden *ReadObject* und *WriteObject* überträgt er deshalb die Form des Grafikobjekts (*EditObject.ShapeType*) in die *RadioGroup* (*rgGrafikForm*) und umgekehrt:

```
procedure TObjectAttrDlg.ReadObject;
begin
  inherited ReadObject;
  rgGrafikForm.ItemIndex := integer(EditObject.ShapeType);
end;

procedure TObjectAttrDlg.WriteObject;
begin
  inherited WriteObject;
  EditObject.ShapeType := GrDoc.TShapeType(rgGrafikForm.ItemIndex);
end;
```

Das Basisformular ruft nun *ReadObject* und *WriteObject* immer dann auf, wenn es erforderlich ist: *ReadObject* wird aufgerufen, wenn das Property *EditObject* neu gesetzt wird. *WriteObject* wird aufgerufen, wenn einer der Schalter OK oder ANWENDEN gedrückt wird. Dank der virtuellen Methoden kann so die Funktionalität dieser Schalter und des Properties schon in der Formular-Basisklasse definiert werden, auch wenn in den abgeleiteten Formularen zusätzliche Aktionen hinzukommen.

Hinweis: Das Beispiel zeigt auch, dass Delphi mehrstufige Vererbung von Formularen unterstützt. Wenn Sie den Frame mit den grundlegenden Objektattributen im Entwurfsmodus verändern (mehr dazu in Abschnitt 3.7.2), wirkt sich dies nicht nur auf das Basisformular mit dem Frame aus, sondern auch die beiden von diesem abgeleiteten Formulare werden sofort geändert.

Umstellung bestehender Formulare auf Vererbung *R10*

Die beschriebene Verbindung zwischen Basisformularen und abgeleiteten Formularen zur Entwurfszeit bedeutet für die Delphi-IDE einigen Verwaltungsaufwand. Ebenso könnte die Tatsache, dass man ein geerbtes Formular über einen Dialog der Objektablage erzeugt, den Anschein erwecken, dass sich die Delphi-IDE lieber nicht vom Benutzer in die Formularvererbung hineinreden lässt. Dies ist jedoch ein Irrtum, denn auch bei der Formularvererbung sind Sie nicht von der Expertenhilfe der Delphi-IDE abhängig, sondern können sie auch »von Hand« durchführen.

Dies eröffnet Ihnen die Möglichkeit, ein bestehendes Formular nachträglich von einem anderen Formular abzuleiten. Führen Sie hierfür folgende Schritte aus:

- Falls das Formular, das Sie verändern wollen, Ihnen etwas bedeutet, legen Sie zunächst eine Sicherheitskopie davon an. Aber auch sonst kann eine Sicherheitskopie es Ihnen später erleichtern, eventuelle Fehler beim Editieren des Formular-Quelltextes rückgängig zu machen.

- Ändern Sie die Deklaration der Klasse des Formulars, das von einem anderen erben soll, im Object-Pascal-Quelltext. Ändern Sie also z.B. *TDialog2 = class(TForm)* in *TDialog2 = class(TBaseDialog)*.
- Editieren Sie das Formular im Textmodus (lokales Menü: ANSICHT ALS TEXT) und ersetzen Sie das erste Schlüsselwort *object* durch *inherited*. So wird beispielsweise aus *object Dialog2: TDialog2* die Zeile *inherited Dialog2: TDialog2*.
- Wenn Sie sich in der Textansicht des Formulars befinden, können Sie auch gleich alle Properties löschen, die das Formular von der Basisklasse erben, also nicht überschreiben soll. Löschen Sie z.B. die Properties *Width* und *Height*, falls das Formular seine Größe von der des Basisformulars übernehmen soll.
- Wenn Sie für das abgeleitete Formular Ereignisse bearbeiten, die auch im Basisformular bearbeitet werden, müssen Sie die geerbten Bearbeitungsmethoden mit *inherited* aufrufen, sofern Sie nicht auf sie verzichten wollen (siehe zurückliegendes Code-Beispiel *TForm1.FormKeyPress*).
- Wenn Sie einen Teil der Fuktionalität des abgeleiteten Formulars in das neue Basisformular auslagern und so an andere abgeleitete Formulare weitergeben wollen, können Sie die entsprechenden Methoden der abgeleiteten Klassen in die Basisklasse verschieben.

Komponenten-Templates

Abschließend sei noch einmal auf die Komponenten-Schablonen (Komponenten-Vorlagen, siehe Kapitel 1.6.4) hingewiesen. Sie kommen ohne komplexe Vererbungsmechanismen aus und ließen sich auch wieder im Beispiel des TreeDesigners verwenden, können jedoch als solche nicht als Teil des TreeDesigner-Projekts auf der CD zum Buch mitgeliefert werden, da Delphi die Komponenten-Schablonen in einem eigenen Verzeichnis verwaltet.

Dem TreeDesigner-Projekt fehlt damit jedoch kein wichtiger Teil, denn die einzige sinnvolle Anwendungsmöglichkeit für eine Komponentenschablone in den gezeigten Dialogen wären die drei Schalter, und auch dann dürfte die Komponentenschablone nur die Beschriftung, andere Äußerlichkeiten und sonstige Properties dieser Schalter definieren (z.B. *ModalResult*, *Cancel* und *Default*). Die Ereignismethoden der drei Schalter sind wie erwähnt von der speziellen Arbeitsweise dieser Dialoge im TreeDesigner abhängig und wären daher in einer Komponentenschablone nicht so gut aufgehoben. Ohne die Ereignis-Methoden ließe sich eine solche Dreischalter-Schablone aber leicht in anderen Formularen und Projekten wiederverwenden.

3.7.2 Frames

Die Funktionsweise der Frames entspricht der der Formularvererbung mit dem Unterschied, dass ein Frame nicht ein ganzes Formular, sondern nur einen rechteckigen Ausschnitt daraus an ein Formular weitergibt. Ein Frame wird wie eine (möglicherweise komplexe) Komponente in ein Formular eingefügt und findet sich danach sowohl in der Klassendeklaration des Formulars als auch in der Komponentenliste des Objektinspektors wieder.

Neue Frames entwerfen

Delphi wird ohne fertige Frames ausgeliefert. Das heißt, dass Sie einen Frame nicht so einfach in ein Formular einfügen können, wie Sie ein neues Formular auf Basis der von Delphi vorgegebenen Formularschablonen erstellen können. Sie müssen vielmehr zuerst selbst einen Frame erzeugen.

Und dies funktioniert im Prinzip genauso wie der Entwurf eines ganz normalen Formulars. Einziger Unterschied ist, dass Sie den Frame über den Menüpunkt DATEI | NEUER FRAME erzeugen. Sie erhalten dadurch einen neuen leeren Frame, der zur Entwurfszeit wie ein normales Formular aussieht, sowie eine neue Object-Pascal-Unit mit der zunächst einmal leeren Frame-Klasse. Zur Laufzeit, wenn der Frame als Teil eines Formular verwendet wird, verliert er den im Entwurfsmodus noch angezeigten Rahmen. Die Frame-Klasse in der Pascal-Unit wird statt von *TForm* von *TFrame* abgeleitet. Die Klasse *TFrame* ist gewissermaßen eine Großcousine von *TForm*, denn beide stammen von der Klasse *TScrollingWinControl* ab – *TForm* über die Zwischenklasse *TCustomForm*, *TFrame* entsprechend über *TCustomFrame*.

Frames verwenden

Der Entwurf, das Schreiben von Ereignismethoden und das Speichern der Frame-Dateien läuft so ab wie bei Formularen. Neuigkeiten gibt es erst, wenn Sie einen bestehenden Frame verwenden wollen. Dazu wählen Sie aus der Komponentenpalette den Schalter mit dem Hinweistext FRAMES (erster Schalter auf der Standard-Seite der Palette) und zeichnen den Umriss des Frames in das Formular ein (dieses Formular wird im Folgenden auch als das *Besitzer-Formular* bezeichnet). Der Frames-Schalter ist sozusagen eine Pseudokomponente, denn erst nach dem Einzeichnen des Umrisses wählen Sie die Klasse des Frames aus, den Sie einfügen wollen. Hierzu zeigt Ihnen Delphi automatisch eine Liste aller zur Verfügung stehenden Frames an.

Wenn Sie den Umriss des Frames zu klein wählen, wird der Frame soweit notwendig mit einem oder zwei Scrollbars ausgestattet, über die Sie die unsichtbaren Bereiche erreichen können (wobei man normalerweise wohl eher geneigt ist, den Umriss nachträglich so einzustellen, dass die Scrollbars wieder verschwinden).

Vererbungsmechanismen

Es gibt einen wesentlichen Unterschied zwischen der Benutzung eines Frames und der einer Komponente, die andere Komponenten enthält. Er liegt darin, dass Sie im Formular noch auf die einzelnen Komponenten des Frames zugreifen können, während Sie die Komponente im Formulareditor nur als ganzes Objekt wählen können.

Dabei greifen dieselben, schon im Zusammenhang mit der Formularvererbung in Kapitel 3.7.1 beschriebenen Mechanismen, die im Folgenden noch einmal zusammengefasst sind. Auch im Zusammenhang mit den Frames können Sie:

- die Properties im Objektinspektor ändern (überschreiben), insbesondere die Position und Größe der Komponenten, die Sie natürlich mit der Maus wie gewohnt einstellen können.
- die Ereignisse der Komponenten bearbeiten. Delphi erzeugt automatisch Ereignismethoden mit einem Aufruf von *inherited*, also einem Aufruf der geerbten Ereignisbearbeitung, den Sie nach Wunsch durch Löschen unterbinden können.
- den Frame selbst verändern. Änderungen wirken sich sofort auf alle Formulare, in denen der Frame enthalten ist, aus, sofern sie dort nicht überschrieben wurden.
- das Überschreiben von Properties rückgängig machen, indem Sie den Wert im Objektinspektor auf den geerbten Wert zurücksetzen oder die entsprechende Angabe aus der Textdarstellung des Frames löschen.
- mehrstufige Vererbung verwenden, also z.B. Frames in Frames oder Frames in abgeleiteten Formularen.

Schließlich sei noch darauf hingewiesen, dass auch ein Frame selbst (wie ein Formular) über eigene Properties und Ereignisse verfügt und dass Sie Frames wie Formulare in Delphis Objektablage speichern können.

Vorteile gegenüber der Formularvererbung

Richtig eingesetzt wirkt sich die Verwendung von Frames positiv auf die Struktur einer Anwendung aus. Die folgenden Vorteile sind beispielsweise durch die Formularvererbung nicht oder nur schwer zu erreichen:

- Es können mehrere Frames in ein Formular eingefügt werden, aber ein Formular kann immer nur von einem Formular erben (oder man muss mehrstufige Vererbung verwenden).
- Frames erlauben es, den Code für verschiedene Formularbereiche zu trennen und in verschiedenen Units zu verwalten.

- Durch Frames lassen sich alternative Implementierungen von Formularbereichen leicht austauschen. Sie benötigen dazu lediglich mehrere Frames mit der gleichen Schnittstelle (z.B. denselben Steuerelementen für den Benutzer oder denselben Properties und Methoden, die vom Besitzer-Formular aufgerufen werden können) und können dann in einem Formular, das eines dieser Frames verwendet, ein Frame mit einer anderen Implementierung einsetzen, ohne sonst noch etwas am Formular ändern zu müssen.
- Die Namen der im Frame enthaltenen Komponenten haben einen eigenen Namensraum (den der Frame-Klasse) und können daher nicht mit den Namen der Komponenten des Formulars in Konflikt geraten.

Auch gegenüber den anderen erwähnten Techniken zur Zusammenfassung mehrerer Komponenten weisen Frames Vorteile auf:

- Gegenüber einer Komponente, die mehrere Komponenten zusammenfasst, gibt es bei den Frames keine Kapselung der im Frame enthaltenen Komponenten. Was aus Sicht des OOP als Nachteil angesehen werden kann, erweist sich in der Praxis als Steigerung der Flexibilität, denn Sie können alle im Frame enthaltenen Komponenten auch im Besitzer-Formular des Frame noch ändern.
- Komponenten-Schablonen bieten zwar dieselbe Flexibilität wie die Frames, was die nachträgliche Veränderung der Einzelkomponenten im Formular angeht, ihnen fehlt aber die den Einzelkomponenten übergeordnete Klasse und die der Strukturierung dienende Unit.

Da Frames so etwas Ähnliches wie eine Verallgemeinerung der Formularvererbung darstellen, kann man lange erfolglos nach Nachteilen gegenüber dieser Vererbung suchen. Im Wesentlichen bestimmt das Einsatzgebiet, ob man die Formularvererbung braucht, denn dieser bleibt noch der wichtige Einsatzzweck vorbehalten, die Grundeinstellungen von Formularen zu definieren und zu vererben. Alle Property-Einstellungen und Ereignismethoden, die dem Formular selbst gehören, lassen sich eben nicht in einem Frame festhalten, sondern nur über eine Formularschablone. In bestimmten Fällen lassen sich ja beide Techniken auch kombinieren: Das leere Formular lässt sich, vielleicht mit einigen grundlegenden Aktionsschaltern, von einer Formularschablone ableiten. Im Formular enthaltene Komponentengruppen, die auch in anderen Formularen vorkommen, können dann über Frames realisiert werden (siehe Beispiel des TreeDesigners).

Frames im TreeDesigner

Der in Abbildung 3.18 gezeigte Frame dient im TreeDesigner als Grundlage für den in Kapitel 3.7.1 beschriebenen Basisdialog zur Einstellungen von Grafikelement-Attributen. Er enthält ein Eingabefeld für die Beschriftung des Grafikelements sowie vier Fel-

der für die Koordinaten des umgebenden Rechtecks. Außerdem enthält er noch einen Schalter, mit dem die Darstellung der Koordinaten zwischen *(x1/y1, x2/y2)* und *(x1/y1, Breite/Höhe)* gewechselt werden kann. Damit gibt dieser Schalter dem Frame einen guten Anlass, zu demonstrieren, dass ein Frame nicht nur aus Komponenten, sondern auch aus Programmlogik bestehen kann, denn die Eingaben in die Felder x2 / y2 sollen ja durch das Programm in entsprechende Breiten- und Höhenangaben umgewandelt werden.

Abbildung 3.18: Der Frame für die Dialoge aus den Bildern 3.15 und 3.16 zur Entwurfszeit

Ändert der Benutzer beispielsweise den Wert *X2* (RECHTE OBERE ECKE X), wird die folgende Methode ausgeführt:

```
procedure TFrameBasicObjectAttrs.EditX2Change(Sender: TObject);
begin
  EditBreite.Number := EditX2.Number - EditX1.Number;
  { Die Breite des Objekts versteht sich inklusive der
    X1-Koordinate, aber exklusiv der X2-Koordinate }
end;
```

Außerdem enthält der Frame noch die bereits in Kapitel 3.7.1 verwendeten Methoden *ApplyChanges* und *CancelChanges*. Das folgende Listing der Frame-Klasse ist lediglich um die Variablen für die vielen Komponenten gekürzt:

```
type
 TFrameBasicObjectAttrs = class(TFrame)
   GroupBox1: TGroupBox;
   ...
   procedure SwitchRelAbsClick(Sender: TObject);
   procedure EditX2Change(Sender: TObject);
   procedure EditY1Change(Sender: TObject);
   procedure EditBreiteChange(Sender: TObject);
   procedure EditHoeheChange(Sender: TObject);
 private
   FEditObject: TGraphicElement;
   procedure SetEditObject(const Value: TGraphicElement);
 public
```

```
    property EditObject: TGraphicElement read FEditObject
                         write SetEditObject;
    procedure ApplyChanges;
    procedure CancelChanges;
  end;
```

TFrameBasicObjectAttrs stellt sich also im Prinzip als eine ganz normale Object-Pascal-Klasse dar, deren Methoden im Besitzer-Formular aufgerufen werden können. Aufrufe von *ApplyChanges* und *CancelChanges* waren ja bereits in früheren Code-Auszügen zu sehen. Daher soll nun die Methode *ApplyChanges* das TreeDesigner-Beispiel für dieses Kapitel abschließen. Sie schreibt die Daten der Steuerelemente in das bearbeitete Grafikelement (Property *EditObject*). Das Besitzer-Formular des Frames ruft sie als Reaktion auf den Schalter ANWENDEN auf.

```
procedure TFrameBasicObjectAttrs.ApplyChanges;
var
  R: TRect;
begin
  if Assigned(FEditObject) then begin
    R.Left := EditX1.Number;
    R.Top := EditY1.Number;
    R.Right := EditX2.Number;
    R.Bottom := EditY2.Number;
    FEditObject.Text := Beschriftung.Text;
    FEditObject.SetNewRect(R);
  end;
end;
```

4 Außerhalb der Komponenten

Dieses Kapitel setzt die in Kapitel 3 begonnene Behandlung der VCL fort, beschäftigt sich aber nicht mehr in erster Linie mit den visuellen Komponenten, sondern mit anderen wichtigen Klassen sowie mit weiteren Themen, die mit der VCL zusammenhängen:

- Abschnitt 4.1 stellt die grundlegenden Datenstrukturen *TList* und *TStrings* vor, auf denen viele Properties von VCL-Klassen basieren. Dabei wird mit *THistoryList* eine neue Klasse von *TStringList* abgeleitet, die später zum Anhängen von Dateilisten an Menüs (Kapitel 4.6.2) und zur Implementierung der visuellen *THistoryCombo*-Komponente verwendet wird (Kapitel 6.5.3).
- Abschnitt 4.2 beschreibt den Aufbau der Windows-Registrierung und der INI-Dateien, die Verwendung der zugehörigen VCL-Klassen (*TRegistry*, *TIniFile*, *TMemIniFile* und *TRegistryIniFile*) sowie die Verwendung der Registry zum Verknüpfen von Dateitypen mit Anwendungen.
- In Abschnitt 4.3 geht es um VCL-Klassen, die bei der Speicherung von beliebigen Daten in Dateien behilflich sind: *TStream*, *TReader* und *TWriter*. Weitere Themen sind die Speicherung polymorpher Objekte, für die einige Hilfsfunktionen der VCL benötigt werden, sowie die nicht in Dateien, sondern im Hauptspeicher liegenden Memory-Streams.
- Zentrale Klasse für die in Abschnitt 4.4 behandelte Grafikausgabe ist *TCanvas*, die eine Zeichenfläche repräsentiert und zahlreiche Methoden für die Grafikausgabe bereitstellt. Auch Zeichenwerkzeuge wie *TPen* und *TBrush* und besitzergezeichnete Komponenten werden in diesem Abschnitt erläutert.
- Abschnitt 4.5 behandelt die von *TGraphic* abgeleiteten Klassen *TBitmap*, *TIcon* und *TMetafile*. Objekte dieser Klassen enthalten bereits eine komplette Grafik und können sich selbst auf eine *TCanvas*-Zeichenfläche ausgeben.
- Abschnitt 4.6 zeigt, dass Sie die Flexibilität der Menüklassen der VCL, die Ihnen der Menü-Designer zur Entwurfszeit bereitstellt, zur Laufzeit an den Benutzer weitergeben können. Auch die eng mit Menüs in Verbindung stehenden Aktionslisten werden erläutert. Außerdem stellt dieser Abschnitt Ihnen eine Komponente vor,

durch deren Einbindung Sie jeder Anwendung mit Hauptmenü oder Aktionsliste zu einem Tastenkürzeleditor verhelfen können, der eine geänderte Tastaturbelegung auch zwischen den Programmläufen speichern kann.

- In Abschnitt 4.7 geht es um die allgemeinen Multitasking-/Multithreading-Grundlagen sowie um die Klasse *TThread*, mit deren Hilfe Sie mehrere Programmteile innerhalb einer Anwendung quasi gleichzeitig ablaufen lassen können.

4.1 Grundlegende Datenstrukturen: TList und TStrings

TList und *TStrings* sind die wichtigsten Klassen der VCL, die sich mit der Datenverwaltung außerhalb von Datenbanken befassen. Besonders *TStrings* wird von den Komponenten der VCL häufig eingesetzt, z.B. in den Properties *TListBox.Items*, *TMemo.Lines* und *TScreen.Fonts*.

4.1.1 TStrings

Aufgabe der Klasse *TStrings* ist es, eine Liste von Strings zu verwalten, wobei Sie jeden String optional mit einem beliebigen Objekt verknüpfen können.

TStrings und TStringList

Die Klasse *TStrings* ist abstrakt und beschreibt nur die Schnittstelle, über die Sie in abgeleiteten Klassen auf die Strings zugreifen. Diese Schnittstelle besteht aus einigen Properties und einer Reihe von Methoden, die in *TStrings* teilweise abstrakt sind. Da die Verwaltung der Strings in dieser Klasse noch nicht implementiert ist, stellt *TStrings* nicht unbedingt eine String*liste*, sondern lediglich eine Sammlung von Strings dar, in der jeder String über einen Index ansprechbar ist.

Die wichtigste von *TStrings* abgeleitete Klasse ist *TStringList*. Sie überschreibt die abstrakten Methoden von *TStrings* und implementiert die Verwaltung der Strings unter Zuhilfenahme der Klasse *TList*. Zu ihren besonderen Fähigkeiten gehören das Abfangen von doppelten Strings (Property *Duplicates*), das automatische Sortieren der Liste (Property *Sorted*) und zwei Ereignisse, mit denen Sie sich über Änderungen informieren lassen können (*OnChange* und *OnChanging*). Wir beschränken uns in diesem Kapitel auf die allgemeinen Fähigkeiten, die in der Klasse *TStrings* definiert sind.

Einige Komponenten der VCL, wie z.B. *TListBox*, verwenden intern andere von *TStrings* abgeleitete Klassen, geben diese nach außen hin jedoch nicht als solche zu erkennen, sondern stellen sie über ein Property des Typs *TStrings* zur Verfügung.

TStrings-Properties

Während *TStrings* über 20 Methoden besitzt, kommt sie mit sehr wenigen Properties aus, die Zugriff auf die Strings sowie die damit verbundenen Objekte gewähren und die Anzahl der Strings angeben (alle Array-Properties beginnen beim Index 0):

Property	**Typ**	**Beschreibung**
Count	Integer	Anzahl der gespeicherten Strings
Objects[Integer]	TObject-Array	mit den Strings assoziierte Objekte
Values[String]	String-Array	Wenn die Stringliste Strings der Form »Schlüssel=Value« enthält, finden Sie mit diesem Property den *Value* zu einem Schlüssel, wenn Sie diesen Schlüssel als Index für *Value*s verwenden. Ein Beispiel finden Sie in Kapitel 4.2.1 unter *Lesen von kompletten Sektionen*.
Strings[Integer]	String-Array	gibt Zugriff auf die einzelnen Strings und ist das *default*-Property der Klasse.
CaseSensitive	Boolean	(neu in Delphi 6) lässt Ihnen die Wahl, ob bei Such-, Sortier und Vergleichsoperationen zwischen Groß- und Kleinschreibung unterschieden werden soll (bis Delphi 5 ist das standardmäßig der Fall).
DelimitedText	String	(neu in Delphi 6) fasst alle Strings zu einem einzigen String zusammen, aus dem sich die einzelnen Strings eindeutig wiederherstellen lassen (z.B. wenn Sie den zusammengefügten String später wieder an das *DelimitedText*-Property zuweisen). Dazu werden die einzelnen Strings mit einem Trennzeichen getrennt und, falls sie Leerzeichen enthalten, mit Anführungszeichen versehen. Das Trennzeichen wird durch *Delimiter*, das Anführungszeichen durch *QuoteChar* vorgegeben.
CommaText	String	wie *DelimitedText*, nur mit festeingestelltem Komma als Trennzeichen (Anführungszeichen werden weiter durch *QuoteChar* festgelegt); Beispielanwendung in Kapitel 1.9.3: Speicherung des Inhalts einer Listbox in der Registry.
QuoteChar	Char	(neu in Delphi 6) Anführungszeichen, das in *DelimitedText* und *CommaText* zum Einschließen von Strings verwendet wird. Enthält der String selbst bereits ein *QuoteChar*, so wird die in *DelimitedText* und *CommaText* verdoppelt (ähnlich der Hochkommatas in einem Object-Pascal-String).
Delimiter	Char	(neu in Delphi 6) Trennzeichen für das *DelimitedText*-Property. Standardmäßig ist dies ein Komma, bei einer Liste von Pfaden bietet sich dagegen die Systemkonstante *PathDelimiter* als Alternative an.
CommaText	String	liefert die Strings in einem speziellen »Systemdaten-Format«, welches
Names[Integer]	String-Array	liefert passend zu *Values[i]* den Namen des Schlüssels (also den Stringteil vor dem Zeichen »=« bzw. einen Leerstring, falls kein »=« vorhanden ist).

Property	Typ	Beschreibung
Text	String	gibt alle Strings in einem einzigen String zurück, wobei die Einzel-Strings durch die in Textdateien verwendete Zeilenende-Markierung (*#13#10*) getrennt werden. Kann auch beschrieben werden, um alle Strings auf einmal neu zu setzen.

Da *Strings* das Standard-Property ist, können Sie auf einen String sowohl mit *StringListe.Strings[StringIndex]* als auch mit *StringListe[StringIndex]* zugreifen. Unter Zuhilfenahme von *Count* können Sie auf einfache Weise alle Strings der Liste durchlaufen, z.B. wie folgt:

```
for i := 0 to StringListe.Count-1 do StringListe[i] := ...
```

Die VCL definiert viele Properties und Klassen, die ähnlich zu bedienen sind wie *TStrings*, obwohl es sich nicht direkt um ein *TStrings*-Objekt handelt. So können Sie beispielsweise die Untermenüpunkte eines Menüpunkts (*TMenuItem*) mit *MenuePunkt[i]* ansprechen, *MenuePunkt.Count* liefert die Zahl der Menüpunkte. Intern ist *TMenuItem* keine Stringliste, sondern verwendet ein *TList*-Objekt zur Speicherung der Untermenüpunkte.

Assoziierte Objekte *R122*

Oft sind die Strings einer Liste mit anderen Objekten des Programms verknüpft. So könnten Sie in einer Listbox beispielsweise die gerade in Ihrer Anwendung geöffneten Fenster anhand ihrer Titelbezeichnungen aufzählen. Der Benutzer könnte eines dieser Fenster auswählen, um es in den Vordergrund zu holen. Sie würden dann den Titel des ausgewählten Fensters im Listbox-Property *Items.Strings[ItemIndex]* finden. Um jedoch das zugehörige Fenster zu finden, müssten Sie nun alle vorhandenen Fenster nach diesem Titel durchsuchen.

Anders verhält es sich in diesem Beispiel, wenn Sie die zugehörigen Fenster gleich mit in der *TListBox*-Komponente speichern. Wenn Sie die Fenstertitel nicht wie folgt einfügen ...

```
Listbox.Items.Add(Form.Caption);
```

... sondern mit dieser Zeile:

```
Listbox.Items.AddObject(Form.Caption, Form);
```

... speichern Sie zusammen mit dem Fenstertitel, der in der ListBox erscheint, eine Referenz auf das Formular (*Form*). *Form* wird von *AddObject* im *Objects*-Array der Stringliste abgelegt, und zwar unter demselben Index, den der im ersten Parameter angegebene String im *Items*-Array erhält.

Um in der beschriebenen Beispielsituation das Formular festzustellen, das zum gerade gewählten Listeneintrag gehört, könnten Sie nun den Ausdruck *TForm(ListBox.Items.Objects[Listbox.ItemIndex])* verwenden. Da *Objects* das Formular nicht als Typ *TForm*, sondern als *TObject* zurückgibt, ist hier eine explizite Typenumwandlung erforderlich.

Sie können auch nachträglich Objekte mit den Strings verknüpfen, indem Sie sie selbst in das *Objects*-Array schreiben, beispielsweise mit

```
with Listbox.Items do
  Objects[IndexOf(Form.Caption)] := Form;
```

... wobei die Methode *IndexOf* und weitere nützliche Methoden von *TStrings* in der nächsten Tabelle erläutert werden.

Wenn Sie auf diese Weise Objekte mit den Strings verbinden, behalten Sie weiter die volle Kontrolle über diese Objekte: *TStrings* verändert diese in keiner Weise. Wenn Sie den zugehörigen String löschen, gibt *TString* das verknüpfte Objekt nicht frei, sondern entfernt lediglich die Referenz zu ihm aus seinem *Objects*-Array.

Ein Praxisbeispiel zum Ausprobieren finden Sie im Projekt *Wecker3d* von Kapitel 1.9.4.

Weitere TStrings-Methoden

Durch die Anwendung von einfachen Methoden können Sie noch viele weitere Aufgaben mit einem *TStrings*-Objekt durchführen. Die Parameter dieser Methoden zeigen und erklären sich zum größten Teil bereits durch die *Code-Parameter*-Funktion des Delphi-Editors, daher sind sie im Folgenden nicht extra erwähnt. Zuerst einmal gibt es natürlich grundlegende Funktionen, die man meistens von einer Liste erwartet:

- Einzelne Strings der Liste löschen Sie mit *Delete*, vertauschen Sie mit *Exchange* und verschieben Sie mit *Move*.
- Um alle Strings zu löschen, rufen Sie *Clear* auf.
- Um Strings an einer bestimmten Position einzufügen, verwenden Sie statt *Add* und *AddObject* die Methoden *Insert* und *InsertObject*.
- Zum Suchen von Einträgen dienen die Methoden *IndexOf* (sucht anhand des String-Textes), *IndexOfObject* (sucht anhand des assoziierten Objekts) und *IndexOfName* (sucht in Strings, die nach dem Muster »Name = Wert« aufgebaut sind, siehe Property *Names*).

Des Weiteren verfügt *TStrings* über eine Reihe schöner »fortgeschrittener« Funktionen:

- Wenn Sie gleich zwei *TStrings*-Kollektionen haben, können Sie sie mit *Equals* vergleichen, mit *AddStrings* aneinander hängen oder mit *Assign* abgleichen.

- Wenn der Inhalt des *TStrings*-Objekts in einer visuellen Komponente angezeigt wird, können Sie am Anfang einer längeren Aktualisierungsoperation (beispielsweise Aufrufe von *Add* und *Delete*) die Methode *BeginUpdate* aufrufen, damit nicht alle Änderungen am Bildschirm langatmig nachvollzogen werden. Wenn die Stringliste fertig ist, erreichen Sie mit *EndUpdate*, dass die visuelle Komponente einmal und von nun an wieder regelmäßig aktualisiert wird.
- Dateioperationen: Mit *LoadFromFile* laden Sie den Inhalt einer Textdatei in das *TStrings*-Objekt. Dabei wird jede Zeile der Datei zu einem String, und wenn Sie den Inhalt des *TStrings*-Objekts nachher über das Property *Text* abfragen, erhalten Sie wieder genau den Inhalt der Textdatei. Entsprechend speichern Sie die Stringliste mit der Methode *SaveToFile*. Diese Methoden sind von weit reichender Bedeutung, da mit ihnen der Inhalt der verschiedensten Komponenten gespeichert werden kann, wie etwa *TListBox.Items* und *TMemo.Lines*.
- Alternativ zu den letztgenannten Methoden können Sie auch *LoadFromStream* und *SaveToStream* verwenden, wenn Sie die Stringliste als Teil eines Streams speichern wollen, in dem sich noch andere Daten befinden.

Hinweis: Wenn Sie Stringlisten mit sehr vielen Strings verwalten, könnte sich die Verwendung der neuen Klasse *THashedStringList* aus der Unit *IniFiles* als nützlich erweisen. Diese Klasse verwendet intern eine Hash-Tabelle zur Beschleunigung der *IndexOf* und *IndexOfName*-Funktionen. Allerdings muss diese Tabelle vor jeder Suche neu aufgebaut werden, falls die Stringliste seit der letzten Suche verändert wurde. Dies dauert natürlich noch länger als eine Suche in einer normalen Stringliste. Daher lohnt sich der Einsatz von *THashedStringList* besonders bei Stringlisten, die einmal initialisiert und danach nur noch durchsucht, aber nicht mehr verändert werden.

4.1.2 Ableiten einer History-Liste von TStringList

Ein kleines Anwendungsbeispiel gibt die Klasse *THistoryList*, die einerseits in Kapitel 6.5.3 bei der Programmierung der visuellen *THistoryCombo*-Komponente verwendet, andererseits in Kapitel 4.6.2 um Menüfunktionen erweitert wird. Der Kern dieser Klasse wird an dieser Stelle beschrieben.

History-Listen *R62*

THistoryList ist von *TStringList* abgeleitet und wird damit zu einer spezialisierten Stringlisten-Klasse. Die Spezialisierung liegt darin, dass *THistoryList*

- eine Längenbegrenzung für die Liste einführt,

- diese Längenbegrenzung bei Hinzufügen von Strings mit der Methode *AddString* überprüft und gegebenenfalls den ältesten String der Liste entfernt
- und dass die Methode *AddString* einen String aus der Mitte der Liste entfernt, wenn er am Anfang erneut eingefügt wird.

Viele Dialogboxen der Delphi-IDE geben Beispiele für History-Listen, die sich genauso verhalten, beispielsweise die aufklappbare Liste im *Suchen*-Dialog: Eine neue Eingabe verschiebt die alten Eingaben eine Listenposition nach unten, die Auswahl einer Eingabe aus der Liste fügt diese Eingabe wieder an den Anfang der Liste, und wenn die Liste eine gewisse Größe erreicht, fallen die letzten Strings am Ende heraus.

Die vereinfachte Deklaration und der Konstruktor von *THistoryList* sehen wie folgt aus:

```
  THistoryList = class(TStringList)
  (* Die THistoryList-Funktionen, die mit Menüs zusammenhängen,
     werden an anderer Stelle beschrieben. *)
  private
    FMaxLen: Integer;
    FUseRegistry: Boolean;
    procedure SetMaxLen(AnInt: Integer);
  public
    constructor Create;
    property MaxLen: Integer read FMaxLen write SetMaxLen;

    procedure AddString(s: string);
    procedure SaveToIni(IniName, IniSection: string);
    procedure LoadFromIni(IniName, IniSection: string);
    property UseRegistry: Boolean read FUseRegistry write FUseRegistry;
  end;

constructor THistoryList.Create;
begin
  inherited Create;
  FMaxLen := 10;
end;
```

Die Methode *AddString* verwendet die von *TStringList* geerbten Elemente *IndexOf*, *Delete*, *Insert* und *Count*, um die oben beschriebene Funktionalität zu gewährleisten:

```
procedure THistoryList.AddString(s: string);
var
  OldIndex: Integer;
begin
  { String schon vorhanden? }
  OldIndex := IndexOf(s);
  if OldIndex <> -1 then
    Delete(OldIndex); { dann das Duplikat löschen }
  Insert(0, s); { auf alle Fälle an erster Stelle einfügen }
```

```
  { Maximallänge überschritten? }
  if Count > FMaxLen then
    Delete(Count-1); { letzten Eintrag löschen }
  { Menü-Methoden, siehe Kapitel 4.6.2: }
  RemoveMenuItems; { Menü in Ursprungszustand versetzen... }
  AddMenuItems; { ...und alle Einträge neu anhängen. }
end;
```

Da das Property *MaxLen* jederzeit änderbar ist, muss die zugehörige Schreibmethode bei einer solchen Änderung gegebenenfalls die Länge der Liste anpassen:

```
procedure THistoryList.SetMaxLen(AInt: Integer);
begin
  { nur Maximallängen zwischen 1 und 100 akzeptieren: }
  if (AInt >= 1) and (AInt <= 100) then begin
    FMaxLen := AInt;
    while Count > FMaxLen do
      Delete(Count-1); { alle überzähligen Einträge löschen }
  end;
  ... Menü-Aktualisierung ...
end;
```

Zum Speichern und Laden der Liste bedient sich *THistoryList* der Klassen *TIniFile* und *TRegistryIniFile*, daher werden die Methoden *SaveToIni* und *LoadFromIni* sowie das Property *UseRegistry* erst in Kapitel 4.2.4 erläutert.

4.1.3 TList

TList ist direkt von *TObject* abgeleitet; und da sie eine wichtige Grundlage für die interne Implementierung der Klasse *TStringList* liefert, verwundert es nicht, dass ihre Methoden und Properties denen der Stringlisten ähneln.

Wichtigster Unterschied zu *TStrings* / *TStringList* ist der Typ der Daten, die *TList* speichert: *TList* erwartet nämlich *keinen* Typ, sondern lediglich einen untypisierten Zeiger (siehe *Pointer* in Kapitel 2.4.6) auf ein beliebiges Objekt, so dass Sie nicht nur Objekte, sondern auch einfache Daten wie Strings in einer *TList*-Liste speichern können. Zahlen können Sie sogar direkt darin speichern, indem Sie sie als *LongInt* ansehen und dann diesen *LongInt* in einen *Pointer* umwandeln, den Sie dem *TList*-Objekt übergeben (dieser *Pointer* ist dann als Zeiger natürlich ungültig).

Wenn Sie ein Objekt wieder aus der Liste auslesen, müssen Sie den Typ *Pointer* wieder in den ursprünglichen Typ zurückverwandeln, und zwar funktioniert hier nur die normale Typenumwandlung (Kapitel 2.4.7), nicht aber die Umwandlung mit dem *as*-Operator (Kapitel 2.3.4). Ein Beispiel dafür gibt die von *TList* abgeleitete Klasse *TGraphicDoc* in Kapitel 5.3.2.

Properties und Methoden

Die Elemente der Liste können Sie über das Property *Items* ansprechen, das bis auf den Typ der Elemente wie das *Strings*-Property von *TStrings* funktioniert. Entsprechend gibt es auch hier ein Property *Count* für die Anzahl der gespeicherten Elemente.

Die meisten Methoden heißen so wie entsprechende Methoden von *TStrings* und arbeiten auch so, deshalb seien sie hier nur aufgezählt: *Add*, *Clear*, *Delete*, *Exchange*, *IndexOf*, *Insert*, *Move* und *Remove* (die Auswahl der Methoden ist natürlich deutlich kleiner als bei *TStrings*).

Die zusätzlichen Methoden *First*, *Last*, *Remove* und *Expand* werden Sie wahrscheinlich selten benötigen, daher sei an dieser Stelle auf die Online-Hilfe verwiesen.

Interne Funktionsweise

Auch die Properties *List* und *Capacity* sowie die Methode *Pack* brauchen Sie wahrscheinlich nicht zu verwenden, jedoch geben diese Aufschluss über die interne Funktionsweise von *TList*. Anders als der Name vermuten lassen könnte, sind die Daten in *TList* nämlich nicht in einer per Zeiger verknüpften Liste, sondern in einem *Pointer*-Array verknüpft.

Wie viele Zeiger dieses Array fassen kann, gibt *Capacity* an. Das Array selbst finden Sie im Property *List*. Falls es vollständig gefüllt sein sollte, wird es beim Hinzufügen eines neuen Elements mit *Add* automatisch erweitert.

In Kapitel 5.3.2 finden Sie ein Beispiel für die Verwendung von *TList* zum Ableiten einer neuen Klasse (der schon erwähnten Klasse *TGraphicDoc*).

4.1.4 Andere Container-Klassen

Während die bisher besprochenen Listenklassen in der zentralen Unit *Classes* definiert werden, gibt es noch einige weitere Listenklassen, die in der vergleichsweise winzigen Unit *Contnrs* untergebracht sind (ab Delphi 5). *Contnrs* stellt einige von *TList* abgeleitete Klassen bereit, die auf spezielle Objekte oder auf kleine Zusatzfunktionen spezialisiert sind. Dieses Kapitel soll nur eine kurze Übersicht geben, da die Klassen unter Nutzung der Online-Hilfe sehr einfach zu verwenden sind.

Während die Elemente von *TList* einfache *pointer* sind, sind die Container-Klassen für konkretere Elemente bzw. Objekte ausgelegt. Diese Objekte könnten Sie zwar auch direkt in einer *TList* speichern, bei der Abfrage eines Listenelements müssten Sie jedoch immer den abstrakten *pointer* in den tatsächlichen Typ umwandeln (z.B. mit *TComponent(List.Items[i])*.

- *TObjectList* ist direkt von *TList* abgeleitet und speichert Instanzen von *TObject* oder davon abgeleiteten Klassen. Sie führt ein neues Property namens *OwnsObjects* ein. Indem Sie dieses auf *True* setzen (was bereits die Voreinstellung ist), weisen Sie die *ObjectList* an, die Objekte automatisch mit *Free* freizugeben, wenn sie aus der Liste gelöscht werden.
- *TComponentList* und *TClassList* verrichten Ähnliches für Listenelemente der Typen *TComponent* und *TClass*.

Einen anderen Vererbungsweg geht *TOrderedList*. Sie ist eine abstrakte Klasse zur Definition von geordneten Listen, auf die mit *Push*, *Pop* und *Peek* zugegriffen werden kann. Methoden wie *Add* und *Delete* sind in dieser abstrakten Definition nicht vorgesehen. *TOrderedList* ist auch nicht von *TList*, sondern von *TObject* abgeleitet und verwendet ein *TList*-Objekt nur intern. Spezielle Ableitungen von *TOrderedList* sind *TStack/TObjectStack* und *TQueue / TObjectQueue*, für die hiermit auf die Online-Hilfe verwiesen wird.

4.2 INI-Dateien und die Windows-Registry

Ein wichtiges Merkmal vieler benutzerfreundlicher Anwendungen ist die Fähigkeit, Einstellungen des Benutzers in Initialisierungsdateien zu speichern, so dass sich der Benutzer nach dem nächsten Programmstart in derselben Umgebung befindet wie vor dem letzten Verlassen des Programms. Zum Speichern solcher Einstellungen gibt es unter Windows zwei Standards: INI-Dateien und die Registrierungsdatenbank (kurz »Registry«), wobei letztere auch intensiv zur Speicherung von Systeminformationen wie z.B. zur Registrierung von COM-Klassen und allen Arten von ActiveX-Servern dient.

4.2.1 Dateien im INI-Format

Seit der Einführung von Windows 95 wird meistens nur noch die Registry verwendet, allerdings liefert Delphi 6 selbst ein Beispiel, wie die INI-Funktionen auch in den 2001 aktuellen Windows-Versionen noch sinnvoll eingesetzt werden können, wenn auch nicht mit Dateien, die die Endung `.ini` tragen: Delphi speichert sowohl die Projektoptionen als auch den zu einem Projekt gehörenden Aufbau des Desktops in Dateien des INI-Formats: Die Dateien für die Projektoptionen tragen den Namen des jeweiligen Projekts mit der Endung `.dof` und der Aufbau des Desktops findet sich in Dateien mit der Endung `.dsk` wieder (sofern Sie die Umgebungsoptionen entsprechend eingestellt haben – Seite *Vorgaben*, Gruppe *Optionen für Autospeichern*). Beide Dateien können Sie in einem normalen Texteditor einsehen und eventuell sogar sinnvoll verändern.

Die VCL erleichtert nicht nur den Zugriff sowohl auf Registry- als auch auf INI-Dateien durch die Klassen *TIniFile* und *TRegistry*, sondern sie stellt auch eine Klasse zur Verfügung, die kompatibel mit *TIniFile* ist, aber in die Registry schreibt. Mit Hilfe dieser Klasse können Sie Programme, die mit INI-Dateien arbeiten, sehr leicht auf die Verwendung der Registry umstellen, indem Sie einfach die Verwendung von *TIniFile* im Programmcode durch *TRegistryIniFile* ersetzen (ab Delphi 4; in vorherigen Versionen gibt es mit *TRegIniFile* eine ähnliche Klasse).

TRegistryIniFile hat mit *TIniFile* einen gemeinsamen Vorfahren: die ebenfalls neu eingeführte Basisklasse *TCustomIniFile*. Hierdurch ist es also möglich, polymorphe *TCustomIniFile*-Objekte zu deklarieren, denen Sie wahlweise ein *TIniFile*-Objekt oder ein *TRegistryIniFile*-Objekt zuweisen können, ohne den Programmcode für das polymorphe Objekt ändern zu müssen (die ältere Klasse *TRegIniFile* ist übrigens nicht zuweisungskompatibel mit *TCustomIniFile*).

Aufbau der INI-Dateien

INI-Dateien sind in mehrere *Sektionen* gegliedert, die jeweils aus Zeilen der Form »Schlüssel=Wert« bestehen. Die von *THistoryList* (Kapitel 4.1.2) angelegte Sektion »Files« kann beispielsweise wie folgt aussehen:

```
[FILES]
History0=Ellipse.TVF
History1=KlassenHierarchie.TVF
History2=VCL2.TVF
History3=Lines.TVF
```

TIniFile

Mit einem Objekt der Klasse *TIniFile* können Sie objektorientiert auf die Einträge einer INI-Datei zugreifen. Nachdem Sie die Unit *IniFiles* eingebunden haben, müssen Sie zunächst ein Objekt der genannten Klasse initialisieren, indem Sie deren Konstruktor mit dem Namen der Datei aufrufen:

```
IniFile := TIniFile.Create(IniName); // manuelle Freigabe erforderlich
```

Danach können Sie mit Hilfe der Methoden *ReadInteger*, *ReadString* und *ReadBoolean* sowie mit *WriteInteger*, *WriteString* und *WriteBoolean* auf die einzelnen Einträge zugreifen. Die Methoden arbeiten nach dem folgenden Muster:

```
Wert := IniFile.Read...(Sektion, Schluessel, Voreinstellung);
Write...(Sektion, Schluessel, Wert);
```

Die *Read*-Methoden geben den Wert des Schlüssels einer bestimmten Sektion zurück. Falls der angegebene Schlüssel nicht existiert, liefern sie den Wert, den Sie im dritten Parameter als Voreinstellung angegeben haben. *ReadInteger* und *ReadBoolean* ersparen es Ihnen, die Zeichenkette aus der Datei in einen Integer- bzw. Boolean-Wert umzuwandeln.

Auch die *Write*-Methoden erhalten in den ersten beiden Parametern den Sektionsnamen und den Schlüssel. Im dritten Parameter geben Sie den Wert an, der als Zeichenkette in die INI-Datei geschrieben wird.

Hinweis: Windows 95/98 verwalten INI-Dateien mit Hilfe eines Caches. Um sicherzugehen, dass bei einem Absturz keine Änderungen an einer INI-Datei verloren gehen, können Sie *TIniFile.UpdateFile* aufrufen. Dadurch bewirken Sie eine sofortige Aktualisierung der zugrunde liegenden Datei.

Lesen von kompletten Sektionen *R114*

Nützlich sind auch die *TIniFile*-Methoden *ReadSection* und *ReadSectionValues*. Sie lesen eine komplette Sektion ein und fügen die Einträge in eine als Parameter übergebene Stringliste ein:

```
  ReadSection(IniSection, StringList)
{ Dieser Aufruf füllt StringList nach dem folgenden Muster:
    History0
    History1  usw. }
  ReadSectionValues(IniSection, Stringlist)
{ füllt StringList nach dem folgenden Muster:
    History0=Ellipse.TVF
    History1=TRY.TVF        usw. }
```

Nur mit *ReadSectionValues* erhalten Sie auch den Teil der Einträge nach dem Gleichheitszeichen. In diesem Fall können Sie das *TStrings*-Property *Values* verwenden, um nur diesen Teil abzufragen. So liefert beispielsweise *Stringlist.Value['History0']* den Wert »Ellipse.TVF« .

Im Zusammenhang mit Sektionen sind übrigens auch die Methode *EraseSection*, die eine Sektion komplett löscht, und die Methode *ReadSections*, die eine Liste aller vorhandenen Sektionen erzeugt, erwähnenswert.

Hinweis: Seit Delphi 4 sind alle besprochenen Properties und Methoden von *TIniFile* schon in der Klasse *TCustomIniFile* definiert, von wo sie nicht nur an *TIniFile*, sondern auch an *TMemIniFile* und *TRegistryIniFile* vererbt werden.

TMemIniFile

Für den schnellen Zugriff auf große INI-Dateien mit vielen Parametern gibt es seit Delphi 4 die Klasse *TMemIniFile*. Sie erzeugt im *Create*-Konstruktor ein Speicherabbild der gesamten INI-Datei und verarbeitet es intern zu Sektionslisten. Wenn Sie mit den schon von *TIniFile* bekannten Methoden Änderungen am Dateiinhalt vornehmen, so werden diese nur an den internen Daten von *TMemIniFile* vorgenommen. Um die Änderungen auch in der echten Datei zu speichern, müssen Sie explizit die Methode *UpdateFile* aufrufen (bei Freigabe eines *TMemIniFile*-Objekts geschieht dies nicht automatisch!) Falls Sie am erwähnten Speicherabbild der INI-Datei interessiert sind: Dieses können Sie mit den Methoden *GetStrings* und *SetStrings* in Form einer Stringliste direkt auslesen oder neu setzen.

4.2.2 Die Windows-Registrierung (Registry)

Die Registry wird von Windows intern verwaltet und stellt sich dem Softwareentwickler (und dem Benutzer, der es wagt, den Registry-Editor zu öffnen) als riesiger Baum von Informationen dar, dessen gröbste Unterteilung durch die *Wurzelschlüssel* vorgenommen wird, welche in Abbildung 4.1 durch den Registry-Editor *RegEdit* dargestellt sind (`regedit.exe` können Sie über das Windows-Menü START | AUSFÜHREN... aufrufen).

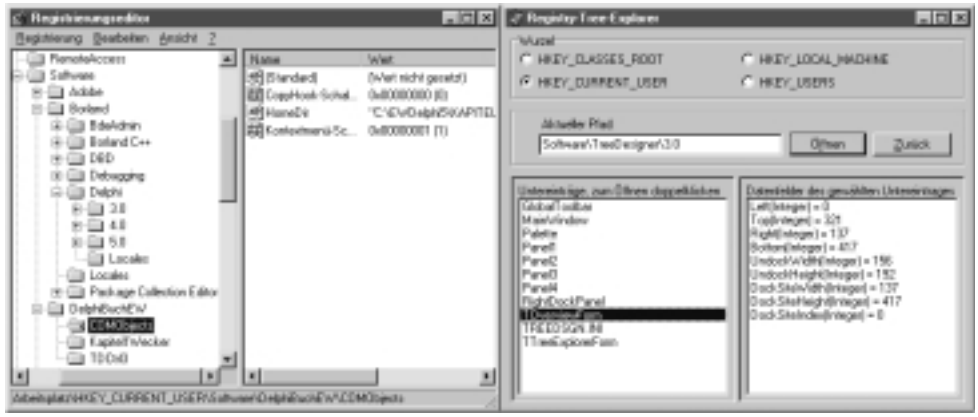


Abbildung 4.1: Der Registrierungseditor von Windows 95 und das Beispielprogramm RegBrows zum Browsen durch die Registry

Die drei wichtigsten Wurzelschlüssel sind:

- *HKey_Classes_Root* enthält Informationen zu allen Windows-Klassen im weiteren Sinn (Verknüpfungen von Dateiendungen mit Anwendungen, COM-Klassen und die Registrierung aller anderen Schnittstellen nach dem Component Object Model).

- *HKey_Current_User* beinhaltet Informationen, die speziell für den aktuellen Benutzer gültig sind. Dieser Schlüssel gibt tatsächlich nur einen Teil des Baumes *HKey_Users* wieder, in dem auch eventuelle weitere Benutzerdaten gespeichert werden.
- *HKey_Local_Machine* speichert Informationen, die für alle Benutzer der Arbeitsstation gemeinsam gelten.

Jeder dieser Schlüssel hat Untereinträge, die jeweils Informationen wie die Sektion einer INI-Datei, aber auch weitere Untereinträge enthalten können. Statt der Angabe einer Sektion ist also bei der Registry immer die Angabe eines Eintrags-Pfades notwendig; so gibt es in *HKey_Local_Machine* beispielsweise den Pfad *System\CurrentControlSet\control\ComputerName*.

Für die Konfigurationsdaten von Anwendungen sind die Untereinträge *Software* der Schlüssel *HKey_Current_User* und *HKey_Local_Machine* gedacht. So legt Delphi seine Informationen beispielsweise jeweils im Schlüssel *Software\Borland\Delphi\6.0* der Bereiche *HKey_Local_Machine* und *HKey_Current_User* ab, wobei es in *HKey_Local_Machine* nur die Pfade und die Delphi-Ausgabe (Standard, Professional oder Enterprise) ablegt, die unbestritten für alle Benutzer der Arbeitsstation gelten. Alle anderen Informationen werden für jeden Benutzer getrennt abgelegt, und zwar jeweils im aktuellen *HKey_Current_User*-Baum.

Browsen durch die Registry mit TRegistry *R157*

Das Beispielprogramm *RegBrows* auf der CD (Abbildung 4.1) zeigt, wie Sie mit Hilfe der VCL-Klasse *TRegistry* durch die Daten der Registrierung browsen können. Das Programm selbst kann hier nicht im Detail besprochen werden, daher sind hier nur die wichtigsten Ausschnitte daraus gezeigt. Die Einbindung der Klasse sowie die Erstellung und Freigabe eines *TRegistry*-Objekts laufen wie gewöhnlich ab. Wenn Sie die Unit *Registry* eingebunden und eine Registry-Variable deklariert haben, erstellen Sie mit

```
R := TRegistry.Create; { Am Schluss Freigabe mit R.Free }
```

ein Objekt, mit dessen Hilfe Sie auf die Registry zugreifen können.

Die beiden wichtigsten Properties von *TRegistry* sind *RootKey* und *CurrentPath*. Beide zusammen geben einen eindeutigen Schlüssel (eine Sektion) der Registry an, den aktuellen oder *geöffneten Schlüssel*. Es kann immer nur ein Schlüssel geöffnet sein, vergleichbar mit dem aktuellen Verzeichnis eines Laufwerks.

Dabei speichert *RootKey* einen der Wurzelschlüssel *HKey_xxx*, und *CurrentPath* gibt den restlichen Pfad an. Wenn Sie in *RegBrows* z.B. den Radioschalter *HKey_Classes_Root* markieren, führt dieses die folgende Zeile aus:

```
R.RootKey := HKEY_CLASSES_ROOT;
```

Das Property *CurrentPath* kann nur gelesen werden. Das Beispielprogramm tut dies, um den aktuellen Pfad im Editierfeld *Aktueller Pfad* anzuzeigen:

```
Edit2.Text := R.CurrentPath;
```

Um den aktuellen Pfad auch zu verändern, verwenden Sie die Methode *OpenKey*. Wenn Sie im Beispielprogramm den Schalter *Öffnen* betätigen, so öffnet dieses den Schlüssel mit dem Pfad, den Sie im Editierfeld angegeben haben:

```
R.CloseKey; // zuerst den bisher geöffneten Schlüssel schließen
R.OpenKey(Edit2.Text, False(*nicht erzeugen*)); // neuen Schlüssel öffnen
```

Falls Sie einen Unterschlüssel des gegenwärtig geöffneten Schlüssels öffnen wollen, können Sie sich den Aufruf von *CloseKey* sparen, wenn Sie bei *OpenKey* keinen vollständigen Pfadnamen, sondern nur den Namen des Unterschlüssels angeben. Das Beispielprogramm tut dies, wenn Sie auf einen Eintrag der Untereintrags-Liste doppelklicken:

```
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  R.OpenKey(ListBox1.Items[ListBox1.ItemIndex], False);
  UpdateLists; // Neuausgabe beider Listen und Neusetzen des Editierfeldes
end;
```

Erstellen neuer Schlüssel

Der zweite Parameter von *OpenKey* gibt an, ob die Methode den Schlüssel erzeugen soll, falls er noch nicht existiert. In den obigen Beispielen ist er immer auf *False* gesetzt, weil das *RegBrws* davon ausgehen kann, dass die in der Liste aufgeführten Untereinträge schon existieren, und weil keine versehentliche Änderung der Registry über das Editierfeld und den Schalter *Öffnen* möglich sein soll.

Wenn Sie der Registry in Ihrer Anwendung neue Schlüssel hinzufügen wollen oder nicht wissen, ob die von Ihrer Anwendung benötigten Schlüssel bereits bestehen, und Sie einfach nur sicherstellen wollen, dass das nach dem Aufruf von *OpenKey* der Fall ist, setzen Sie den zweiten *OpenKey*-Parameter auf *True*.

GetKeyNames und GetValueNames

Immer, wenn ein gültiger Schlüssel geöffnet ist, können Sie mit den Methoden *GetKeyNames* und *GetValueNames* die Namen der Untereinträge (*GetKeyNames*) und der

Dateneinträge (*GetValueNames*) abfragen. Das Beispielprogramm verwendet diese Funktionen, um die Untereinträge in der ersten ListBox anzuzeigen:

```
procedure TForm1.UpdateDataList;
var
  Strings: TStrings;
begin
  Strings := TStringList.Create; // ein Stringlistenobjekt bereitstellen
  R.GetKeyNames(Strings);        // Stringliste mit Untereinträgen füllen
  ListBox1.Items := Strings;     // Stringliste von TListBox setzen
  ...
```

Um die Dateneinträge des gerade gewählten Untereintrags in der zweiten ListBox anzuzeigen, muss sie zuerst diesen gewählten Untereintrag öffnen. Das Setzen des *Items*-Properties der ListBox funktioniert wie im letztgenannten Fall:

```
R.OpenKey(ListBox1.Items[ListBox1.ItemIndex], False);
R.GetValueNames(Strings);
ListBox2.Items := Strings;
Strings.Free;
```

Hinweis: Die *OpenKey*-Anweisung, die beim Doppelklicken auf die Liste der Untereinträge ausgeführt wird (Methode *ListBox1DblClick*), setzt voraus, dass der Schlüssel geöffnet ist, der die in der Liste gezeigten Untereinträge besitzt. Weil die zuletzt gezeigten vier Zeilen diesen Pfad verändern, speichert *RegBrows* vor Ausführung dieser vier Zeilen den aktuellen Registry-Pfad (mit anderen Worten also den aktuellen Schlüssel) in einer Stringvariablen zwischen, um ihn danach mit *CloseKey* und *OpenKey* wiederherstellen zu können. Dies ist jedoch ein Detail, das nur auf den Aufbau des Beispielprogramms zurückzuführen ist; es wurde daher nicht abgedruckt.

Die Daten der Registry

Das Browsen der Registry alleine ist nicht übermäßig sinnvoll, da Windows hierfür ja schon den Registry-Editor zur Verfügung stellt. Wir benötigen daher noch Methoden zum Schreiben und Lesen der Registry-Einträge. Die Methoden zum Lesen sollen hier als Beispiel genügen, denn im Wesentlichen kommt es nur auf das Prinzip der verschiedenen Datentypen an, das beim Lesen wie beim Schreiben von Einträgen dasselbe ist.

Die Dateneinträge innerhalb eines Registry-Schlüssels, also die Einträge, die mit *GetValueNames* gelesen werden können und im Beispielprogramm in der rechten Liste angezeigt werden, entsprechen den einzelnen Zeilen einer INI-Datei-Sektion. Im Gegensatz zu INI-Dateien gestattet die Registry jedoch auch binäre Eintragungen variabler Länge.

TRegistry stellt eine Reihe von Methoden bereit, mit denen Sie die Daten der Einträge auslesen können. Je nach Datentyp wählen Sie eine der Methoden *ReadString*, *ReadInteger*, *ReadDateTime* usw. aus.

Wenn Sie den Typ eines Eintrags nicht vorher kennen, können Sie ihn mit *GetDataType* abfragen. *RegBrows* fragt den Typ in einer *case*-Abfrage ab, liest dabei alle nicht-binären Daten mit *ReadString* bzw. *ReadInteger* und hängt ihren Inhalt an den Eintrag der Liste an. Für binäre Einträge zeigt es nur die Länge der Daten an:

```
for i := 0 to Items.Count-1 do begin
  ItemName := Items[i];
  case R.GetDataType(ItemName) of
    rdUnknown: Items[i] := ItemName+' hat unbekannten Typ';
    rdExpandString,
    rdString: Items[i] := ItemName+'(String) = '+R.ReadString(ItemName);
    rdInteger: Items[i] := ItemName+'(Integer) = '
                           +IntToStr(R.ReadInteger(ItemName));
    rdBinary: Items[i] := ItemName+' hat binären Typ, Länge: '
                          +IntToStr(R.GetDataSize(ItemName));
  end;
end;
```

Binäre Daten können Sie auch mit *ReadDateTime*, *ReadCurrency* usw. lesen, sofern die Daten die passende Größe haben. Normalerweise werden Sie diese Methoden nur aufrufen, wenn Sie die Daten selbst mit *WriteDateTime*, *WriteCurrency* usw. geschrieben haben, so dass Sie dann den Datentyp gar nicht mehr abzufragen brauchen und die Lesefunktion sofort aufrufen können.

Der Standardeintrag

Ein weiterer kleiner Unterschied zu INI-Dateien ist, dass jeder Schlüssel der Registry einen *Standardeintrag* besitzen kann. Ein solcher Eintrag besitzt keinen eigenen Namen und wird in der Spalte *Name* der rechten Fensterhälfte des Registrierungseditors mit dem Hinweis »(Standard)« versehen. Mit *TRegistry* sprechen Sie einen solchen Eintrag an, indem Sie den leeren String als Namen verwenden, z.B. *R.ReadInteger('')*.

TRegIniFile

Alternativ zu den Lese- und Schreibmethoden von *TRegistry* können Sie auch auf die Methoden von *TRegIniFile* zurückgreifen, um mit der Registry zu arbeiten. *TRegIniFile* ist von *TRegistry* abgeleitet und stellt Methoden zur Verfügung, deren Namen und Parameter mit denen von *TIniFile* übereinstimmen. Einziger weiterer Unterschied ist, dass Sie im Sektionsparameter der Methoden von *TRegIniFile* keinen Abschnitt der INI-Datei, sondern einen kompletten Pfad der Registry angeben, z.B. *Software\MeineAnwendung\Sektion*. *TRegIniFile* schreibt per Voreinstellung in den Baum *HKey_Current_User*, Sie können jedoch auch jeden anderen Registry-Bereich ansprechen, indem Sie das von *TRegistry* geerbte *RootKey*-Property neu setzen.

TRegistryIniFile

TRegistryIniFile stellt einfach eine Hülle um die Klasse *TRegIniFile* dar, d.h., dass ein *TRegistryIniFile*-Objekt intern ein *TRegIniFile*-Objekt verwendet, um seine Funktionen auszuführen (dieses Objekt ist über das Property *TRegistryIniFile.RegIniFile* auch von außen ansprechbar). Der Grund, warum Sie diese neue Klasse *TRegistryIniFile* der alten Klasse vorziehen könnten, liegt in der Polymorphie der Basisklasse *TCustomIniFile*. Kapitel 4.2.4 gibt ein Beispiel für die Verwendung einer *TCustomIniFile*-Variable, der zur Laufzeit entweder ein *TRegistryIniFile*- oder ein *TIniFile*-Objekt zugewiesen wird.

4.2.3 Die Registry und die Windows-Shell

Eine wichtige Funktion der Registry ist es, Daten über die im System installierten Anwendungen zu sammeln, die nicht nur eine anwendungsinterne, sondern eine systemweite Bedeutung haben. Hierzu gehören die Daten der Dateittypen, die von einer Anwendung bearbeitet werden können, sowie zahlreiche in Zusammenhang mit OLE und COM stehende Informationen. Es gibt ganze Bücher, die sich nur mit der Registry befassen, wir werden daher an dieser Stelle nur einen ganz kleinen Teil der Registry untersuchen können (siehe auch Kapitel 8.5.2 zur Registrierung von Shell-Erweiterungen).

Registrieren einer Anwendung *R148*

Um eine Anwendung für einen bestimmten Dateityp zu registrieren und den Dateitypen mit einer Beschreibung sowie mit einem Icon zu versehen, müssen Sie zwei neue Schlüssel unter *HKey_Classes_Root* anlegen, wobei einer der Schlüssel weitere Untereinträge hat (siehe Abbildung 4.1 unten, Schlüssel *TreeDesignerDoc*).

Die folgende Aufzählung erklärt die einzelnen Schritte, die im nächsten Abschnitt noch einmal durch ein Listing zusammengefasst werden:

- Zunächst benötigt die Anwendung eine eigene Dateikennung, im Fall des TreeDesigners aus Kapitel 5 ist das beispielsweise `.tvf` (TreeViewerFile).
- Diese Dateiendung müssen Sie in der Registry eintragen. Fügen Sie dazu direkt unter dem Schlüssel *HKey_Classes_Root* die Dateikennung inklusive des Punktes als neuen Schlüssel ein und setzen Sie den namenlosen Standardeintrag dieses Schlüssels auf die Bezeichnung, mit der Sie sich im weiteren Verlauf auf diesen Dokumenttyp beziehen wollen. Delphi registriert z.B. den Dateityp `.pas` als *DelphiUnit*. Für das gleich folgende Listing lautet die Bezeichnung für den Dateityp `.tvf` »TreeDesignerDoc« .
- Legen Sie nun unter *HKey_Classes_Root* einen weiteren Schlüssel für den Dokumenttyp an, also z.B. *TreeDesignerDoc*. Dieser Schlüssel ist dafür vorgesehen, verschiedene Daten bezüglich des Dokumenttyps zu sammeln.

- Der Standardeintrag dieses Schlüssels enthält zunächst den Namen des Dateityps, der in Explorer-Fenstern beispielsweise im Modus ANSICHT | DETAILS in der Spalte *Typ* angezeigt wird. Diese Bezeichnung kann beispielsweise »TreeDesigner Dokument« heißen.
- Damit beim Doppelklick auf eine Datei dieses Typs automatisch die zugehörige Anwendung gestartet wird, geben Sie diese im Untereintrag *Shell \ Open \ Command* des zuletzt angelegten Dokumenteintrags an. Neben dem Pfad der EXE-Datei müssen Sie auch die notwendigen Befehlszeilenparameter angeben, wobei der Platzhalter »%1« für den Namen der zu ladenden Datei steht. Windows ersetzt den Platzhalter beim Doppelklick auf eine Datei durch deren Namen und setzt voraus, dass Ihre Anwendung diesen Befehlszeilenparameter auch abfragt und die angegebene Datei automatisch lädt (wie der TreeDesigner das macht, ist im Abschnitt *Kindfenster bei Programmstart öffnen* in Kapitel 5.7.4 beschrieben). Entsprechend lässt sich die Delphi-IDE unter anderem durch einen Doppelklick auf eine `.pas`-Datei starten, da Delphis Installationsprogramm unter *DelphiUnit \ Shell \ Open \ Command* einen entsprechenden Eintrag zum Start von `delphi32.exe` in die Registry schreibt.
- Um schließlich für alle Dateien mit der registrierten Kennung nicht nur eine eigene Typenbezeichnung, sondern auch ein bestimmtes Icon anzuzeigen, fügen Sie dem Schlüssel des Dokumenttyps einen *DefaultIcon*-Untereintrag hinzu, dessen Standardwert den Namen einer Datei angibt, die das Icon enthält, sowie einen Index, der eines von eventuell mehreren Icons dieser Datei auswählt.

Registrierungsdateien

Die Windows-Registry bietet neben der programmgesteuerten und der manuellen Methode eine weitere Möglichkeit, Einträge zu verändern und hinzuzufügen. Hierbei werden *Registrierungsdateien* verwendet; das sind Dateien mit der Endung `.reg`, die fast wie eine INI-Datei aufgebaut sind und die Sie auch im Registrierungseditor über den Menüpunkt REGISTRIERUNG | REGISTRIERUNGSDATEI EXPORTIEREN... erzeugen können.

Das folgende Beispiel fasst die im letzten Abschnitt beschriebenen Einträge zusammen, mit denen der TreeDesigner in der Windows-Shell registriert wird. Wenn Sie die Verzeichnisse Ihrer Installation anpassen und die Datei in den Registrierungseditor importieren, stellt die Shell `.tvf`-Dateien von nun an mit dem ersten Icon dar, das in der EXE-Datei des TreeDesigners enthalten ist, gibt ihren Typ als »TreeDesigner Dokument« an und startet den TreeDesigner automatisch, wenn Sie auf eine solche Datei doppelklicken.

```
REGEDIT4

[HKEY_CLASSES_ROOT\.tvf]
@="TreeDesignerDoc"
```

```
[HKEY_CLASSES_ROOT\TreeDesignerDoc]
@="TreeDesigner Dokument"

[HKEY_CLASSES_ROOT\TreeDesignerDoc\Shell\Open\Command]
@="C:\\TreeDesigner Verzeichnis\\TREEDSGN.EXE %1"

[HKEY_CLASSES_ROOT\TreeDesignerDoc\DefaultIcon]
@="C:\\TreeDesigner Verzeichnis\\TREEDSGN.EXE, 1"
```

Die erste Zeile gibt mit »RegEdit4« quasi eine Dateikennung, danach folgen einzelne Abschnitte, deren Überschriften in eckigen Klammern den vollständigen Pfad eines Registry-Schlüssels angeben.

In diesen Abschnitten folgen wie in den Abschnitten einer INI-Datei einzelne Werte nach dem Muster »Schlüssel=Wert«, wobei Strings in Anführungszeichen geschrieben werden und der umgekehrte Schrägstrich »\« als »\\« geschrieben wird (da ein einzelner Schrägstrich die in C üblichen Steuerzeichenangaben einleitet). Der Standardeintrag eines jeden Schlüssels wird mit dem Zeichen »@« gekennzeichnet.

Sie können Registrierungsdateien manuell im Registrierungseditor über REGISTRIERUNG...\REGISTRIERUNGSDATEI IMPORTIEREN... in die Registry einfügen oder automatisch, indem Sie `regedit.exe` aufrufen und als Parameter den Namen der `.reg`-Datei angeben.

4.2.4 Speichern der History-Liste in Registry bzw. INI-Datei

Als Beispiel zur Verwendung der Klassen *TIniFile* und *TReg(istry)IniFile* dient nun die Speicherfunktion der in Kapitel 4.1.2 vorgestellten History-Liste. Die Klasse *THistoryList* kann die Einträge der History-Liste sowohl in der Registry als auch in einer INI-Datei speichern. Für die Verwendung einer INI-Datei setzen Sie das standardmäßig auf *True* gesetzte Property *THistoryList.UseRegistry* auf *False*. Zuständig für Speichern und Laden sind die Methoden *SaveToIni* und *LoadFromIni*. Die Bedeutung der beiden Parameter hängt davon ab, ob Sie die Registry oder eine INI-Datei verwenden:

- Im Falle einer INI-Datei bezeichnet *IniName* den Namen einer INI-Datei und *IniSection* gibt die Abschnittsüberschrift an.
- Im Falle der Registry meint *IniName* einen Pfad der Registry, *IniSection* gibt einen Untereintrag für diesen Pfad an.

Die Namen der einzelnen Dateneinträge entstehen durch Verknüpfung des Strings »History« mit einem Index, so dass sich ein INI-Dateiabschnitt wie in Kapitel 4.2.1 gezeigt ergibt. Der wesentliche Punkt im folgenden Listing ist, dass nur die Erzeugung des Objekts *IniFile* von der Einstellung des *UseRegistry*-Properties abhängig ist. *IniFile* ist ein polymorphes Objekt. Der Aufruf der Methode *IniFile.WriteString* funktioniert also sowohl mit einem *TRegistryIniFile*- als auch mit einem *TIniFile*-Objekt:

```
procedure THistoryList.SaveToIni(IniName, IniSection: string);
var
  i: integer;
  IniFile: TCustomIniFile;
begin
  if FUseRegistry then
    IniFile := TRegistryIniFile.Create(IniName)
  else
    IniFile := TIniFile.Create(IniName);
  try
    { Hinweis: Falls mehr als Count Einträge gespeichert sind,
      werden die überzähligen Einträge nicht überschrieben. }
    for i := 0 to Count-1 do
      IniFile.
         WriteString(IniSection, 'History'+IntToStr(i), Strings[i]);
  finally
    IniFile.Free;
  end;
end;
```

Entsprechend liest *LoadFromIni* die Einträge mit *ReadString* wieder ein. Dabei kann es sein, dass weniger als *MaxLen* Strings in der INI-Datei gespeichert sind. In diesem Fall liefert *ReadString* für die letzten Strings den als Voreinstellung angegebenen Leerstring zurück:

```
{ Ausschnitt aus THistoryList.LoadFromIni }
  for i := 0 to FMaxLen-1 do begin
    s := IniFile.ReadString(IniSection, 'History'+IntToStr(i), '');
    if s <> '' then
      Add(s);
  end;
```

Damit ist alles Wichtige zur Beispielklasse *THistoryList* gesagt. Ein Wiedersehen mit dieser Klasse wird es in Kapitel 6.5.3 geben, denn dort wird sie in die Komponente *THistoryCombo* eingebaut.

4.2.5 Die TStateSaver-Komponente

Es gibt sehr viele kleine Anwendungsmöglichkeiten für Konfigurationsdateien, die nicht genutzt werden, weil das Implementieren des Codes zum Speichern und Wiederherstellen der Einstellungen zu umständlich ist. Dies gilt besonders für Software-Entwickler, die während der Programmentwicklung ein bestimmtes Programm ständig neu starten und zum Testen einer bestimmten Funktion immer wieder dieselben Einstellungen im Programmfenster vornehmen müssen, z.B. bestimmte Markierungsschalter wählen, einen Text eingeben, ein Verzeichnis aus dem tief strukturierten Verzeichnisbaum der Festplatte auswählen usw. Mit Delphi können Sie sich hier zwar teilweise Erleichterung verschaffen, indem Sie die häufig benötigten Einstellungen bereits zur Entwurfszeit im Formular vornehmen. Doch wenn Sie sich dann während

eines weiteren Testlaufs dazu entschließen, die Einstellungen zu ändern, dann hat das keine Auswirkungen auf die Entwurfszeit-Einstellungen und beim nächsten Programmstart werden wieder die alten Einstellungen verwendet.

TStateSaver *R112*

In der Komponentenbibliothek dieses Buches finden Sie mit *TStateSaver* eine Komponente, die hier Abhilfe schaffen soll. Es genügt, ein Exemplar dieser nicht-visuellen, also zur Entwurfszeit unsichtbaren Komponente auf ein Formular abzulegen und die folgenden Einstellungen vorzunehmen:

- In *ResFileName* geben Sie den Namen einer Konfigurationsdatei ein, sofern Sie nicht mit der Voreinstellung zufrieden sind.
- In *Section* geben Sie an, in welchem Abschnitt der INI-Datei die automatische Speicherung stattfinden soll. Sie können also in *ResFileName* eine Datei angeben, die Sie an anderer Stelle auch manuell mit *TIniFile* bearbeiten. Durch die eigene Sektions-Überschrift sind alle von *TStateSaver* automatisch gespeicherten Daten von Ihren manuell gespeicherten Daten klar getrennt.
- In *UseHomeDirectory* legen Sie fest, ob die INI-Datei im Home-Verzeichnis des aktuellen Benutzers abgelegt werden soll. Ist dieses Property *False*, wird als Speicherort das Verzeichnis der ausführbaren Datei verwendet.

Dies genügt bereits, damit Sie eine automatische Speicherung der folgenden Properties von Komponenten Ihres Formulars erhalten: *T(Custom)Edit.Text*, *TSpinEdit.Value*, *TMaskEdit.Text*, *TTrackBar.Position*, *TCheckBox.Checked*, *TRadioButton.Checked*, *TRadioGroup.ItemIndex*, *TTabControl.TabIndex*, *TPageControl.ActivePageIndex*. All diese Eigenschaften werden beim nächsten Programmstart automatisch wieder hergestellt. Die Liste dürfte wohl die am häufigsten verwendeten Komponenten mit ihren wichtigsten zur Laufzeit veränderten Properties enthalten, jedoch können Sie die Liste auch leicht auf Ihre speziellen Erfordernisse erweitern.

TStateSaver erweitern

Hierzu genügt es nämlich, die *TStateSaver*-Methoden *SaveState* und *RestoreState* zu erweitern. Zur Illustration sei die *SaveState*-Methode (etwas gekürzt) wiedergegeben, die *RestoreState*-Methode unterscheidet sich im Prinzip nur durch die Verwendung von *Read...*- statt *Write...*-Methoden und soll daher unabgedruckt auf der CD verbleiben:

```
procedure TStateSaver.SaveState;
var
  i: Integer;
  IniFile: TMemIniFile;
  C: TComponent;
begin
```

```
  if Owner <> nil then begin
    IniFile := TMemIniFile.Create(GetFullIniPath);
    try
      if SaveWindowState and (Owner is TCustomForm) then
        SaveWindowPos(IniFile, FSection, Owner as TCustomForm);
      for i := 0 to Owner.ComponentCount - 1 do begin
        C := Owner.Components[i];
        if C.Name = '' then continue; // weiter mit nächstem Fall
        if FExcludeList.IndexOf(C.Name) <> -1 then continue;
        if C is TCustomEdit then with C as TEdit do
          IniFile.WriteString(FSection, Name, Text)
        else if C is TTrackBar then with C as TTrackbar do
          IniFile.WriteInteger(FSection, Name, Position)
        else if C is TSpinEdit then with C as TSpinEdit do
          IniFile.WriteInteger(FSection, Name, Value)
        else if C is TRadioGroup then with C as TRadioGroup do
          IniFile.WriteInteger(FSection, Name, ItemIndex)
        ...
        ... (ein Fall pro unterstützter Komponentenklasse) ...
        ...
        else if C is TCheckBox then with C as TCheckBox do
          IniFile.WriteInteger(FSection, Name, Integer(Checked))
      end;
    finally
      IniFile.UpdateFile;
      IniFile.Free;
    end;
  end;
end;
```

Um dies z.B. durch eine von Ihnen gewählte Komponente *TMyControl* zu erweitern, fügen Sie einfach einen neuen Fall in *SaveState* ein:

```
...
else if C is TMyControl then with C as TMyControl do
   IniFile.WriteType(FSection, Name, PropertyName)
```

Analog dazu gilt es, in die hier nicht abgedruckte Methode *RestoreState* Folgendes einzufügen:

```
else if C is TMyControl then with C as TMyControl do
   TabIndex := IniFile.ReadType(FSection, Name, PropertyName)
```

Dabei setzen Sie für *PropertyName* den Namen des Properties ein, das gespeichert werden soll, und entsprechend seines Typs passen Sie auch den Aufruf Write*Type* an.

Funktionsweise

Das Listing von *SaveState* enthüllt bereits das Funktionsprinzip von *TStateSaver*: Da die Komponente in das Formular eingefügt wird, wird das Formular zu ihrem Besitzer und ist folglich aus der Sicht der Komponente über das *Owner*-Property anzusprechen.

Über das Formular erhält *TStateSaver* nun Zugriff auf alle darin enthaltenen Komponenten. *SaveState* muss also nur das *Components*-Array durchlaufen und den Typ der diversen Komponenten testen. Falls es sich um einen unterstützten Typ handelt, wird dafür ein Eintrag in der INI-Datei erzeugt, dessen Name sich aus dem Namen der Komponente aus dem *Components*-Array ableitet (theoretisch könnten auch mehrere Einträge für mehrere Properties der Komponente erzeugt werden). Das Ergebnis von *SaveState* könnte beispielsweise so aussehen (natürlich ohne die Kommentare):

```
[StateSaverAutoSave] // <- Name im ResFileName-Property
Edit1=tempfilename.xyz // TEdit1.Text wurde gespeichert
CheckBox1=1            // TCheckBox.Checked als Zahl
TrackBarIterations=40  // TTrackBar.Position
```

Der automatische Aufruf der *SaveState*- und *RestoreState*-Methoden kommt wie folgt zustande: *RestoreState* wird in der virtuell überschriebenen Methode *Loaded* aufgerufen, die von der CLX aufgerufen wird, wenn die Komponente aus einer Formulardatei geladen wurde (siehe Kapitel 6.4.2), zum Aufruf der *SaveState*-Methode klinkt sich die Komponente in das *OnClose*-Ereignis des Formulars ein. Sie verknüpft dieses mit der folgenden Ereignisbearbeitungsmethode:

```
procedure TStateSaver.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  if Assigned(FOldCloseEventMethod) then
    FOldCloseEventMethod(Sender, Action);
  SaveState;
end;
```

Darin wird auch die vor der Verknüpfung zwischengespeicherte alte Bearbeitungsmethode für das *OnClose*-Ereignis (falls Sie also zur Entwurfszeit auch eine Methode für *OnClose* geschrieben haben) aufgerufen. Zu Konflikten zwischen Ihrer *OnClose*-Methode und der StateSaver-*OnClose*-Methode kommt es nur dann, wenn Sie *OnClose* zur Laufzeit verändern wollen. Bei Verwendung von *TStateSaver* sollten Sie dies also vermeiden (oder Sie müssen die Automatik von *TStateSaver* abschalten und *TStateSaver.SaveState* manuell beim Schließen des Formulars aufrufen).

Weitere Properties von TStateSaver *R113*

StateSaver besitzt noch zwei weitere Properties, die zum Abschluss noch erwähnt werden sollen:

- *SaveWindowState* ist ein boolesches Property, in dem Sie die Komponente anweisen können, Formularposition, -größe und -zustand (*WindowState*) ebenfalls zu speichern (im gleichen Abschnitt der INI-Datei wie die anderen Daten; die Namen der Fensterstatus-Einträge beginnen mit »FormState«). Die Wiederherstellung findet beim nächsten Programmstart automatisch statt.

- *ExcludeList* ist eine *TStringList*, in der Sie eine Liste von Komponentennamen angeben, die *nicht* gespeichert werden sollen, was immer es für einen Grund dafür geben mag (die Komponente wird bereits vom Programm mit einem passenden Vorgabewert initialisiert, sie zeigt vertrauliche Informationen an etc.)

4.3 Streams und Ablageobjekte

Die objektorientierte Entsprechung für die in Kapitel 2.8.2 beschriebenen Dateien sind die Streams, deren grundsätzliches Wesen von der abstrakten Basisklasse *TStream* definiert wird.

4.3.1 Stream-Klassen

TStream definiert nur die virtuellen und abstrakten Methoden, mit denen Sie Daten in einem Stream speichern und von dort lesen können. Es ist Aufgabe der von *TStream* abgeleiteten Klassen, zu definieren, *wo* diese Daten gespeichert werden. Die VCL-Unit *Classes* enthält bereits einige konkrete Stream-Typen: *THandleStream*, *TFileStream*, *TStringStream* und *TMemoryStream*.

Da *THandleStream* nur eine Zwischenstufe zu *TFileStream* ist und Sie einen *TMemoryStream* mit einem einfachen *Create*-Konstruktor erstellen können, konzentrieren wir uns hier auf *TFileStream*. Sie öffnen einen solchen Stream und damit eine Datei wie folgt:

```
Stream := TStream.Create(Dateiname, fmOpenRead);
```

Im ersten Parameter geben Sie einen Dateinamen an, bei Bedarf mit Pfad, im zweiten Parameter legen Sie den so genannten Dateimodus fest, der besagt, wie Ihre Anwendung und andere Anwendungen auf die Datei zugreifen können, solange diese geöffnet ist.

Mit *fmOpenRead*, *fmOpenWrite* und *fmOpenReadWrite* erhalten Sie Lese- oder Schreibzugriff auf eine bestehende Datei oder beides gleichzeitig. Wenn Sie eine neue Datei anlegen wollen, die eine eventuell schon bestehende Datei mit dem gleichen Namen überschreibt, verwenden Sie den Modus *fmCreate*. Diese Konstanten können Sie in einer *or*-Verknüpfung mit den Konstanten *fmShareExclusive*, *fmShareDenyWrite*, *fmShareDenyRead* und *fmShareDenyNone* verknüpfen, um anderen Anwendungen jedweden Zugriff auf diese Datei oder nur Lese- bzw. nur Schreibzugriff zu untersagen (z.B. *fmOpenRead or dmShareDenyWrite*).

4.3.2 Der Inhalt von Streams

In Streams können Sie natürlich alle Daten schreiben, die Sie auch in normalen Dateien speichern können, jedoch kennen Streams keinerlei Datentypen. Statt dessen erwarten sie einen beliebigen Variablenparameter, dessen Größe Sie in einem zweiten Parameter angeben. Die typische Speicherung von Daten in einem Stream sieht damit wie folgt aus:

```
Stream.WriteBuffer(Points, sizeof(Points));
Stream.WriteBuffer(FShapeType, sizeof(FShapeType));
Stream.WriteBuffer(FText, length(FText)+1);
```

Entsprechend können Sie die Daten mit *ReadBuffer* wieder lesen. *WriteBuffer* und *ReadBuffer* erzeugen eine *EReadError*- bzw. *EWriteError*-Exception, falls es zu einem Lese- oder Schreibfehler kommt. Beide Methoden gehören noch nicht zu den grundlegendsten Methoden und Properties von *TStream*, die in der folgenden Tabelle zusammengefasst sind:

Typ	Name	Beschreibung
property	Position: LongInt	gibt die aktuelle Lese-/Schreibposition im Stream an.
property	Size: LongInt	gibt die Größe des Streams (bzw. die Dateigröße) in Byte an.
function	Write(var Buffer, Count): LongInt	schreibt *Count* Bytes von *Buffer* in den Stream, Funktionsergebnis ist die Zahl der erfolgreich geschriebenen Bytes.
function	Read(var Buffer, Count): LongInt	liest *Count* Bytes aus dem Stream und speichert sie in *Buffer*, liefert die Zahl der tatsächlich gelesenen Bytes.
function	Seek(Offset, Modus): LongInt	*Modus=soFromBeginning* – setzt die aktuelle Position auf *Offset*, *Modus=soFromCurrent* – bewegt die aktuelle Position um *Offset* Bytes nach vorne oder nach hinten, *Modus=soFromEnd* – positioniert die Schreib-/Lesemarke *Offset* Bytes vom Stream-Ende entfernt.

Alle anderen Lese- und Schreibmethoden bauen auf *Read* und *Write* auf. Zum einen gibt es die schon erwähnten *ReadBuffer* und *WriteBuffer*, die lediglich die Methoden *Read* bzw. *Write* aufrufen und dabei überprüfen, ob deren Ergebnis mit der gewünschten Byte-Anzahl (*Count*) übereinstimmt, und im Fehlerfall die genannten Exceptions erzeugen. Zum anderen dienen die Methoden *WriteComponent(Res)* und *ReadComponent(Res)* dazu, Komponenten innerhalb von Formularen zu speichern, allerdings berufen sich diese Methoden hauptsächlich auf die im nächsten Kapitel besprochenen Hilfsklassen.

Ein vollständiges Beispiel zum Speichern eines Dokuments in einem Stream finden Sie in Kapitel 5.3.3.

Hinweis: Sollten Ihre Programme Umgang mit Dateien pflegen, die größer als 4 GByte sind, können Sie in diesen erst ab Delphi 6 beliebige Positionen ansteuern, da die *Seek*-Methode nun im *Offset*-Parameter auch *Int64*-Werte unterstützt.

4.3.3 Polymorphe Objekte speichern

Mit Hilfe von RTTI (siehe Kapitel 2.3.6), virtuellen Konstruktoren und Klassenreferenzen (jeweils Kapitel 2.3.4) sowie mit einigen Routinen der Unit *Classes* können Sie auch Objekte von beliebigen Klassen so speichern, dass Sie sie beim Lesen komplett rekonstruieren können. Dazu genügt es nicht, nur die Daten des Objekts in die Datei zu schreiben. Damit beim Lesen der richtige Konstruktor aufgerufen werden kann, müssen Sie auch die Klasse des Objekts speichern. Ohne die Spezialitäten Object Pascal könnte das Lesen einer solchen Datei beispielsweise so aussehen:

```
{ Lesen der Objektklasse }
Stream.Read(ClassId, sizeof(ClassId));
{ Erstellen des Objekts }
case ClassId of
  1: NewObject := Class1.Create;
  2: NewObject := Class2.Create;
end;
{ Lesen der Objektdaten }
NewObject.ReadData
```

Schon in Kapitel 2.3.4 konnte eine derartige Fallunterscheidung durch die Verwendung von Klassenreferenzen vermieden werden, und auch in diesem Fall lassen sich Klassenreferenzen nutzbringend einsetzen. Vereinfacht gesehen können wir das obige Beispiel wie folgt ändern:

```
ClassRef := LiesKlassenreferenz(Stream);
NewObject := ClassRef.Create;
NewObject.ReadData;
```

Das Problem hierbei ist die Funktion *LiesKlassenreferenz*, denn sie existiert nicht. Die folgenden Routinen der Unit *Classes* helfen dabei, eine solche Funktion selbst zu implementieren:

```
procedure RegisterClass(AClass: TPersistentClass);
function FindClass(const ClassName: string): TPersistentClass;
```

Mit *RegisterClass* nehmen Sie die im Parameter angegebene Klassenreferenz in einer VCL-internen Liste auf, so dass Sie diese mit *FindClass* später über den Namen der Klasse abfragen können. Wenn Sie dann den Namen einer Klasse im Stream speichern, etwa wie in der folgenden Prozedur für das Objekt *obj* ...

```
procedure WriteClassRef(Writer: TWriter; obj: TObject);
begin
  Writer.WriteString(obj.ClassName);
end;
```

... können Sie mit Hilfe der *FindClass*-Funktion eine Klassenreferenz des gespeicherten Objekts wiederherstellen:

```
function ReadClassRef(R: TReader): TClass;
var
  ClassName: String;
begin
  ClassName := R.ReadString;
  Result := FindClass(ClassName);
end;
```

Die dritte Version des obigen Beispiels lautet dann:

```
ClassRef := ReadClassRef(ClassName);
NewObject := ClassRef.Create;
NewObject.ReadData;
```

Der Einfachheit halber verwendet dieses Beispiel die in Kapitel 4.3.4 behandelten *TFiler*-Klassen, welche die oben verwendeten Methoden *ReadString* und *WriteString* zum Schreiben und Lesen von Strings zur Verfügung stellen.

Vollständiges Beispiel zur polymorphen Objektspeicherung *R108*

Da es an dieser Stelle nicht darum gehen soll, die Daten einer Klasse zu speichern (ein Thema, das in Kapitel 5.3.3 erörtert wird), verwendet dieses Beispiel zwei »leere« Klassen, für die aber zur Veranschaulichung schon Methoden zum Speichern und Laden der Daten deklariert werden. Wie Sie der Deklaration von *RegisterClass* und *FindClass* aus der VCL entnehmen können, akzeptieren diese Funktionen nur Klassen, die von *TPersistent* abgeleitet sind, daher müssen auch die beiden Klassen des Beispiels von *TPersistent* abstammen. Für die Polymorphie des gesamten Vorgangs ist es wichtig, dass die Methoden zum Speichern und Laden der Daten aus der Datei ebenfalls polymorph sind, also in einer gemeinsamen Basisklasse, hier *TBaseClass*, als virtuell deklariert sind:

```
type
  TBaseClass = class(TPersistent)
    procedure WriteToFile(W: TWriter); virtual; abstract;
    procedure ReadFromFile(R: TReader); virtual; abstract;
  end;
  TBaseClassRef = class of TBaseClass;
  Class1= class(TBaseClass)
    procedure WriteToFile(W: TWriter); override;
    procedure ReadFromFile(R: TReader); override;
  end;
  Class2= class(TBaseClass)
```

```
    procedure WriteToFile(W: TWriter); override;
    procedure ReadFromFile(R: TReader); override;
  end;
```

Die folgende Prozedur wählt per Zufallszahl eine dieser beiden Klassen aus, erzeugt eine Objektinstanz davon und schreibt den Klassennamen in eine Datei (diesen Code finden Sie auf der CD eingebunden in das Projekt *PolyStrm*):

```
procedure PolymorpheObjekteSpeichern;
var
  S: TStream;
  Writer: TWriter;
  object1: TBaseClass;
begin
  S := TFileStream.Create(FileName, fmCreate);
  Writer := TWriter.Create(S, 2000);
  Randomize;
  if Random(10) < 5 then object1 := Class1.Create
    else object1 := Class2.Create;
  object2 := Class2.Create;
  WriteClassRef(Writer, object1);
  object1.WriteToFile(Writer);

  Writer.Free;
  S.Free;
end;
```

Die folgende Methode liest die von *SpeichernBeispiel* erzeugte Datei und rekonstruiert dabei das oben zufällig ausgewählte Objekt, wobei sie wie schon beschrieben die VCL-Methode *FindClass* verwendet. Vorher muss sie jedoch zuerst alle in Frage kommenden Klassen mit *RegisterClass* registrieren:

```
procedure TForm1.LesenPolymorpherDaten;
var
  S: TStream;
  Reader: TReader;
  ClassRef: TBaseClassRef;
  object1: TBaseClass;
begin
  RegisterClass(Class1);
  RegisterClass(Class2);

  S := TFileStream.Create(FileName, fmOpenRead);
  Reader := TReader.Create(S, 2000);
  ClassRef := TBaseClassRef(ReadClassRef(Reader));
  object1 := ClassRef.Create;
  object1.ReadFromFile(Reader);

  Reader.Free;
  S.Free;
end;
```

Sie können auf diese Weise beliebige von *TPersistent* abgeleitete Klassen registrieren und brauchen dann nur noch den Namen der Klasse im Stream zu speichern, um die Objekte später wie im obigen Beispiel mit Hilfe der Klassenreferenz wiederherstellen zu können.

Weitere Funktionen der Unit *Classes* erlauben Ihnen, die Registrierung zurückzunehmen, Alias-Namen zu definieren und ganze Arrays von Klassenreferenzen auf einmal an- und abzumelden. Informationen dazu finden Sie in der Online-Hilfe zur *Classes*-Unit oder im VCL-Quelltext. Die VCL selbst verwendet *RegisterClass*, *FindClass* und die verwandten Routinen dazu, die Komponenten eines Formulars ähnlich wie in den obigen Beispielen zu speichern.

Klassenregistrierung unabhängig von TPersistent

Die Beschränkung der VCL-Methoden *FindClass* und *RegisterClass* auf *TPersistent*-Ableitungen ist nicht immer akzeptabel, zumal sie sich sehr leicht umgehen lässt, indem man einfach eigene Versionen von *FindClass* und *RegisterClass* schreibt, die beliebige Klassen erlauben. Der TreeDesigner aus Kapitel 5 etwa speichert polymorphe Grafikelemente, deren Klasse *TGraphicElement* von *TObject* abgeleitet ist. Es wäre sicher nicht besonders lohnend, wenn nun jedes einzelne *TGraphicElement* zu einem *TPersistent*-Objekt gemacht werden würde, nur um die VCL-Methoden *RegisterClass* und *FindClass* verwenden zu können.

Statt dessen verwendet der TreeDesigner also eine neue Version dieser beiden Methoden, die Sie für Ihre eigene Verwendung in der Unit *StreamUtil* finden. Ansonsten folgt die polymorphe Speicherung von Grafikobjekten im TreeDesigner dem oben beschriebenen Muster.

```
var
  ClassList: TList;

function FindClass(AClassName: String): TClass;
var
  i: Integer;
begin
  for i := 0 to ClassList.Count - 1 do begin
    Result := ClassList[i];
    if Result.ClassNameIs(AClassName) then Exit;
  end;
end;

procedure RegisterClass(C: TClass);
begin
  if ClassList.IndexOf(C) = -1 then
    ClassList.Add(C);
end;
```

4.3.4 TReader und TWriter

TReader- und *TWriter*-Objekte sind Hilfsobjekte, die die Handhabung eines Streams vereinfachen. Während *TReader* komfortable Methoden zur Verfügung stellt, mit denen Sie Daten aus einem Stream lesen können, widmet sich *TWriter* dem ebenso komfortablen Schreiben der Daten.

Die gemeinsame Basisklasse dieser beiden Klassen ist *TFiler* (sie beschreibt ein allgemeines »Ablageobjekt«, das mit einem Stream verbunden ist). Als wichtige Gemeinsamkeiten definiert sie die Methoden *DefineProperty* und *DefineBinaryProperty*, die jedoch erst bei der Programmierung von Komponenten interessant werden.

Initialisierung

Um ein *TReader*- oder *TWriter*-Objekt zu erstellen, rufen Sie den Konstruktor der Klasse wie folgt auf:

```
R := TReader.Create(Stream, 2048);
W := TWriter.Create(Stream, 2048);
```

Im ersten Parameter geben Sie ein bereits initialisiertes Stream-Objekt an, aus dem *TReader* seine Daten bezieht bzw. in das *TWriter* seine Daten schreibt. Der zweite Parameter ist in beiden Fällen die Größe, die das Objekt seinem internen Datenpuffer zuweisen soll.

Arbeitsteilung zwischen Streams und Ablageobjekten

Sinn der Unterscheidung zwischen den *TStream*- und den *TFiler*-Klassen ist eine größere Unabhängigkeit zwischen der Speicherung eines Datenstroms (Streams) und dem Aufbau der Daten. Während *TStream* nur unstrukturierte Datenblöcke kennt und sich darauf konzentriert, die Daten an einem bestimmten Ort, beispielsweise in einer Datei, unterzubringen, kennen die *TFiler*-Klassen verschiedene Datentypen wie Fließkommazahlen, Integerzahlen und Strings, verlassen sich beim Datentransport aber ganz auf *TStream*. Auch die scheinbar leistungsfähigen *ReadComponent*- und *WriteComponent*-Methoden von *TStream* arbeiten intern mit *TFiler*-Objekten.

Aufgrund dieser Unabhängigkeit können Sie von *TFiler* oder *TReader* theoretisch eigene Klassen ableiten, die weitere Datentypen speichern könnten und mit denen Sie zum Beispiel sowohl *TMemoryStreams* als auch *TFileStreams* benutzen könnten (wäre die *TReader*/*TFiler*-Funktionalität in der Klasse *TStream* integriert, müssten Sie dazu *zwei* neue Klassen ableiten – jeweils eine von *TMemoryStream* und eine von *TFileStream*).

Trotz dieser flexiblen allgemeinen Struktur sind auch die *TFiler*-Klassen sehr stark auf das Speichern von Formularen und Komponenten ausgerichtet, so dass hier nur die einfachen Möglichkeiten dieser Klassen zusammengefasst werden.

Verwendung

Die einfachste Verwendung der *TFiler*-Klassen besteht in der Speicherung der einfachen Datentypen: Hierfür definiert *TReader* die Methoden *ReadBoolean*, *ReadChar*, *ReadInteger*, *ReadFloat*, *ReadCurrency*, *ReadString*, *ReadWideString*, *ReadVariant*, *ReadInt64* und weitere Methoden für selten verwendete Typen, wobei Sie mit *ReadInteger* alle Integertypen bis einschließlich *LongInt* lesen können und von *ReadFloat* Fließkommazahlen des Typs *Extended* erhalten. *TWriter* enthält die entsprechenden Schreibmethoden. Der Aufruf ist in allen Fällen denkbar einfach:

```
Writer.WriteInteger(IntegerVar);
IntegerVar := Reader.ReadInteger;
```

Listen

Als etwas komplexere Aktion unterstützen die beiden Klassen auch das Schreiben und Lesen von Listen. Eine solche Liste besteht aus einer Startmarkierung, einer Folge von Datenblöcken, deren Struktur Sie definieren, und einer Endemarkierung.

- Das Schreiben einer Liste beginnt mit dem Aufruf der Methode *WriteListBegin*, die die Startmarke setzt. Danach speichern Sie die einzelnen Listenelemente und rufen schließlich die Methode *WriteListEnd* auf.
- Das Lesen der Liste beginnt mit der Methode *ReadListBegin*. Danach können Sie in einer Schleife alle Listenelemente auslesen, wobei Sie von der Methode *TReader.EndOfList* erfahren, wann Sie das Listenende erreicht haben. Schließlich beenden Sie das Lesen durch den Aufruf von *ReadListEnd*.

Wie der gesamte Ablauf konkret aussieht, demonstriert ein vollständiges Beispiel zur Verwendung von *TReader* und *TWriter* ab Seite 542 in den Abschnitten *Speichern der Tastenkürzel* und *Lesen der Tastenkürzel*.

Interne Arbeitsweise

Die oben beschriebenen Methoden sind zwar sehr komfortabel, verursachen aber einigen Mehraufwand: Anstatt lediglich die reinen Daten zu speichern, also zwei Bytes für eine *ShortInt*-Zahl, ein Byte für ein Zeichen usw., stellen sie jedem Datenbestandteil eine Typenkennung voran. So überprüft beispielsweise *WriteInteger*, ob der übergebene Zahlenwert in den Bereich eines Bytes passt, und schreibt in diesem Fall nur dieses

eine Byte – plus dem Byte für die Typenkennung. Eine möglichst effiziente Speicherung größerer Datenmengen ist also nur mit den *TStream*-Klassen möglich (ein Beispiel dazu gibt Kapitel 5.3.3).

Diese Umstände haben jedoch den Vorteil, dass die mit *TReader* und *TWriter* geschriebenen Dateien problemlos zwischen verschiedenen Plattformen ausgetauscht werden können, ob das nun zwischen einer 16-Bit- und 32-Bit-Versionen einer Anwendung ist oder zwischen den heutigen 32-Bit-Anwendungen und zukünftigen 64-Bit-Nachfolgern. Wenn Sie beispielsweise früher unter 16-Bit-Windows mit Delphi 1 einen (16-Bit-)*Integer* mit *WriteInteger* geschrieben haben und diesen heute in einer 32-Bit-Version der Anwendung mit *ReadInteger* in einen (32-Bit-)Integer lesen, stellt *TReader* fest, dass der Wert in der Datei nur zwei Bytes beansprucht (wenn er im Wertebereich eines Bytes liegt, nur ein Byte). Sie liest dann nur die in die Datei geschriebenen Bytes und schreibt sie in den 32-Bit-Integer-Wert.

Zum Vergleich damit würde *TStream* heute, wenn Sie etwa mit

```
var
  Int: Integer;
begin
  Stream.Read(Int, sizeof(Int));
```

versuchen würden, einen 32-Bit-Integer zu erhalten, den 16-Bit-Wert aus der alten Datei nicht korrekt lesen. Es würden vier statt zwei Bytes gelesen und alle nachfolgenden Leseoperationen würden von der falschen Position starten.

4.3.5 Memory-Streams

Nicht immer sollen die Daten eines Programms in einzelnen Dateien gespeichert werden, sondern auch die Windows-Registry und die Zwischenablage sind beliebte Speicherziele. Dabei kommt es häufig vor, dass genau dieselben Daten, die mit *SaveToStream* oder ähnlichen Methoden in eine Datei geschrieben werden können, z.B. auch in die Zwischenablage kopiert werden sollen.

Die Polymorphie stellt einen Mechanismus dar, mit dem vorhandene *SaveToStream*-Methoden für die Datenablage in Zwischenablage und Registry wiederverwendet werden können. *TStream* ist ja eine abstrakte Klasse und jede Methode, die in ein *TStream*-Objekt schreibt, weiß nicht, was sich tatsächlich hinter diesem Stream verbirgt (es sei denn, sie ist besonders neugierig und verwendet Abfragen wie *ClassName* oder den *is*-Operator).

TMemoryStream ist eine von *TStream* abgeleitete Klasse, die die von *TStream* definierten Lese- und Schreibmethoden für einen Bereich des Hauptspeichers implementiert. Das Beschreiben eines solchen Streams ist denkbar einfach, denn *TMemoryStream* passt den

Speicherbereich von selbst so an, dass alle Daten hineinpassen. Sie können also munter drauflosschreiben wie in eine Datei.

Am Schluss stellt sich allerdings die Frage, was mit dem Memory-Stream geschehen soll. Wird das Stream-Objekt nämlich freigegeben, gehen die Daten verloren. Zum Kopieren der Daten in die Zwischenablage oder in die Registry benötigen Sie daher direkten Zugriff auf den zugrunde liegenden Speicherbereich des Memory-Streams. Dieser ist über die Properties *TMemoryStream.Memory* (Zeiger auf den Bereich) und *Size* (Größe des Bereichs in Byte) möglich.

Die folgenden Zeilen zeigen ganz allgemein die grundsätzlichen Schritte zur Verwendung eines Memory-Streams:

```
MemoryStream := TMemoryStream.Create;
try
  // Schreiben der Daten:
  MemoryStream.Write(...);
  XYZ.SaveToStream(MemoryStream);
  // Endgültiges Ablegen der Daten (Registry, Zwischenablage,...):
  Lege_Daten_ab(MemoryStream.Memory, MemoryStream.Size);
finally
  MemoryStream.Free;
end;
```

Das Lesen eines Memory-Streams ist nur ein wenig komplizierter, denn nun kommen die Daten z.B. von außen aus der Zwischenablage und müssen erst wieder in den Memory-Stream zurückbefördert werden. Dazu stellen Sie den Memory-Stream *vorher* mit *SetSize* auf die erforderliche Größe und kopieren danach die Daten in seinen Speicherbereich, z.B. mit *Move*. Erst dann können die Daten wie aus jedem anderen Stream ausgelesen werden:

```
MemoryStream := TMemoryStream.Create;
try
  MemoryStream.SetSize(GetDataSize(Datenblock));
  Move(Datenblock, MemoryStream.Memory^, MemoryStream.Size);
  MemoryStream.Read(...);
  XYZ.ReadFromStream(MemoryStream);
finally
  MemoryStream.Free;
end;
```

In diesem Buch finden Sie drei konkrete Beispiele zur Anwendung eines Memory-Streams, bei denen Sie auch »real ausführbaren Code« für das Kopieren der Speicherbereiche und das Lesen/Beschreiben des Streams vorfinden:

- Kapitel 1.9.4, R**117,** Seite 166: Speichern von Programmdaten in der Registry (CD: `Kapitel1\Wecker3d.dpr`)

- Kapitel 5.8.2, R**28**, Seite 763: Speichern von Docking-Layouts in der Registry (TreeDesigner, Unit *MDIForm*)
- Kapitel 8.1.2, R**68**, Seite 1038: Kopieren von Programmdaten in die Zwischenablage (TreeDesigner, Unit *DocForm*).

Suchen Sie in den letzten beiden Units einfach nach *TMemoryStream* - die Namen der Methoden können sich ändern und werden daher nicht hier genannt.

4.4 Grafikausgabe

Auch auf dem Gebiet der Grafikprogrammierung nimmt die VCL dem Entwickler viel Arbeit ab und verschont ihn völlig von der Windows-Schnittstelle, in diesem Fall von der Schnittstelle des GDI (Graphical Device Interface). Dies war ein großer Fortschritt gegenüber Borland Pascal, dessen *ObjectWindows*-Bibliothek hierfür überhaupt keine Klassen zur Verfügung stellte.

Grafikausgabe zur richtigen Zeit

Dieses Kapitel beschreibt die Komponenten der VCL, mit denen Sie den größten Teil der Grafikausgabe durchführen können. In der Praxis können Sie zwar so gut wie überall in der Anwendung Methoden zur Grafikausgabe aufrufen, in den meisten Fällen ist dies jedoch nur in Zusammenhang mit dem *OnPaint*-Ereignis sinnvoll. Wie Sie dieses Ereignis indirekt erzeugen, wenn sich die Grafik geändert hat, sowie Weiteres zu der Steuerung der Grafikausgabe lesen Sie in Kapitel 5.5; in diesem Kapitel geht es lediglich um die Grafikausgabe als solche.

4.4.1 Die Klasse TCanvas

Zentrales Objekt bei der Grafikausgabe mit der VCL ist immer eine Zeichenfläche der Klasse *TCanvas*. Sie zeichnet sich durch eine sehr einfache Handhabung aus. Während man auf einer niedrigeren Ebene diverse an der Grafikausgabe beteiligten Objekte erst initialisieren und nach dem Zeichnen dann Aufräumarbeiten vornehmen muss, entfällt dies alles mit *TCanvas*. Wenn Sie das richtige Ereignis abwarten – wie z.B. das schon erwähnte *OnPaint* –, können Sie in der Ereignismethode sofort mit dem eigentlichen Zeichnen anfangen, das *Canvas*-Objekt initialisiert sich dann automatisch und soweit erforderlich.

Grafik-Properties und -Methoden

Betrachten wir zunächst ein kleines Beispiel zum Zeichnen einer drei Pixel breiten horizontalen Linie 200 Pixel unterhalb der Titelzeile eines Fensters:

```
with Canvas do begin
  Pen.Width := 3;
  Pen.Color := clRed;
  MoveTo(0, 200);
  LineTo(ClientWidth, 200);
end;
```

In diesem Beispiel zeigen sich zwei Grundprinzipien der Verwendung von *TCanvas*:

- Die Attribute der Grafik (Breite und Farbe der Linien, Farbe zum Füllen etc.) werden eingestellt, indem Sie Properties verändern. Viele Attribute sind den Kategorien Stift und Pinsel zugeordnet, die Properties der Klasse *TCanvas* sind. Um beispielsweise die Stiftfarbe zu verändern, müssen Sie ein darin geschachteltes Property verändern: *TCanvas.Pen.Color.*
- Die eigentliche Grafikausgabe findet in *TCanvas* mit einfachen Methoden statt, die im Namen mit den Funktionen der Windows-Grafikschnittstelle GDI übereinstimmen (z.B. *MoveTo/LineTo*, *Rectangle*, *Ellipse*).

TCanvas-Instanzen

Um diese Properties und Methoden nutzen zu können, benötigen Sie zuerst einmal eine Instanz der Klasse *TCanvas*, die für die Zeichenfläche Ihres Fensters zuständig ist. Sie brauchen eine solche weder selbst zu erzeugen, noch müssen Sie lange nach einer bestehenden Instanz suchen, denn jedes Formular besitzt bereits ein *Canvas*-Property. Sie können also den Code aus dem obigen Beispiel direkt in eine mit dem *OnPaint*-Event verknüpfte Methode oder eine andere Methode Ihres Formulars einsetzen.

Oft soll jedoch nicht die gesamte Fläche des Formulars für die Grafikausgabe verwendet werden, weil auch noch Platz für einige andere Komponenten benötigt wird. Manchmal nimmt die Grafik auch nur einen besonders kleinen Teil des Formulars ein, z.B. in Dialogboxen, die in einer kleinen Vorschaugrafik anzeigen, wie sich die gerade eingestellten Optionen auswirken. In diesen Fällen ist es sinnvoller, auf das *Canvas*-Objekt des Formulars zu verzichten und die Grafikausgabe auf eine eigene Komponente zu beschränken.

Folgende Komponenten verfügen bereits über ein *Canvas*-Property:

- *TPaintBox* sollten Sie verwenden, um die Grafik auf einen Teil des Formulars zu beschränken. Ihr einziger Zweck ist es, eine Zeichenfläche zur Verfügung zu stellen; ihr besonderer Vorteil ist, dass sie nicht nur kleiner, sondern auch größer als das Formular sein kann, was sich zum Scrollen ausnutzen lässt (siehe Kapitel 5.5.4).
- *TPrinter* stellt das Papier, Folien, T-Shirts oder was immer bedruckt wird, als *TCanvas*-Objekt dar, das Sie grundsätzlich so behandeln können wie jedes andere *Canvas*-Objekt auch. Beachtenswerte Details und ein Beispiel finden Sie in Kapitel 5.6.4.

- *TBitmap*, die Klasse für Pixelgrafiken (Bitmaps), stellt eine Zeichenfläche zur Verfügung, die nicht unbedingt am Bildschirm sichtbar sein muss; mehr über *TBitmap* lesen Sie in Kapitel 4.5.2.
- Auch Dateien können das Ziel einer Zeichenoperation sein, und zwar im Falle der Metadateien, die allerdings in einem Windows-spezifischen Format gespeichert werden. In Delphi werden Metadateien von der Klasse *TMetaFile* gekapselt. Auch wenn diese nicht direkt über ein *Canvas*-Property verfügt, soll sie in dieser Aufzählung nicht fehlen, denn sie verwendet die Klasse *TCanvas* auf eine dem *Canvas*-Property ähnliche Weise, siehe hierzu Kapitel 4.5.1.
- In vielen Komponenten, die sich normalerweise selbst zeichnen, können Sie die Kontrolle über die Grafikausgabe übernehmen, so z.B. in *TListBox*, *TListView*, *TComboBox*, *TStringGrid*, *TTabSet* und *THeaderControl*. Auch bei diesen zeichnen Sie mit einem *Canvas*-Property, müssen dabei aber die Grafikausgabe selbst auf die zu zeichnende Zeile bzw. Zelle des Steuerelements beschränken. Ein Beispiel für eine selbst gezeichnete Listbox finden Sie in den Beispielprogrammen *OwnerDraw1* und *OwnerDraw2*, die in Kapitel 4.5.4 besprochen werden; eine selbst gezeichnete *TDrawGrid*-Komponente im Programm *DrawGrid* (siehe Grafikbeispiel in Kapitel 4.4.3).
- Wenn Sie in selbst geschriebenen Komponenten eine Grafik zeichnen möchten, leiten Sie die Komponente entweder von *TGraphicControl* oder von *TCustomControl* ab, je nachdem, ob die Komponente mit einem Windows-Fenster verbunden sein soll. Beide Klassen definieren gleichermaßen ein geschütztes *Canvas*-Property, das nur in bestimmten Klassen wie *TImage* als *public* deklariert wird.
- Der Vollständigkeit halber soll auch das Formular als Besitzer eines *Canvas*-Objekts nicht in dieser Aufzählung fehlen.

Wenn Sie nicht das *Canvas* des Formulars, sondern das einer Komponente verwenden, haben Sie den Vorteil, dass die Grafikausgabe am Rand der Komponente abgeschnitten wird (was als *Clipping* bezeichnet wird) und dass das Koordinatensystem relativ zur Komponente und nicht relativ zum Formular ist (die Koordinate (0/0) befindet sich also in der linken oberen Ecke der Komponente bzw. von deren Arbeitsbereich). Durch diese Unabhängigkeit vom Formular ist es auch leicht möglich, die gesamte Grafikausgabe an eine andere Stelle des Formulars zu verschieben, ohne die Methodenaufrufe zur Grafikausgabe zu verändern.

Für die Kenner der Gerätekontexte und Handles, wie sie auf Systemebene vorkommen, sei noch einmal betont, dass dieses *Canvas*-Property wie alle anderen Properties funktioniert. Es steht zur Verfügung, solange sein Besitzer, in diesem Fall also das Formular, existiert. Das bedeutet, dass Sie jederzeit Attribute verändern oder zeichnen

können. Die gesetzten Attribute, wie z.B. die Breite des Stifts, bleiben dauerhaft gültig, obwohl die zugehörigen GDI-Objekte zwischendurch freigegeben werden können.

Canvas-Elemente

Die wichtigsten Properties von *TCanvas* sind die im nächsten Kapitel beschriebenen Zeichenwerkzeuge *Pen*, *Brush* und *Font*. Den Methoden zur Grafikausgabe widmet sich das darauf folgende Kapitel. Wichtig ist auch noch das Property *Handle*, mit dem Sie GDI-Funktionen aufrufen können, wie in Kapitel 5.5.5 beschrieben.

Die wenigen weiteren Elemente der Klasse *TCanvas* sind eher selten verwendbar. So informieren Sie z.B. die beiden *TCanvas*-Events darüber, dass die Zeichenfläche gleich verändert wird (*OnChanging*), oder darüber, dass sie gerade verändert wurde (*OnChange*). Beide Ereignisse werden aufgerufen, wenn Sie Grafik mit einer der *TCanvas*-Methoden ausgeben. Der manchmal notwendige Aufruf von GDI-Funktionen mit Hilfe des Canvas-Handles geht allerdings teilweise an *TCanvas* vorbei, so dass diese beiden Ereignisse nicht hundertprozentig verlässlich sind (teilweise darum, weil das Lesen des *Handle*-Properties zwar zu einem *OnChanging*-Ereignis führt, durch den direkten Aufruf einer GDI-Methode aber kein *OnChange*-Ereignis erzeugt wird).

4.4.2 Zeichenwerkzeuge

Das Beispiel des letzten Abschnitts hat gezeigt, wie leicht es ist, grafische Objekte auszugeben. Die Methoden, die diese Objekte zeichnen, erwarten normalerweise nur die Koordinaten der Objekte. Die anderen Angaben wie Farbe, Füllmuster und Zeichenmodus werden als Eigenschaften der Zeichenfläche behandelt. Das hat den Vorteil, dass Sie nicht jedes Mal alle Attribute als Methodenparameter angegeben müssen, wenn Sie bei vielen aufeinander folgenden Grafikbefehlen immer dieselben Attribute verwenden.

Damit bei der großen Zahl der Zeichenattribute die Übersichtlichkeit nicht verloren geht, sind einige davon zu den Zeichenwerkzeugen Stift und Pinsel gruppiert worden; die Gruppierung der Schriftattribute zu Schriftarten ist wohl selbstverständlich. Auch die Schriftarten sollen hier als »Zeichenwerkzeuge« behandelt werden.

Die VCL besitzt natürlich auch für jeden Zeichenwerkzeugtyp eine eigene Klasse und für diese drei ähnlichen Klassen eine Basisklasse: *TGraphicsObject*. Diese weist jedoch als einziges neues und öffentliches Element nur das Ereignis *OnChange* auf, das bei jeder Veränderung eines der Properties, die in den abgeleiteten Klassen hinzugefügt werden, eintritt. Weitere Neuigkeiten gibt es erst bei den drei speziellen Zeichenwerkzeug-Klassen.

Stifte

Ein Stift (Klasse *TPen*) ist ein Zeichenwerkzeug mit den vier Eigenschaften *Color*, *Width*, *Style* und *Mode*: Dem *Color*-Property können Sie einen der 16,7 Millionen Farbwerte oder eine der Systemfarben (z.B. *clActiveCaption*) zuweisen und in *Width* geben Sie an, wie viele logische Einheiten (per Voreinstellung sind dies die Pixel) die vom Stift gezogene Linie breit sein soll. Eine Breite von 0 ist möglich, meint aber in jedem Fall die Breite eines Pixels, unabhängig davon, wie breit eine logische Einheit ist.

Um wirklich die Breite 0, also eine unsichtbare Linie zu erhalten, müssen Sie den Stil des Stifts auf *psClear* setzen. Die anderen möglichen Stile sind *psSolid*, *psDash*, *psDot*, *psDashDot* und *psDashDotDot* und sorgen für verschiedene Arten von gestrichelten und gepunkteten Linien. Sie können diese Stile einfach ausprobieren, indem Sie eine *TShape*-Komponente (Seite *Zusätzlich*) auf ein Formular setzen und das Property *Pen.Style* auf den gewünschten Stil einstellen.

Ein besonderer Stil ist *psInsideFrame*. Er bewirkt, dass die Linie, falls sie ein grafisches Objekt wie ein Rechteck oder eine Ellipse umgibt, nicht außerhalb der Koordinaten dieses Objekts gezeichnet wird. Ohne diesen Stil wird eine mehr als ein Pixel breite Linie zur Hälfte innerhalb, zur Hälfte außerhalb der eigentlichen Objektbegrenzung gezeichnet.

Stifte, die breiter als ein Pixel sind, können nur in den Stilen *psSolid* und *psClear* zeichnen, demnach ist es auch kein zusätzlicher Verlust, wenn *psInsideFrame* nur durchgezogene Linien darstellen kann.

Die Zeichenfunktionen verwenden den aktuellen Stift (*TCanvas.Pen*) für die äußeren Linien von Flächen wie Rechtecken, Ellipsen und Polygonen und natürlich für einzelne Linien selbst. Stiftfarbe, -breite, -stil und -modus sind somit einerseits vier von vielen Attributen der Zeichenfläche, andererseits sind sie als eigenständiges *TPen*-Objekt manipulierbar.

TPen.Mode

Das vierte Attribut verdient eine etwas eingehendere Behandlung, da es nicht zum Stift, wie ihn Windows definiert, gehört und von Kennern der Windows-Programmierung vielleicht an anderer Stelle gesucht werden könnte.

Anders als reale Stifte können die Stifte von grafischen Benutzeroberflächen die Zeichenfläche perfekt abdecken, ohne sich mit der vorher da gewesenen Farbe zu vermischen oder diese durchschimmern zu lassen. Dies ist jedoch nur die Voreinstellung: In *Mode* können Sie neben dem Standard *pmCopy* 15 verschiedene andere Zeichenmodi einstellen, die jeweils eine andere Kombination von Stiftfarbe und Farbe der Zeichenfläche bewirken.

Diese Modi können allerlei Effekte verursachen oder die additive Farbmischung von Wasserfarben simulieren, in den meisten Fällen wird jedoch ein ganz bestimmter Modus benötigt: Wenn Sie zwei gleiche Linien im *pmNotXor*-Modus übereinander zeichnen, heben sich beide gegenseitig auf, so dass sich dieser Modus besonders gut dafür eignet, vorläufige Linien bei Mausziehoperationen zu zeichnen (siehe Kapitel 5.4).

In den anderen Modi können Sie Stift und Zeichenfläche auch mit AND und OR verknüpfen und optional das Ergebnis mit NOT umkehren. Mit den Modi *pmBlack* und *pmWhite* bringen Sie Stifte aller Farben dazu, schwarz bzw. weiß auf jedem Untergrund zu zeichnen.

Wichtig ist, dass *Pen.Mode* eigentlich nicht dem Stift als Attribut zuzurechnen ist, denn es wirkt sich auf *alle* Zeichenvorgänge aus, insbesondere also auch auf das, was eigentlich mit dem »Pinsel« gezeichnet wird. Sie können dies bereits mit der schon erwähnten *TShape*-Komponente ausprobieren. Zeichnen Sie zwei sich teilweise überlappende *Shapes* in ein Formular ein und verändern Sie *Color* und *Mode* der oberen Komponente, um die Effekte zu beobachten.

Farben

Für die Farben können Sie dieselben Werte und vorgegebenen Farbkonstanten wie im Objektinspektor in den *Color*-Properties der Steuerelemente und des Formulars (z.B. *clBlack, clWhite, $808080*) verwenden. Ausgewählte Farbtöne aus den 16,7 Millionen möglichen Werten müssen Sie nicht direkt angeben, sondern Sie können sie von der Funktion *RGB* »berechnen« lassen. Sie übergeben *RGB* drei Werte für den Rot-, Grün- und Blau-Anteil der Farbe, die jeweils zwischen 0 und 255 liegen dürfen, und erhalten von *RGB* einen 4 Byte großen Farbwert des Typs *TColor* für die verschiedenen *Color*-Properties:

```
Pen.Color := RGB(255, 120, 80); { maximales Rot, 120 grün, 80 blau }
```

Der Farbwert besteht intern aus einem Steuerbyte und je einem Byte für die drei Grundfarben. In Grafikmodi mit 256 oder weniger Farben verwendet Windows Mischfarben, falls Sie eine Farbe angeben, die nicht zur Systempalette oder einer selbst definierten Palette gehört. Wenn Sie eigene Paletten definieren, können Sie statt *RGB* auch die Funktionen *PaletteRGB* und *PaletteIndex* verwenden (hier kommt dann das »Steuerbyte« des Farbwerts ins Spiel). Zur Definition von eigenen Farbpaletten benötigen Sie Funktionen des Windows-API, die Sie in der Online-Hilfe zum Windows-API finden. Da heutige Grafikkarten wohl meist in ergonomischen Farbmodi mit mindestens 32767 Farben arbeiten, in denen es keine Palette mehr gibt, wenden wir uns gleich den Pinseln zu.

Pinsel

Mit Pinseln füllt Windows das Innere der gezeichneten Formen. Die Attribute eines Pinsels sind Farbe, Stil und Bitmap, für die jeweils ein Property zuständig ist:

- In der Farbe *Color* geben Sie wie bei *TPen.Color* einen *TColor*-Farbwert an.
- In *Style* wählen Sie eines der vordefinierten Muster aus: *bsSolid* für vollständige Füllung mit der Farbe, *bsClear* zum Abschalten der Füllung, *bsBDiagonal*, *bsFDiagonal*, *bsHorizontal*, *bsVertical*, *bsCross* und *bsDiagCross* für schräg ansteigende, schräg absteigende, horizontale und vertikale Linien sowie für zwei verschiedene Arten der Schraffur (auch diese Stile können Sie mit *TShape*-Komponenten ausprobieren, da *TShape* auch über ein Property *Brush.Style* verfügt).
- Falls diese vorgegebenen Muster nicht reichen, können Sie im Property *Bitmap* ein *TBitmap*-Objekt angeben, das eine 8*8 Pixel große Bitmap enthält. Dieses wird dann anstelle der Farbe zum Füllen der Fläche verwendet. Zu *TBitmap* siehe Kapitel 4.5.2.

Schriftarten

Anders als Stift und Pinsel handelt es sich bei den Schriftarten nicht nur um eine Sammlung von Attributen der Zeichenfläche, sondern um Objekte, die in eigenen Dateien definiert werden (z.B. in TTF-Dateien für TrueType-Schriften), wobei die Dateien für die Programmierung keine Rolle spielen.

Die Klasse *TFont* beinhaltet eine vereinfachte Beschreibung einer Schriftart in Form der folgenden Properties:

Property	**Typ**	**Bedeutung**
Color	TColor	Farbe der Schrift
Height	Integer	Schrifthöhe in Pixeln
Name	TFontName	Name der Schriftart
Pitch	Integer	legt mit den Werten *fpDefault*, *fpVariable*, *fpFixed* fest, ob die Abstände der Zeichen so groß sein sollen, wie von der gewünschten Schriftart vordefiniert, oder ob Windows variablen ('I' ist schmaler als 'M') oder festen ('I' ist genauso breit wie 'M') Zeichenabstand einhalten und dafür ggf. die Schriftart wechseln soll.
PixelsPerInch	Integer	wird von der VCL intern verwendet, um *Height* in *Size* umzurechnen und umgekehrt.
Size	Integer	Höhe der Schrift in Punkten
Style	TFontStyle	eine Menge, die die Werte *fsBold*, *fsItalic*, *fsUnderline* und *fsStrikeOut* enthalten kann (fett, kursiv, unterstrichen und durchgestrichen)

Die meisten Eigenschaften der Schriftart sind stets auch in den Dialogen zur Schriftauswahl anzutreffen, einen von ihnen stellt der Objektinspektor als Property-Editor für die *Font*-Properties der verschiedenen Komponenten zur Verfügung.

Die API-Funktionen von Windows zur Definition einer Schriftart (z.B. *CreateLogFont*) arbeiten mit einigen weiteren Schriftattributen, jedoch sind diese dazu bestimmt, aus einer Reihe unbekannter Schriftarten eine auszuwählen, die bestimmte Kriterien (die zusätzlichen Attribute) möglichst gut erfüllt. Solange Sie wissen, wie die Schrift heißt, die Sie benötigen, genügen die Properties von *TFont*. Oft darf ohnehin der Benutzer die Schriftart mit einer Dialogbox auswählen.

Mit mehreren Zeichenwerkzeugen arbeiten

Um die Stiftattribute Ihres *Canvas*-Objekts zu setzen, haben Sie zwei Möglichkeiten: Normalerweise gehen Sie so vor wie im obigen Beispiel und setzen die Properties von *Canvas.Pen*, *Canvas.Brush* und *Canvas.Font* einzeln. Da die Objekte *Pen*, *Brush* und *Font* jedoch selbst Properties sind, können Sie sie auch als Ganzes neu setzen.

Sie können z.B. dynamisch neue Stift-Objekte erstellen und diese dem Property *Canvas.Pen* zuweisen. Das folgende Beispiel erzeugt ein dynamisches *TPen*-Objekt:

```
NewPen := TPen.Create;
... { Einstellen der Properties }
Canvas.Pen := NewPen;
...
Pen.Free;
```

In diesem Fall sind Sie lediglich dafür verantwortlich, das mit *Create* erstellte Objekt wieder mit *Free* (oder *Destroy*) zu löschen. Sie können es direkt löschen, nachdem Sie es dem Property *Canvas.Pen* zugewiesen haben, ohne *Canvas.Pen* dadurch in irgendeiner Form zu beeinträchtigen.

Auch wenn Sie wie in diesem Beispiel Stifte, Pinsel oder Schriften dynamisch erzeugen, geht die VCL verantwortungsvoll mit den empfindlichen Windows-Ressourcen um: So reserviert sie die entsprechenden Windows-Zeichenobjekte nur bei Bedarf, was zur Folge hat, dass Sie problemlos 10000 verschiedene Stifte als *TPen* reservieren können. (Die reine Erstellung dieser 10000 Stifte verbraucht ca. 200 KByte Speicher, aber keine GDI-Ressourcen.)

Ein praktisches Beispiel für die dynamische Erzeugung von Schrift-, Pinsel- und Stiftobjekten gibt der TreeDesigner – jedes seiner Grafikobjekte enthält ein Exemplar jedes dieser Objekttypen (siehe Kapitel 5.3.3).

Alte Canvas-Einstellungen wiederherstellen

Manchmal ist es wichtig, die Änderungen an den *Brush-*, *Pen-* und *Font*-Einstellungen einer Zeichenfläche wieder rückgängig zu machen. Zum Beispiel, wenn eine Zeichenfläche durch mehrere Methoden gezeichnet wird, was etwa bei besitzergezeichneten Komponenten vorkommen kann, wenn der Hintergrund der Komponente von der Komponente selbst gezeichnet wird und die einzelnen Einträge in einer von Ihnen geschriebenen Ereignisbearbeitungsmethode. Wenn Sie hier in Ihrer Methode z.B. einen anderen Pinsel einstellen, wirkt sich das auch auf die weitere Grafikausgabe der Komponente aus.

Um die Einstellungen der Grafikwerkzeuge auf einfache und standardisierte Weise zu sichern, können Sie die *TRecall*-Klassen *TBrushRecall*, *TPenRecall* und *TFontRecall* verwenden (ab Delphi 6). Objekte dieser Klassen speichern bei ihrer Erzeugung den aktuellen Zustand eines Zeichenwerkzeugs und stellen ihn bei ihrer Freigabe wieder her. Zusammen mit einem *try...finally*-Block können Sie so sicherstellen, dass Ihre Methode nach außen hin keine Nebeneffekte auf ein bestimmtes Zeichenwerkzeug hat. Ein Musterbeispiel zur Verwendung von *TBrushRecall* (die beiden anderen Klassen funktionieren analog dazu):

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
var
  PrevBrush: TBrushRecall;
begin
  PrevBrush := TBrushRecall.Create(Canvas.Brush);
  try
    // hier der Code, der den Pinsel verändert:
    Canvas.Brush.Color := clBlack;
    ...
  finally
    PrevBrush.Free; // Anfangszustand wiederherstellen
  end;
end;
```

4.4.3 Grafikmethoden

Die folgende Tabelle fasst die einfachen Methoden von *TCanvas* zur Grafikausgabe zusammen:

Methode	**Parameter**	**Beschreibung**
MoveTo	x, y	bewegt den Stift an eine neue Position, ohne zu zeichnen.
LineTo	x, y	zeichnet von der aktuellen Position bis zu (x, y) und setzt die aktuelle Stiftposition auf diesen Wert (der Punkt [x, y] selbst wird nicht gezeichnet).
Rectangle	x1, y1, x2, y2	zeichnet Quadrate und Rechtecke.

Methode	Parameter	Beschreibung
RoundRect	x1, y1, x2, y2, x3, y3	rundet die Ecken der Rechtecke mit Ellipsenvierteln ab. Die Größe der Ellipse, die geviertelt an die Rechteckränder verteilt werden soll, geben Sie in den Parametern *x3* und *y3* (Breite und Höhe) an.
Ellipse	x1, y1, x2, y2	zeichnet Kreise und Ellipsen.
Arc Cord Pie	x1, y1, x2, y2, x3, y3, x4, y4	zeichnen Ellipsenbögen, »abgeschnittene« Ellipsen und »Tortenstücke«.
PolyBezier, PolyBezierTo	array of TPoint	zeichnen eine Folge von Bézier-Kurven. Der Array-Parameter besteht aus einer Folge von Punkten, wobei sich hier jeweils ein Stützpunkt und zwei Kontrollpunkte abwechseln.
Polygon, Polyline	array of TPoint	zeichnen Linienfolgen und gefüllte Polygone.
TextOut	x, y, Text	Textausgabe.
TextRect	TRect, x, y,	Textausgabe innerhalb eines Rechtecks.

Linien zeichnen

Um eine einzelne Linie zu zeichnen, benötigen Sie zwei Methodenaufrufe: Zuerst setzen Sie den Stift mit *MoveTo* auf den Beginn der Linie und zeichnen die Linie dann mit *LineTo*. Genau genommen zeichnet *LineTo* jedoch den Endpunkt nicht mit, damit dieser, wenn er bei der nächsten Linie wieder als Startpunkt verwendet wird, nicht doppelt gezeichnet wird (was in bestimmten *Pen.Mode*-Einstellungen durchaus sichtbar werden kann).

Vorteilhaft wirkt sich die Trennung des Linienziehens in *LineTo* und *MoveTo* erst aus, wenn Sie mehrere verbundene Linien zeichnen. Hierfür bietet sich auch die gleich besprochene Methode *PolyLine* an.

Formen zeichnen

Zum Zeichnen von Rechtecken und Ellipsen genügt der Aufruf einer einzigen der in der obigen Tabelle gezeigten Methoden. Als Parameter geben Sie jeweils ein Rechteck an, in dem sich die Form befinden soll. Ähnlich wie bei den Linien werden auch hier nicht alle Koordinaten in die Zeichnung eingeschlossen: die Koordinaten *x2*/*y2* gehören nicht mehr zum gezeichneten Rechteck.

Die Ellipsenfunktion ist zwar durch die Rechteck-Koordinaten sehr einfach zu bedienen, hat aber auch einen Nachteil: Sie kann keine »schrägen« Ellipsen zeichnen, also solche, die entstünden, würde man die einfachen Ellipsen um weniger als 90 Grad drehen. Um solche Ellipsen zu erhalten, müssten Sie auf Windows-API-Ebene mit Koordinatentransformationen arbeiten, die aber nur unter Windows NT/2000, nicht

aber in der Windows 9x-Linie zur Verfügung stehen. (Auch die CLX stellt Ihnen die entsprechenden Funktionen unter Windows 9x zur Verfügung.)

Alle Funktionen zum Zeichnen von Formen bedienen sich des gerade aktiven Stifts und füllen die Fläche außerdem mit dem gewählten Pinsel.

Arc, Chord und Pie

Ellipsen spielen auch bei drei weiteren Methoden eine Rolle. Beispielhafte Werke der Methoden *Arc*, *Chord* und *Pie* sehen Sie in Abbildung 4.2 (das Programm heißt Canvas-Magnetism). *Arc* ist eine reine Linienfunktion und zeichnet einen Abschnitt der Ellipsenumrandung. *Chord* arbeitet so, als teilte sie die Fläche einer Ellipse mit einer Geraden in zwei Teile, von denen nur einer gezeichnet wird. *Pie* schneidet ein mehr oder weniger großes Tortenstück aus der Ellipse und zeichnet einen der beiden daraus entstehenden Flächenteile.

Die ersten vier Parameter aller drei Funktionen geben das umgebende Rechteck der grundlegenden Ellipse an. Dazu gesellen sich vier weitere Parameter, die zwei Punkte angeben, aus denen sich der Winkel errechnen lässt, der gezeichnet werden soll. Dies geschieht folgendermaßen: Von jedem der beiden Punkte berechnen die Funktionen eine Gerade, die die Ellipse teilt und dabei durch den Mittelpunkt geht. Zwei dieser Geraden teilen die Ellipse in vier Teile. Die Funktionen bewegen sich dabei gegen den Uhrzeigersinn von der ersten zur zweiten Geraden und zeichnen diesen Abschnitt. (Im Programm *CanvasMagnetism* aus Abbildung 4.2 können Sie den Punkt (x3 / y3) mit der linken, den Punkt (x4 / y4) mit der rechten Maustaste festlegen.)

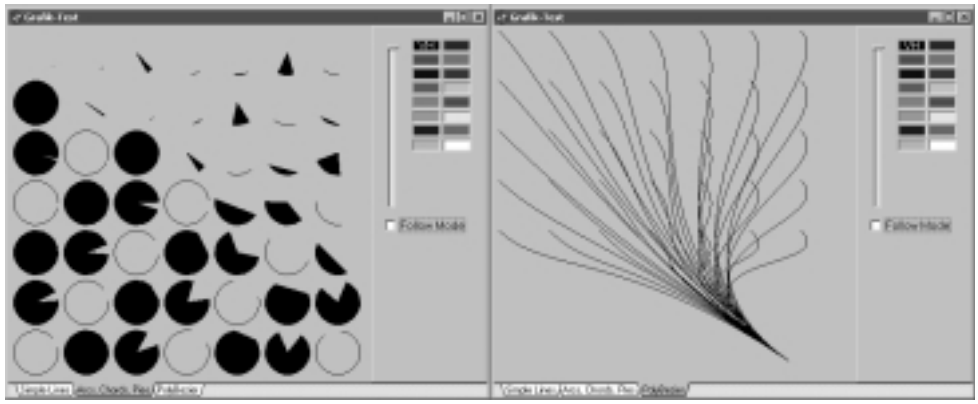


Abbildung 4.2: Beispielprogramm CanvasMagnetism. Linkes Fenster, unterste Reihe von links nach rechts: Pie, Arc und Chord. Rechts: Bézier-Kurven. Zum Verständnis des Programmnamens müssen Sie die Maus über die verschiedenen Seiten des Fensters bewegen.

Auch diese drei Funktionen bedienen sich des gerade aktiven Stifts, *Chord* und *Pie* füllen die Fläche außerdem mit dem gewählten Pinsel.

Komplexere Objekte

Die Methoden *Polyline* und *Polygon* erlauben es schließlich, auch Formen mit komplizierterer Umrandung zu zeichnen, solange diese aus geraden Linien besteht. Diese Linien werden als Array von Punkten (*TPoint*) angegeben, so als würden die Linien mit einem *MoveTo* und vielen aufeinander folgenden *LineTo*-Befehlen gezeichnet werden. Genau das tut *Polyline* auch, während *Polygon* das entstandene Konstrukt auch noch ausfüllt und dabei den Endpunkt der letzten Linie mit dem Anfangspunkt der ersten Linie verbindet.

Der folgende Ausschnitt zeichnet mit *PolyLine* ein nicht ausgefülltes Dreieck, wobei das Array direkt angegeben ist:

```
Canvas.Polyline([Point(90, 10), Point(210, 10), Point(150, 120)]);
```

Polygon wird im Folgenden dazu verwendet, ein Sechseck zu zeichnen, dessen Berechnung z.B. so aussehen könnte wie in der Methode *PaintHexagon* des TreeDesigners (siehe Kapitel 5.5.2):

```
var
  Points: array[0..5] of TPoint;
begin
  { Berechnung von Points }
  ...
  Canvas.Polygon(Points);
```

Die Methoden *PolyBezier* und *PolyBezierTo* werden ähnlich aufgerufen. *PolyBezier* benötigt zumindest vier Punktangaben: den Startpunkt, zwei Kontrollpunkte sowie einen Endpunkt. Für jeden weiteren Stützpunkt verlängern Sie diese Liste um jeweils zwei Kontrollpunkte, gefolgt von dem jeweiligen Stützpunkt. Das Beispielprogramm *CanvasMagnetism* zeichnet eine Reihen von einfachen Linien mit nur zwei Stützpunkten (siehe Abbildung 4.2).

Textausgabe

Wenn Sie die Methode *TextOut* einfach so verwenden, setzt sie die linke obere Ecke des Textes an die in den Parametern übergebene Position. Der Hintergrund des Textes wird mit dem aktuellen Pinsel gefüllt. Falls dieser jedoch nicht den Stil *bsSolid* hat, wird kein Hintergrund gezeichnet, der bisherige Hintergrund bleibt also unter dem Text sichtbar (aus der Sicht des GDI wird der Text dann im Hintergrundmodus *transparent* gezeichnet).

In vielen Fällen ist die Textausgabe nicht ganz so einfach: So soll z.B. oft die Ausrichtung des Textes berücksichtigt werden, und sofern Sie keine Schrift mit festem Zeichenabstand verwenden, ist es oft interessant, den Platzbedarf einer Zeichenkette zu erfahren.

Für letztere beiden Aufgaben besitzt *TCanvas* weitere Methoden: Mit *TextWidth* erhalten Sie die Breite, die der String mit der aktuellen Schriftart (*TCanvas.Font*) benötigen würde, mit *TextHeight* die Höhe. Beide Funktionen geben Abmessungen in virtuellen Koordinaten zurück, per Voreinstellung also in Pixeln.

Die folgende Zeile berechnet mit *TextHeight* die Y-Position *TopPos*, an die ein String geschrieben werden muss, damit seine vertikale Mitte in der Höhe *VMiddle* liegt:

```
TopPos := VMiddle-CanvasRef.TextHeight('A') div 2;
```

Dazu zieht die Anweisung die halbe Texthöhe von der gewünschten Mitte ab. Dieses Beispiel ist der Textausgabe aus Kapitel 5.5.2 entnommen, einem Kapitel, das auch zeigt, wie Sie andere Ausrichtungen des Textes erreichen.

Die zweite Textausgabemethode von *TCanvas* ist *TextRect*. Sie (bzw. die von ihr aufgerufene GDI-Funktion *ExtTextOut*) führt automatisches Clipping des Textes an einem im ersten Parameter angegebenen Rechteck durch und verhält sich ansonsten wie *TextOut* (Beispiel siehe Kapitel 5.5.5).

Weitere TCanvas-Methoden

Zu den restlichen Methoden gehören weitere Variationen zum Zeichnen eines Rechtecks, eine Methode zum Füllen einer Fläche und Methoden, die Grafiken oder Bitmaps zeichnen. Die folgende Übersicht verzichtet zugunsten der Online-Hilfe auf Parameterbeschreibungen:

Methode	**Beschreibung**
DrawFocusRect	zeichnet einen Rechteck-Umriss im XOR-Modus (siehe Pen.Mode, *pmNotXor*).
FillRect	zeichnet die Füllung eines Rechtecks – ohne Umriss.
FrameRect	zeichnet den Umriss eines Rechtecks *mit dem Pinsel* – ohne Füllung.
FloodFill	füllt eine Fläche, in der ein gegebener Punkt liegt, mit einer Farbe aus.
BrushCopy	füllt eine Fläche mit einem Teil eines *TBitmap*-Objekts unter Verwendung eines Pinsels zum Zeichnen transparenter Bereiche.
CopyRect	kopiert ein Rechteck aus einem anderen *TCanvas*-Objekt auf die Zeichenfläche.
Draw	zeichnet ein *TGraphic*-Objekt an eine bestimmte Position (Beispiel in Kapitel 4.5.4).
StretchDraw	füllt ein Rechteck mit einem *TGraphic*-Objekt aus (Beispiel in Kapitel 4.5.4).

Pixeloperationen

In einem *TCanvas*-Objekt können Sie auch einzelne Pixel lesen und setzen. *TCanvas* macht die Pixel in einem indizierten Property zugänglich, das sehr einfach anzusprechen ist. Um die Farben an Position *x* / *y* abzufragen und zu ändern, genügen beispielsweise die folgenden Zeilen:

```
FarbeAnPosition := Pixels[x, y];
Pixels[x, y] := NeueFarbe;
```

Die Indizes des *Pixels*-Arrays haben übrigens eine Breite von 32 Bit, auch wenn das 16-Bit-GDI von Windows 95 / 98 hier nur 16-Bit-Werte unterstützt.

Pixeloperationen sind jedoch nicht gerade ein Vorbild in Sachen Geräteunabhängigkeit. Einerseits weisen die Pixel auf verschiedenen Gerätetypen wie Monitoren und Druckern gravierende Größenunterschiede auf, andererseits gibt es Geräte, die gar keine Pixel kennen, wie z.B. die Plotter. Dennoch kann das *Pixels*-Property sehr nützlich sein, wenn Sie es beim *Canvas* eines *TBitmap*-Objekts verwenden. Auch werden Sie statt der VCL einmal die CLX verwenden sollten, müssen Sie dort bei *TCanvas* auf ein *Pixels*-Array verzichten.

Weitere GDI-Funktionen

Das GDI weist einige weitere wichtige Funktionsbereiche auf, die in *TCanvas* nicht durch Methoden repräsentiert sind. In diesen Fällen müssen Sie die API-Funktionen des GDI direkt aufrufen und diesen das Handle Ihres *Canvas*-Objektes übergeben. Auf diese Weise können Sie zum Beispiel

- unterschiedliche Abbildungsmodi (virtuelle bzw. logische Koordinatensysteme) einstellen (siehe Kapitel 5.6.2),
- eigenes Clipping durchführen (siehe Kapitel 5.5.5) und
- mehr Kontrolle über die Textausgabe ausüben (z.B. mit *ExtTextOut* oder mit *SetTextAlign*, für ein Beispiel zu letzterer Funktion siehe Kapitel 5.5.2).

Informationen zu allen GDI-Funktionen erhalten Sie in der Win32-Online-Hilfe. Die meisten GDI-Funktionen sind bereits in den Methoden und Properties der VCL gekapselt. Wenn Sie einige GDI-Funktionen direkt aufrufen, können Sie die VCL sogar durcheinander bringen, z.B. wenn Sie mit *SelectObject* per Hand ein neues Zeichenwerkzeug wählen, während das *Canvas*-Objekt annimmt, dass noch das von ihm gewählte Zeichenwerkzeug aktiv ist.

4.4.4 Besitzergezeichnete Komponenten

Da dieses Buch mit dem TreeDesigner aus Kapitel 5 über eine umfangreiche Beispielanwendung verfügt, die Grafiken im Arbeitsbereich von Fenstern ausgibt, sollen sich die Beispiele für die Grafikausgabe in diesem Kapitel eines anderen Gebiets annehmen: dem der besitzergezeichneten Steuerelemente. Unter »Besitzer« kann man zweierlei verstehen: das Formular, das die Methoden zum Zeichnen eines Steuerelements enthält, oder eben die Person, die das Formular programmiert.

Grundsätzliche Vorgehensweise

Viele Komponenten der VCL bieten neben ihrem Standardaussehen und einigen Properties, mit denen Sie dieses beeinflussen können, die Möglichkeit, die gesamte Komponente oder Teile davon selbst zu zeichnen. Auch dieser Prozess beruht wieder auf der Ereignisorientierung: Die Komponenten senden Ihnen eine Nachricht, wenn das Neuzeichnen erforderlich ist, und Sie stellen eine Methode bereit, die auf diese Nachricht reagiert. Während sich Zeichenflächen (*TPaintBox*, siehe Kapitel 5.5.2) bei einem *OnPaint*-Ereignis neu zeichnen, enthalten die Namen der Ereignisse von benutzergezeichneten Komponenten normalerweise den Begriff *OnDraw*, z.B. *OnDrawItem* oder *OnDrawCell*, *OnDrawTab*, *OnCustomDraw* oder *OnAdvancedCustomDrawItem*. Typisch für diese Ereignisse sind drei Arten von Parametern:

- ein Parameter, der besagt, welches von mehreren Elementen gezeichnet werden soll, z.B. welcher Eintrag einer Listbox oder eines TreeViews,
- die Koordinaten des Rechtecks, in welchem das Element gezeichnet werden soll.
- ein Statusparameter, der den Zustand des Elements angibt, z.B. ob das Element ausgewählt, fokussiert, deaktiviert ist usw.

Die Methode, die das *OnDraw...*-Ereignis bearbeitet, kann die Grafik nun mit Hilfe dieser Parameter und des *Canvas*-Objekts, das von der Komponente zur Verfügung gestellt wird, ausgeben.

Des Weiteren unterscheidet man bei besitzergezeichneten Komponenten zwei wichtige Kategorien: Komponenten, deren dargestellte Elemente eine feste Größe haben, und Komponenten, bei denen diese Größe variiert. Bei *TListBox* können Sie z.B. mit den Stilen *lbOwnerDrawFixed* und *lbOwnerDrawVariable* zwischen beidem wählen, bei *TTabSet* haben die einzelnen Tabs immer eine von Ihnen einzeln festlegbare variable Größe und bei *TDrawGrid* ist die Größe immer fest vorgegeben.

Falls die Größe der Elemente variabel ist, gilt es normalerweise, neben *OnDraw...* noch ein zweites Ereignis zu bearbeiten, dessen Name mit *OnMeasure* beginnt. Dieses stellt Ihnen einen Variablenparameter zur Verfügung, in dem Sie angeben, wie viel Platz ein bestimmtes Element in Anspruch nimmt.

Verfügbare Komponenten

Die Zahl der Komponenten, an deren Bildschirmdarstellung Sie mitwirken können, ist unter Delphi 6 zweistellig: Von den in allen Delphi-Versionen vorhandenen Komponenten können Sie *TabSet*, *ListBox*, *ComboBox*, *DrawGrid*, *DBGrid* und *Outline* selbst zeichnen, außerdem die *Win32*-Komponenten *TabControl*, *ListView*, *TreeView*, *Toolbar*, *StatusBar* und *HeaderControl*. Schließlich lässt sich seit Delphi 4 auch *TMenuItem* durch das Besitzerformular zeichnen.

Verwenden von TDrawGrid R84

Das Beispielprogramm *DrawGrid* (Abbildung 4.3) verwendet die 64x64 Zellen eines *DrawGrids* dafür, einen Farbübergang zu zeichnen, wobei jede Zelle mit Zeilen- und Spaltennummern beschriftet wird. Wir benötigen hierfür nur eine einzige Methode, die jede einzelne Zelle ausgibt. Sie ist mit dem *TDrawGrid*-Ereignis *OnDrawCell* verknüpft und hat die folgenden Parameter:

- *Col* und *Row* spezifizieren die zu zeichnende Zelle.
- *Rect* gibt an, in welchem Bereich der Zeichenfläche (*TDrawGrid.Canvas*) gezeichnet werden soll.
- *State* enthält Flags, die über den Status der Zelle Aufschluss geben: *gdSelected* (markiert; dieses Flag ist nur dann bei mehreren Zellen gleichzeitig möglich, wenn die Option *goRangeSelect* in *Options* gewählt ist), *gdFixed* (am Rand befestigte Zelle) und *gdFocused* (Zelle hat den Tastaturfokus; dies ist immer nur bei einer Zelle der Fall, *TDrawGrid* zeichnet bereits automatisch eine Umrandung für diese Zelle).

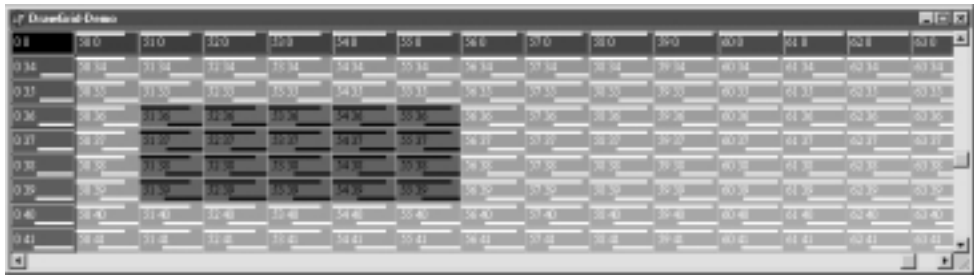


Abbildung 4.3: DrawGrid-Komponenten lassen ihre Zellinhalte vom Besitzerformular zeichnen.

Das Beispielprogramm berücksichtigt von *State* nur das Flag *gdSelected* und invertiert die Zelle, falls es gesetzt ist, mit der GDI-Funktion *InvertRect*. Alle anderen im Folgenden verwendeten Funktionen und Properties wurden bereits besprochen:

```
procedure TForm1.DrawGrid1DrawCell(Sender: TObject;
   Col, Row: LongInt; Rect: TRect; State: TGridDrawState);
begin
  with DrawGrid1.Canvas do begin
    { Zeichnen eines Hintergrunds: }
    Brush.Style := bsSolid;
    Brush.Color := RGB(Col*4, Row*4, 0);
    FillRect(Rect);
    { Zeichnen des Balkens über den Zahlen }
    Brush.Color := clWhite;
    Rect2 := ...(langweilige Berechnung, siehe CD-ROM)
    FillRect(Rect2);
    { Zeichnen des Balkens unter den Zahlen }
    Rect2 := ...(langweilige Berechnung, siehe CD-ROM)
    FillRect(Rect2);
    { Beschriftung der Zelle }
    Brush.Style := bsClear; { transparente Textausgabe }
    with Rect do
      TextOut(Left+4, Top+4, Format('%d %d', [Col, Row]));
    { Zusätzliche Hervorhebungen der selektierten Zellen: }
    if gdSelected in State then
      InvertRect(Handle, Rect);
  end;
end;
```

Hinweis: *TDrawGrid* besitzt im Gegensatz zu *TStringGrid* kein *Cells*-Property. Es verfügt jedoch wie *TStringGrid* über eine Möglichkeit, den Inhalt der Zellen zu editieren (Flag *goEditing* im Property *Options*). Wenn Sie den vom Benutzer eingegebenen Text speichern wollen, müssen Sie dies beim Ereignis *OnSetEditText* tun. Wenn der Benutzer erneut versucht, den Text zu verändern, erstellt *TDrawGrid* wieder ein Editierfeld und fragt Sie über das Ereignis *OnGetEditText*, wie der bisherige Text lautet.

TStringGrid erbt von *TDrawGrid* die Möglichkeit, vom Besitzer gezeichnet zu werden. Wenn Sie also ein Gitter-Element selbst zeichnen und gleichzeitig Text in jeder Zelle speichern wollen, ist die Verwendung von *TStringGrid* die einfachste Lösung.

TTreeView-CustomDraw-Ereignisse

Das besitzergesteuerte Zeichnen von TreeViews ist eine Funktion der Common Control-Bibliothek von Microsoft, in der das TreeView-Element definiert wird. Die VCL von Delphi kapselt die recht komplizierten Vorgänge um dieses besitzergesteuerte Zeichnen in einfach zu bearbeitenden TreeView-Events, ohne dadurch viel von der ursprünglichen Flexibilität des besitzergesteuerten Zeichnens zu verlieren.

Die Komponente *TTreeView* verfügt in allen Versionen über die folgenden Ereignisse:

```
OnCustomDrawItem(Sender: TCustomTreeView; Node: TTreeNode;
    State: TCustomDrawState; var DefaultDraw: Boolean);
OnCustomDraw(Sender: TCustomTreeView; const ARect: TRect;
    var DefaultDraw: Boolean);
```

Die Parameter *Node* und *State* beantworten die Frage »Was soll in welchem Zustand gezeichnet werden?«, ähnlich wie das beim ersten Beispiel des besitzergezeichneten *DrawGrids* der Fall war. Der Parameter für das »Wo« fehlt, denn das Rechteck können Sie vom Objekt *Node* selbst erfragen, und zwar über die Funktion *Node.DisplayRect*.

Statt dessen haben die beiden Ereignisse einen zusätzlichen Variablen-Parameter *DefaultDraw*. Wenn Sie diesen auf *False* setzen, unterbinden Sie das standardmäßige Zeichnen des TreeViews. Bedenken Sie jedoch, dass zu diesem Standardverhalten auch das Erzeugen weiterer *CustomDraw*-Ereignisse gehört. Wenn Sie also in *OnCustomDraw* *DefaultDraw* auf *False* setzen, erwartet der TreeView, dass Sie die gesamte Ausgabe bereits selbst erledigt haben. Der TreeView zeichnet danach weder selbst einen einzelnen Eintrag, noch erzeugt er ein *OnCustomDrawItem*-Ereignis dafür.

Hinweis: Manche besitzerzeichenbare Komponenten führen mit ihren Ereignissen keinen Parameter wie *DefaultDraw*, statt dessen muss man über ein Property einstellen, *ob* die Komponente durch den Besitzer gezeichnet wird. Dies gilt dann für die gesamte Komponente sozusagen nach dem »Alles-oder-Nichts«-Prinzip. Die Vorgehensweise von *TTreeView* mit den *DefaultDraw*-Parametern ist natürlich erheblich flexibler.

In Delphi 5 wurde *TTreeView* durch die folgenden Ereignisse erweitert:

```
OnAdvancedCustomDrawItem(Sender: CustomTreeView; Node: TTreeNode;
    State: TCustomDrawState; Stage: TCustomDrawStage;
    var PaintImages, DefaultDraw: Boolean);
OnAdvancedCustomDraw(Sender: TCustomTreeView; const ARect: TRect;
    Stage: TCustomDrawStage; var DefaultDraw: Boolean);
```

Der bei beiden zu findende zusätzliche Parameter *Stage* folgt daraus, dass beide Ereignisse sowohl vor dem standardmäßigen Zeichnen als auch danach auftreten, während die zuvor genannten Ereignisse lediglich zu Beginn des Zeichnens auftreten. *Stage* kann einen der Werte *cdPrePaint* (vor dem Zeichnen) und *cdPostPaint* (nach dem Zeichnen) annehmen.

Wenn Sie auf *cdPostPaint* warten, können Sie die vom TreeView standardmäßig angezeigten Einträge durch eine eigene Bildschirmausgabe *erweitern*. Während alles, was Sie in *cdPrePaint* ausgeben, durch eine eventuelle durch den TreeView durchgeführte Bildschirmausgabe überdeckt werden kann, haben Sie bei *cdPostPaint* die Gelegenheit, die Standard-Bildschirmausgabe selbst zu »übermalen«.

Hinweis: Wenn Sie vorher schon im Stadium *cdPrePaint* den Parameter *DefaultDraw* auf *False* gesetzt haben, tritt kein *cdPostPaint*-Ereignis auf. Dies ist auch nicht nötig, da Sie alles, was Sie in *cdPostPaint* zeichnen wollen, schon in *cdPrePaint* zeichnen können, ohne Gefahr, dass es danach durch eine Standard-Bildschirmausgabe verdeckt wird.

Das Ereignis *OnAdvancedCustomDrawItem* hat gegenüber seiner einfachen Variante noch einen weiteren Vorteil: Im Parameter *PaintImages* können Sie auch, wenn der TreeView ansonsten selbst für die Bildschirmausgabe sorgen soll, das Zeichnen der Icons neben den Baumeinträgen unterbinden (*PaintImages = False)*. Falls der TreeView ansonsten Icons zeichnen würde (dafür muss er mit einer Bilderliste verbunden sein), lässt er eine Lücke, die Sie dann im Stadium *cdPostPaint* füllen können. Auf diese Weise erhalten Sie keinen komplett besitzergezeichneten TreeView, sondern einen TreeView mit besitzergezeichneten Bildsymbolen.

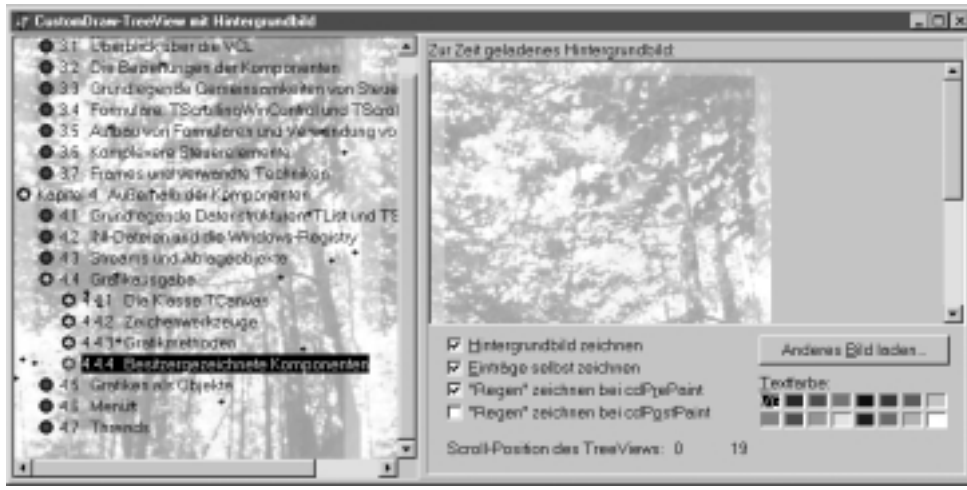


Abbildung 4.4: Das Beispielprogramm DrawTreeView mit einem komplett selbst gezeichneten TreeView-Element

DrawTreeView-Beispielprogramm

Das Beispielprogramm *DrawTreeView* (Abbildung 4.4) verwendet einige der *CustomDraw*-Ereignisse von *TTreeView*, um den gesamten Inhalt selbst auszugeben und dabei ein Bild als Hintergrund zu verwenden. Auf der CD finden Sie neben der hier beschriebenen Version auch eine spezielle Version für die Delphi-Versionen 3 und 4, in denen die Ereignisse *OnAdvancedCustomDraw[Item]* noch nicht zur Verfügung stehen.

Für das Zeichnen des Hintergrunds wartet das Programm auf das *OnCustomDraw*-Ereignis, für die Ausgabe der einzelnen Einträge auf *OnAdvancedCustomDrawItem*. Da

ein wichtiger Vorteil dieses *Advanced*-Ereignisses gegenüber dem normalen Ereignis erst bei Verwendung von *cdPostPaint* zum Tragen kommt, soll das Beispielprogramm *cdPostPaint* dazu verwenden, eine eventuelle Standardausgabe des TreeViews zu verändern, und zwar durch ein paar blaue »Konfetti«, die über den gezeichneten Eintrag zufällig verteilt werden (im Fenster des Programms alternativ auch als »Regen« bezeichnet). Wie erwähnt, gibt es aber nur dann ein *cdPostPaint*-Ereignis, wenn der Zeichenvorgang nicht schon bei *cdPrePaint* dadurch abgeschlossen wird, dass *DefaultDraw* auf *False* gesetzt wird. Im Beispielprogramm können Sie durch Markierungsschalter diese und andere Optionen manuell ein- und ausschalten.

Die selbst gezeichnete Grafik des Beispielprogramms soll wie folgt aussehen, sofern alle Optionen gewählt sind:

- Als Hintergrund soll eine Grafik dargestellt werden, die zur Laufzeit geändert werden kann (siehe Abbildung 4.4 und Schalter ANDERES BILD LADEN).
- Auf diesem Hintergrund sollen die TreeView-Einträge in der voreingestellten Schriftart des TreeViews gezeichnet werden, die Farbe soll jedoch durch den Benutzer verändert werden können (Farbwahlelement im Fenster rechts unten). Der Texthintergrund soll transparent sein, d.h. das Hintergrundbild soll zwischen den Zeichen »hindurchscheinen«.
- Expandierbare Einträge, die normalerweise mit einem Plus-Symbol versehen werden, sollen durch einen gefüllten Kreis, nicht expandierbare Einträge durch einen leeren Kreis gekennzeichnet werden (die Kreise wurden gewählt, weil sie einfacher zu zeichnen sind als die Minus- und Plus-Symbole).
- Der selektierte Eintrag (*State* = *odsSelected*) wird nicht mit transparentem Text dargestellt, sondern mit der voreingestellten dunklen Hintergrundfarbe, damit er besser zu erkennen ist.

Bevor sich aus den bisherigen Überlegungen eine funktionierende Methode bauen lässt, gilt es noch ein paar Besonderheiten zu beachten.

Einfluss des Scrollings auf das Zeichnen

Die Koordinaten des Zeichenrechtecks (*Node.DisplayRect*) beziehen sich nur auf das sichtbare Rechteck. Ein TreeView kann aber sowohl horizontal als auch vertikal gescrollt werden. Wird der Inhalt des TreeViews nun mit der horizontalen Bildlaufleiste nach links verschoben, muss außer dem *DisplayRect* zusätzlich diese Verschiebung in die Zeichenkoordinaten einberechnet werden. Mit anderen Worten: Da *DisplayRect.Left* den Beginn des sichtbaren Teils des TreeView-Eintrags angibt, muss der tatsächliche Beginn des Eintrages links davon positioniert werden. Beim vertikalen Scrollen stellt sich dieses Problem nicht, da ein Eintrag am oberen Ende des TreeViews entweder ganz oder gar nicht sichtbar ist, also nicht teilweise herausgescrollt werden kann.

Die X-Koordinate für den Beginn der Zeichnung wird im Programm auch nicht aus dem *DisplayRect* errechnet, sondern aus der Einrückungstiefe des TreeViews (*TTreeView.Indent*) multipliziert mit der Zahl der Ebenen, die der Knoten eingerückt ist (*TTreeNode.Level*; das *DisplayRect* wird nur für die Y-Koordinate verwendet).Um die Scroll-Position zu ermitteln, ruft das Beispielprogramm die API-Funktion *GetScrollPos* auf und übergibt dieser das Handle des Steuerelements, dessen Scrollbar abgefragt werden soll, also *TCustomTreeView.Handle*. Die Scroll-Position wird in *ScrollX* gespeichert und von der X-Koordinate abgezogen:

```
procedure TForm1.TreeView1AdvancedCustomDrawItem(Sender: TCustomTreeView;
  Node: TTreeNode; State: TCustomDrawState; Stage: TCustomDrawStage;
  var PaintImages, DefaultDraw: Boolean);
var
  ScrollX, NodeHeight, x, y: integer;
  ARect: TRect;
begin
  ARect := Node.DisplayRect(False);
  DefaultDraw := True;
  if Stage = cdPrePaint then begin
    ScrollX := GetScrollPos(Sender.Handle, SB_HORZ);
    NodeHeight := (ARect.Bottom-ARect.Top+1);
    if cbCustomDrawItem.Checked then begin
      x := Node.Level * (Sender as TTreeView).Indent - ScrollX;
      with Sender.Canvas do begin // Zeichnen auf dem TreeView-Canvas
        (*** 1. Zeichnen des Plus/Minus-Symbol-Ersatzes: ***)
        Pen.Width := 3;
        Brush.Color := clRed;
        if Node.Expanded or (Node.HasChildren = False) then
          Brush.Style := bsClear
        else Brush.Style := bsSolid;
        if Node.Selected then
          Pen.Color := clRed
        else Pen.Color := clBlue;
        Ellipse(x+3, ARect.top+3, x+NodeHeight-6, ARect.top+NodeHeight-6);
        (*** 2. Zeichnen des Textes: ***)
        (Sender as TTreeView).Font.Color := ColorGrid1.ForegroundColor;
        if not (cdsSelected in State) then begin // Item nicht selektiert
          // transparenten Hintergrund für den Text verwenden:
          SetBkMode(Sender.Canvas.Handle, TRANSPARENT);
          TextOut(x+NodeHeight, ARect.top, Node.Text)
        end else begin // Item ist selektiert
          SetBkMode(Sender.Canvas.Handle, OPAQUE); // Hintergrund "fest"
          TextOut(x+NodeHeight, ARect.top, Node.Text)
        end;
      end;
      DefaultDraw := False;
    end;
    if cbPrePaintRegen.Checked then begin
      DrawRain(ARect, Sender.Canvas);
```

```
        DefaultDraw := False;
      end;
    end;

    if Stage = cdPostPaint then
      if cbPostPaintRegen.Checked then begin
        DrawRain(ARect, Sender.Canvas);
        DefaultDraw:=False;
      end;
  end;
```

Hinweis: Die Einstellung von Zeichenattributen (*Brush*, *Pen*, *Font*) kann in einem TreeView eigentümliche Effekte haben, die zumindest teilweise in einem fehlerhaften Verhalten der VCL begründet liegen. Dies ist ein Grund dafür, dass oben die Schriftfarbe nicht über das *Font*-Objekt von *TCanvas*, sondern über das des TreeViews angepasst und transparenter Text nicht mit einem Pinselstil von *bsClear*, sondern über die Windows-API-Funktion *SetBkMode* eingestellt wird. Eine genauere Beschreibung finden Sie im Quelltext auf der CD.

Ein Hintergrundbild für einen TreeView *R88*

Das Hintergrundbild im Beispielprogramm soll gekachelt nebeneinander gezeichnet werden, so dass es, wenn es kleiner ist als der TreeView, trotzdem die gesamte sichtbare Fläche ausfüllt. Außerdem soll es sich beim Scrollen zusammen mit den Einträgen des TreeViews verschieben.

Beim Zeichnen des Hintergrundbildes spielt daher nicht nur die horizontale Scroll-Position eine Rolle, sondern auch die vertikale, denn durch das Scrollen kann das Hintergrundbild links und oben teilweise aus dem sichtbaren Bereich heraustreten.

Die folgende Methode wird mit dem Ereignis *OnCustomDraw* verknüpft, wird also aufgerufen, bevor die einzelnen Einträge in *OnAdvancedCustomDrawItem* gezeichnet werden. Sie berechnet neben *ScrollX* auch eine Variable *ScrollY* und zeichnet dann in zwei ineinander geschachtelten *while*-Schleifen die Bitmap, die der Benutzer in der *TImage*-Komponente des rechten Fensterteils ausgewählt hat. Damit der gesamte sichtbare Bereich abgedeckt wird, müssen die *while*-Schleifen so lange laufen, wie die Startkoordinaten für die nächste Kopie des Bildes kleiner sind als die Endkoordinaten des TreeView-Zeichenbereichs (übergeben im Ereignis-Parameter *ARect*):

```
procedure TForm1.TreeView1CustomDraw(Sender: TCustomTreeView;
  const ARect: TRect; var DefaultDraw: Boolean);
var
  ScrollX, ScrollY: Integer;
  x, y: Integer;
  Bitmap: TBitmap;
  NodeHeight: Integer;
```

```
begin
  if cbUseBackgroundPic.Checked then begin
    ScrollX := GetScrollPos(Sender.Handle, SB_HORZ);
    if (Sender as TTreeView).Items.Count = 0 then
      ScrollY := 0
    else begin
      with (Sender as TTreeView).Items[0].DisplayRect(False) do
        NodeHeight := Bottom - Top+1;
      ScrollY := GetScrollPos(Sender.Handle, SB_VERT)*(NodeHeight);
    end;
    with Sender.Canvas do begin
      Brush.Color := clWhite;
      FillRect(ARect);
      Bitmap := Image1.Picture.Bitmap;
      x := -ScrollX;
      while (x < ARect.Right) do begin
        y := -ScrollY;
        while (y < ARect.Bottom) do begin
          Draw(x, y, Image1.Picture.Bitmap);
          inc(y, Bitmap.Height);
        end;
        inc(x, Bitmap.Width);
      end;
    end;
    DefaultDraw := True;
  end else
    DefaultDraw := True;
end;
```

Das eigentliche Zeichnen der Bitmap findet mit der *TCanvas*-Methode *Draw* statt, welche zusammen mit der Bitmap-Klasse selbst (*TBitmap*) im nächsten Kapitel erläutert wird. Wichtige Voraussetzung für die Wirkung des Hintergrundbildes ist, dass die einzelnen Einträge im transparenten Hintergrundmodus (*Brush.Style = bsClear*) gezeichnet werden.

4.5 Grafiken als Objekte

Neben den im letzten Kapitel besprochenen grafischen Objekten, die Sie mit *TCanvas*-Methoden erzeugen, können Sie auch komplette Grafiken am Stück zeichnen. Unter Windows gibt es drei Typen dieser Grafiken, die in der VCL durch die folgenden Klassen gekapselt sind:

- *TBitmap*-Objekte sind Bitmapgrafiken beliebiger Größe, die Sie neu erzeugen oder aus einer Datei und aus der Zwischenablage laden können.
- *TIcon* steht für ein einzelnes Icon, einer speziellen Bitmap, die nur bestimmte Größen annehmen darf.

- *TMetafile* schließlich kapselt die Metadateien, in denen Windows Folgen von Grafikbefehlen speichert. Die Grafik einer Metadatei setzt sich also nicht aus Pixeln, sondern aus den Objekten zusammen, die Sie z.B. mit den *TCanvas*-Methoden zeichnen können.

4.5.1 Die drei TGraphic-Klassen

Wenn Sie wissen, wie kompliziert sich die gesamte Handhabung von Bitmaps, Metadateien und Icons mit dem Windows-API gestaltet, werden Sie die Methoden der Klasse *TGraphic* besonders zu schätzen wissen. *TGraphic* ist die gemeinsame Basisklasse der drei in der Einleitung genannten Typen und definiert abstrakte virtuelle Methoden, mit denen Sie eine Grafik aus einer Datei oder aus der Zwischenablage laden bzw. sie dort speichern können.

Da die drei abgeleiteten Klassen die folgenden Methoden jeweils auf ihre Weise implementieren (soweit notwendig), können Sie diese bei allen drei Grafiktypen gleichermaßen verwenden:

Methode	**Beschreibung**
Assign(Source: TPersistent)	überschreibt *TPersistent.Assign*, weist dem Grafik-Objekt den Inhalt eines *TPicture*-Objekts zu.
LoadFromFile(FileName: String)	öffnet eine `.bmp`-, `.ico`- oder `.wmf`-Datei, liest die Grafik und schließt die Datei wieder. Seit Delphi 3 können Sie auch `.jpg`-Dateien lesen, wenn Sie die Unit *Jpeg* in Ihr Projekt einbinden.
SaveToFile(FileName: String)	erstellt eine neue `.bmp`-, `.ico`- oder `.wmf`-Datei und schreibt die Grafik im Standardformat hinein.
LoadFromStream(TStream)	liest die Grafik von der aktuellen Stream-Position (wobei der Stream beispielsweise eine offene Datei sein kann).
SaveToStream(TStream)	schreibt die Grafik an die aktuelle Stream-Position.

Außerdem gibt es die folgenden Properties:

Property/Ereignis	**Beschreibung**
Empty	*True*, wenn noch keine Grafik mit den oben aufgezählten Methoden geladen oder zugewiesen wurde
Height	Höhe der Grafik in Pixeln
Modified	*True*, wenn die Grafik seit dem letzten Speichern geändert wurde
Width	Breite der Grafik in Pixeln
OnChange	Verknüpfung zu einer Methode, die bei jeder Änderung der Grafik aufgerufen wird, auch bei jeder Änderung des *Canvas* einer Bitmap

Während die Bitmaps das Kapitel 4.5.2 ganz für sich haben, werden die beiden anderen Klassen im Folgenden kurz beschrieben.

Icons

Icons sind Bitmaps, die vorwiegend zur Symbolisierung von Anwendungen und Dokumenten verwendet werden (einige Klassen der VCL wie *TApplication* und *TForm* verfügen zu diesem Zweck über ein *Icon*-Property). Da Icons an strenge Größenvorgaben gebunden sind, werden für viele alltägliche Aufgaben innerhalb einer Anwendung Bitmaps statt Icons verwendet, so z.B. für die Bilder auf den SpeedButtons. Während Icon-Symbole entweder 32x32, 32x16 oder 64x64 Pixel groß sein müssen, können Sie Bitmaps beispielsweise auch in der Größe 20x20 erzeugen.

Ein Vorteil von Icons gegenüber Bitmaps ist, dass Sie mehrere in Farbe und Auflösung unterschiedliche Versionen eines Icons als ein Bild (eine Icon-Ressource, eine Icon-Datei) speichern können (in Delphis Bildeditor drücken Sie einfach auf den Schalter *Neu*, um einem gerade editierten Icon eine neue Version hinzuzufügen). Windows kann nun in unterschiedlichen Grafikmodi jeweils ein anderes Icon auswählen, um die Anforderungen der Grafikkarte möglichst gut zu erfüllen. So benötigen Sie normalerweise kein Icon im Format 32x16, da dieses für Grafikmodi mit lang gezogenen Pixeln gedacht ist, es aber seit Einführung des VGA-Standards fast nur noch Grafikmodi mit quadratischen Pixeln gibt.

Sie können *TIcon*-Objekte zur Laufzeit einfach über den parameterlosen Konstruktor *Create* erstellen, mit den von *TGraphic* geerbten Methoden laden und speichern sowie mit den *TCanvas*-Methoden *Draw* und *StretchDraw* am Bildschirm ausgeben wie eine Bitmap. Falls Sie das *Icon*-Property eines Formulars oder der Anwendung verändern wollen, brauchen Sie diesem jedoch kein neues Icon-Objekt zuzuweisen, sondern können es direkt verändern:

```
Application.Icon.LoadFromFile('BUSY.ICO');
```

Metadateien

Metadateien sind die einfachste Form von geräteunabhängiger Grafik, denn sie bestehen intern einfach aus einer Folge von Grafikausgabebefehlen. Diese sind zwar auf Windows festgelegt, also nicht ohne weiteres auf andere Plattformen übertragbar, dafür ist die Austauschbarkeit unter verschiedenen Windows-Anwendungen aber umso besser. Natürlich wird eine Anwendung ihre Dokumente kaum primär in Metadateien speichern, denn die interne Datenstruktur der Anwendung ist meist völlig anders aufgebaut, aber häufig erlauben es Programme, eine Grafik als Metadatei zu exportieren oder zu importieren, wobei dies entweder über die Zwischenablage oder über eine Datei möglich ist.

Dank der Basisklasse *TGraphic* lässt sich *TMetafile* genauso einfach aus einer Datei oder aus der Zwischenablage laden, am Bildschirm ausgeben und in eine Datei oder in die Zwischenablage kopieren wie *TIcon*. Das Ausgeben einer Metadatei am Bildschirm besteht darin, dass die Grafikbefehle, die in der Datei gespeichert sind, auf die *Canvas*-Zeichenfläche (bzw. auf das darunter liegende Windows-Objekt) angewendet werden. Dieser Vorgang wird auch als »Abspielen« der Metadatei bezeichnet (API-Funktion *PlayMetafile*).

Im Gegensatz zu Icons ist es bei Metadateien sehr leicht vorstellbar, dass Sie sie auch im Programm verändern bzw. von Grund auf erzeugen wollen. Hierfür stellt Ihnen die VCL die Klasse *TMetafileCanvas* zur Verfügung.

Metadateien erzeugen

Das folgende Beispiel ist dem TreeDesigner aus Kapitel 5 entnommen. Mit dem Menüpunkt BEARBEITEN | KOPIEREN können Sie dort die gesamte Grafik des aktuellen Dokuments als Metadatei in die Zwischenablage kopieren.

Im ersten Schritt muss mit einem normalen Konstruktor-Aufruf ein neues Metadatei-Objekt erzeugt werden. Die Größe der Metadatei wird über die Properties *Width* und *Height* festgelegt, die im TreeDesigner auf die exakte Größe der Grafik gesetzt werden soll (und nicht etwa auf die Gesamtgröße der Zeichenfläche). Die Größe der Grafik wird mit der TreeDesigner-Funktion *GetPaintRect* ermittelt:

```
procedure TDocumentForm.Kopieren1Click(Sender: TObject);
var
  Metafile: TMetafile;
  MFCanvas: TMetafileCanvas;
  R: TRect;
begin
  Metafile := TMetafile.Create;
  Document.GetPaintRect(R); // Äußere Umrisse der Grafik ermitteln
  Metafile.Width := R.Right-R.Left; // Breite der Grafik
  Metafile.Height := R.Bottom-R.Top; // Höhe der Grafik
```

Bevor Sie in eine solche neue Metadatei zeichnen können, benötigen Sie ein *TCanvas*-Objekt, das jedoch noch nicht als Property in der Metadatei enthalten ist. Sie müssen es mit einem weiteren Konstruktoraufruf erzeugen; im Falle des TreeDesigners sieht dieser Aufruf wie folgt aus:

```
  MFCanvas := TMetafileCanvas.Create(Metafile, 0);
```

Da *TMetafileCanvas* von *TCanvas* abgeleitet ist, können Sie ein solches Objekt wie ein normales *TCanvas*-Objekt verwenden; allerdings gibt es in der inneren Funktionsweise erhebliche Unterschiede zwischen einer normalen und einer Metadatei-Zeichenfläche, auf die hier aber nicht weiter eingegangen werden kann.

Im Beispiel des TreeDesigners findet vor der eigentlichen Ausgabe der Grafik noch die Einrichtung des Koordinatensystems in der Art statt, dass die Grafik exakt auf die zur Verfügung stehende Fläche der Metadatei abgebildet wird. Für die vier dazu notwendigen Programmzeilen sei auf die CD verwiesen, eine eingehendere Besprechung des Themas der Koordinatensysteme steht noch für Kapitel 5.6 auf dem Programm.

Den Abschluss unserer Zwischenablage-Kopieroperation bilden die folgenden Zeilen:

```
    Document.PaintAll(MFCanvas);
    MFCanvas.Free;
    Clipboard.Assign(Metafile);
    Metafile.Free;
  end;
```

Document.PaintAll erledigt die gesamte Grafikausgabe. Das Bemerkenswerte an diesem Aufruf ist, dass *PaintAll* eigentlich nur zur Ausgabe der Grafik auf dem Bildschirm gedacht ist (siehe Kapitel 5.2.2). Für *PaintAll* ist der *TCanvas*-Parameter ein polymorphes Objekt, das normalerweise für einen Bereich eines Fensters, in diesem Fall aber für eine Metadatei-Zeichenfläche steht.

Der Aufruf von *MFCanvas.Free* ist wieder typisch für Metadateien, denn Sie müssen die Zeichenfläche der Metadatei freigeben, damit die aufgebaute Grafik im Metadatei-Objekt zur Verwendung freigegeben wird. Die »Verwendung« besteht im obigen Beispiel darin, die Metadatei in die Zwischenablage zu kopieren (zur Zwischenablage siehe Kapitel 8.1). Im *Assign*-Aufruf legt die VCL eine komplette Kopie der Metadatei an, nach diesem Aufruf können Sie also weiter mit dem Metadatei-Objekt arbeiten, es z.B. mit der *TCanvas*-Methode *Draw* am Bildschirm ausgeben. Der TreeDesigner benötigt die Metadatei jedoch nicht mehr und gibt sie (bzw. das VCL-Objekt *Metafile*) einfach frei.

4.5.2 Bitmaps

In den meisten Fällen, in denen einzelne Pixel manipuliert werden, werden diese nicht direkt auf dem Bildschirm verändert, sondern in einer Bitmap, die sich im Speicher befindet. Mit Methoden von *TCanvas* können Sie Bitmaps vom Speicher auf den Bildschirm kopieren.

Vorteile von Speicherbitmaps

Programme, die sehr intensiven Gebrauch von Bitmaps machen, sind Mal- und Bildbearbeitungsprogramme. Bei diesen werden durch die Verwendung von Bitmaps mehrere Probleme gelöst:

- Da andere Fenster das Fenster des Bildbearbeitungsprogramms auf dem Bildschirm verdecken können, muss das Bild noch an einer anderen Stelle als auf dem Bildschirm gespeichert werden.
- Bitmaps im Speicher sind nicht an die Abmessungen des Bildschirms gebunden, sondern können viel größer sein.
- Auch die Farben eines Bitmaps lassen sich im Speicher geräteunabhängig behandeln. So kann ein Bildbearbeitungsprogramm ein Echtfarben-Bild laden und intern in einer Echtfarben-Bitmap speichern, die dann in einem Bildschirmmodus mit nur 256 oder 64 K Farben dargestellt wird. Bei der Darstellung muss Windows die 24-Bit-Farbinformationen dann in Farbindizes einer 256-Farben-Palette umrechnen bzw. auf die 16 Bit des 64-K-Farbmodus beschneiden.
- Schließlich können Bitmaps, wenn sie z.B. vom Speicher auf den Bildschirm kopiert werden, auch in der Auflösung angepasst werden, damit die Größen von Original und Abbildung übereinstimmen. Wenn Sie z.B. ein mit 300 dpi gescanntes Bild auf dem Bildschirm in gleicher Größe darstellen wollen, muss die Auflösung theoretisch auf die Auflösung des Monitors umgerechnet werden (z.B. 120 dpi). Bei der Größenveränderung einer Bitmap kommt es natürlich zu Informationsverlusten (wenn bei der Verkleinerung Pixel weggelassen werden) oder zu Klötzchenbildung (wenn bei der Vergrößerung Pixel einfach vervielfacht werden).

Geräteunabhängigkeit bei Bitmaps

Die genannten Punkte, besonders die letzten drei, lassen sich als Geräteunabhängigkeit zusammenfassen, wobei die Unabhängigkeit wegen der Verzerrungen und Informationsverluste bei der Größenänderung weit weniger groß ist als bei Metadateien.

TBitmap und TCanvas

Eine Bitmap weist sehr große Ähnlichkeit mit einem *TCanvas*-Objekt auf: Sie können auf beiden verschiedene grafische Objekte ausgeben, einzelne Pixel verändern und Sie können Ausschnitte zwischen beiden hin- und herkopieren. Diese Ähnlichkeiten gründen ausnahmsweise nicht darin, dass die eine Klasse von der anderen abgeleitet ist oder dass beide eine gemeinsame Basisklasse besitzen, sondern darin, dass *TBitmap* ein *TCanvas*-Objekt *enthält* (anzusprechen unter dem Property *Canvas*).

Da *TCanvas* bereits eine Zeichenfläche repräsentiert und mit dem *Pixels*-Array über den wesentlichen Bestandteil einer Bitmap verfügt, benötigt *TBitmap* nicht viel mehr als sein *Canvas*-Property und die von *TGraphic* geerbten (und überschriebenen) Methoden.

Zu diesen wenigen Erweiterungen gehören die Methode *Dormant* und die Properties *Monochrome*, *Handle* und *Palette*: Im Property *Monochrome* erfahren Sie, ob die Bitmap einfarbig ist, und in *Handle* und *Palette* finden Sie zwei intern verwendete Handles auf die GDI-Ressourcen von Windows (die Bitmap und die zugehörige Palette). Mit der Methode *Dormant* können Sie die durch die Bitmap belegten GDI-Ressourcen freigeben, ohne die Bitmap selbst zu verlieren (*TBitmap* speichert die Daten dann in einem selbst verwalteten Speicherbereich und erzeugt erst bei Bedarf wieder eine GDI-Bitmap).

Bitmaps erzeugen *R83*

Durch das *Canvas*-Property hat also die Klasse *TBitmap* ihrer Verwandten *TIcon* die Möglichkeit voraus, die einmal geladene Grafik auch zu verändern. Passend dazu können Sie mit *TBitmap* auch ganz neue Bitmaps erzeugen. Dazu genügt es, ein neues Bitmap-Objekt mit *Create* zu erzeugen und die Properties *Height* und *Width* zu verändern, anstatt die Bitmap mit einer der *TGraphic*-Methoden zu laden:

```
Bitmap := TBitmap.Create;
Bitmap.Height := 200;
Bitmap.Width := 200;
{ Ausgabe von Grafik: }
Bitmap.Canvas.Pixels[0, 0] := RGB(rot, gruen, blau);{ einen Punkt setzen }
Bitmap.Canvas.Polygon(...); { ein Polygon zeichnen }
```

4.5.3 TPicture

Die Klasse *TPicture* ist eine Klasse, die Metadateien, Bitmaps und Icons abstrakt als »Bilder« ansieht und die durch ihren Namen leicht mit der Klasse *TGraphic* verwechselt werden kann. Während jedoch *TGraphic* als Elternklasse für einen der drei Grafiktypen steht, *enthält* ein Objekt der Klasse *TPicture* ein von *TGraphic* abstammendes Objekt. Sinn der Klasse *TPicture* ist es, zur Laufzeit zwischen den drei Grafiktypen zu wechseln, ohne dass Sie dafür verschiedene Objekte verwalten müssen.

TImage

Zu allem Überfluss gibt es auch noch die Komponente *TImage*, deren Name zwar so gut wie das Gleiche meint wie *TPicture*, die aber eine visuelle Komponente ist, welche ein *TPicture*-Objekt enthält. Sie erlaubt Ihnen, zur Entwurfszeit im Formular einen Platz für ein Bild zu reservieren oder auch schon ein Bild zu laden (über das *Picture*-Property im Objektinspektor). Zur Laufzeit müssen Sie normalerweise nur mit dem *Picture*-Property umgehen, so dass Sie alle im Folgenden beschriebenen Vorgehensweisen von *TPicture* auf *TImage* übertragen können.

TPicture-Methoden

Die einfachste Verwendung von *TPicture* besteht in der Verwendung der Methoden *LoadToFile* und *SaveToFile*, die den Typ der Datei anhand der Endung automatisch erkennen und ein entsprechendes Objekt erstellen:

```
Picture1.LoadFromFile('GREETING.BMP');
Picture2.LoadFromFile('LOGO.WMF');
```

TPicture verfügt auch über die Properties *Height* und *Width*, die die Größe der Grafik zurückgeben. Grafiken, die Sie wie im obigen Beispiel laden, können Sie mit den *TCanvas*-Methoden *Draw* und *StretchDraw* ausgeben, so dass Sie für diese einfache Verwendung gar nicht mehr über *TPicture* zu wissen brauchen. Unter dieser Oberfläche ist *TPicture* jedoch eine sehr flexible Grafikobjekt-Verwaltungszentrale.

Aufbau von TPicture

Zentraler Bestandteil eines *TPicture*-Objekts ist ein *TGraphic*-Objekt, das Sie über das Property *Graphic* ansprechen können. Je nachdem, ob es sich dabei um eine Bitmap, ein Icon oder eine Metadatei handelt, können Sie es auch über die Properties *Bitmap*, *Icon* und *Metafile* ansprechen. Der große Vorteil dieser Properties gegenüber eigenständigen *TGraphic*-Objekten ist, dass Sie sich um die Konstruktion und die Freigabe dieser Objekte nicht zu kümmern brauchen. So können Sie nacheinander Grafiken verschiedenen Typs in das *Picture* laden:

```
Picture.Bitmap.LoadFromFile(...);
Canvas.Draw(0, 0, Picture);
Picture.Metafile.LoadFromFile(...)
```

Jedesmal, wenn Sie eines der Properties *Bitmap*, *Icon* und *Metafile* ansprechen, prüft *TPicture*, welcher Objekttyp gerade aktiv ist. Ist dies ein anderer als der von Ihnen gewählte, gibt *TPicture* das alte Objekt frei und erzeugt ein neues. Im obigen Beispiel wird also die zuerst geladene Bitmap beim zweiten *LoadFromFile*-Aufruf durch eine Metadatei ersetzt.

Das einzige Objekt, das Sie noch selbst konstruieren und freigeben müssen, ist *TPicture* selbst (*Picture := TPicture.Create* und *Picture.Free*), allerdings können Sie sich auch das ersparen, wenn Sie beim Entwurf des Formulars eine *TImage*-Komponente einfügen, die ja wie alle anderen Komponenten des Formulars automatisch verwaltet wird. Auf deren Property *Picture* können Sie so zugreifen wie auf ein eigenständiges *TPicture*-Objekt.

Grafikobjekte kopieren

Über den bisher gezeigten Zugriff hinaus können Sie den Properties *Graphic*, *Bitmap*, *Icon* und *Metafile* auch selbst erzeugte Objekte zuweisen:

```
    Bitmap := TBitmap.Create;
    Picture.Bitmap := Bitmap;
  { oder Picture.Graphic := Bitmap }
    Bitmap.Free;
```

TPicture kopiert den Inhalt der so zugewiesenen Objekte mit der in *TPersistent* eingeführten Methode *Assign* (siehe Kapitel 3.1.1) in sein eigenes Grafik-Objekt, verändert das zugewiesene Original-Objekt also nicht, so dass Sie dies wie im obigen Beispiel selbst wieder freigeben müssen.

Sie können auch jegliche Grafik aus dem *TPicture*-Objekt entfernen, indem Sie dem entsprechenden Property den Wert *nil* zuweisen. Da all dies auch für das *Picture*-Property der *TImage*-Komponente gilt, zum Abschluss ein Beispiel für diese Komponente:

```
  { Löschen der gerade angezeigten Grafik: }
    Image.Picture.Graphic := nil;
  { oder einfach: Image.Picture := nil }
```

4.5.4 Bitmaps für eine OwnerDraw-Listbox

Das Beispielprogramm *OwnerDraw1* enthält eine besitzergezeichnete Listbox, in der alle Bitmap-Dateien eines Verzeichnisses angezeigt werden (siehe Abbildung 4.5). Zur Wahl dieses Verzeichnisses sind eine *TDriveComboBox*- und eine *TDirectoryListBox*-Komponente so verknüpft, wie in Kapitel 1.9.3 beschrieben. Statt in einer *TFileListBox*-Komponente werden die Dateien des gewählten Verzeichnisses jedoch in einer normalen Listbox angezeigt, deren *Style*-Property auf *lbOwnerDrawVariable* eingestellt wurde und um die es hier geht.

Die erste Programmversion geht übrigens nicht besonders schonend mit dem Speicher um, da sie alle Bitmaps gleichzeitig und unkomprimiert in den Speicher lädt. Zur Ansicht von Verzeichnissen mit vielen großen Bitmaps sollten Sie daher unbedingt die zweite Programmversion *OwnerDraw2* verwenden, die am Schluss dieses Kapitels vorgestellt wird.

Hinweis: Die ebenfalls auf der CD befindliche Version *OwnerDraw3* zeichnet die Bitmaps nicht nur in die Listbox, sondern auch noch in ein selbst gezeichnetes Menü. Auch im Menü wird mit *OnMeasureItem*- und *OnDrawItem*-Nachrichten gearbeitet. Das besitzergezeichnete Menü weist damit zur besitzergezeichneten Listbox kaum Unterschiede in der Implementierung auf und wird im folgenden nicht weiter besprochen. Da besitzergezeichnete Menüs erst seit Delphi 4 zur Verfügung stehen, können Sie mit früheren Delphi-Versionen nur *OwnerDraw1* und *OwnerDraw2* erzeugen.

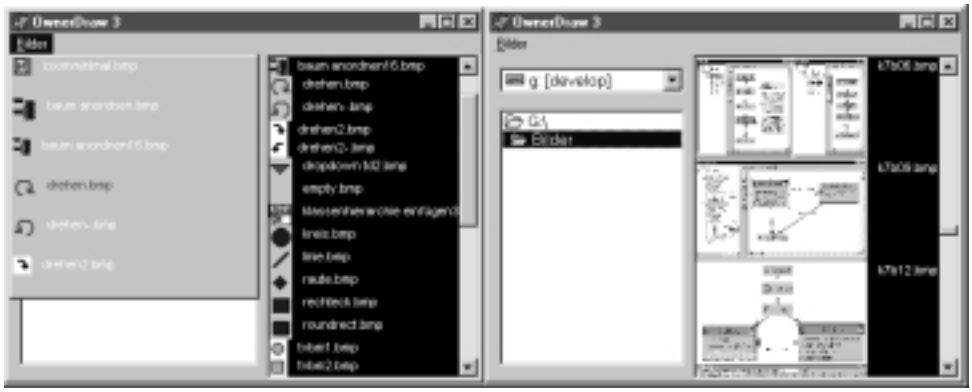


Abbildung 4.5: Eine OwnerDraw-ListBox und ein selbst gezeichnetes Menü im Beispielprogramm OwnerDraw3

Die Formularklasse enthält drei zusätzliche Variablen und zwei selbst erzeugte Methoden:

```
TForm1 = class(TForm)
  ...
public
  Dir: string;
  MaxHeight: Integer;
  Bitmaps: TList;
  procedure LoadFiles;
  procedure FreeListItems;
end;
```

Dir enthält das Verzeichnis, dessen Bitmap-Dateien geladen werden sollen. Es wird von der Laufzeitbibliothek automatisch als leerer String initialisiert, so dass die Dateien des aktuellen Verzeichnisses gezeigt werden.

MaxHeight gibt die maximale Höhe in Pixeln an, die eine Bitmap in der Listbox-Komponente für sich beanspruchen darf. Sie wird beim *OnCreate*-Ereignis mit dem Wert 90 initialisiert.

Bitmaps ist eine Liste der Klasse *TList*, die ebenfalls in der *OnCreate*-Methode initialisiert und mit einem Aufruf von *LoadFiles* mit *TBitmap*-Objekten gefüllt wird:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MaxHeight := 90;
  Bitmaps := TList.Create;
  LoadFiles;
end;
```

Bitmaps laden *R123*

LoadFiles löscht zunächst den Inhalt der Listbox und die Bitmapliste und durchsucht dann das gerade gesetzte Verzeichnis (*Dir*) nach Bitmap-Dateien (zu den Standardfunktionen *FindFirst* und *FindNext* siehe Kapitel 2.8.1):

```
procedure TForm1.LoadFiles;
var
  SearchRec: TSearchRec;
  Bitmap: TBitmap;
begin
  with ListBox1 do begin
    Clear;
    FreeListItems;
    if FindFirst(Dir+'*.BMP', faAnyFile, SearchRec) = 0 then begin
      repeat
        Bitmap := TBitmap.Create;
        Bitmap.LoadFromFile(Dir+SearchRec.Name);
        Bitmaps.Add(Bitmap);
        Items.Add(LowerCase(SearchRec.Name));
      until FindNext(SearchRec) <> 0;
      SysUtils.FindClose(SearchRec);
    end;
  end;
end;
```

Für jede gefundene Datei erstellt die Methode mit *TBitmap.Create* ein neues Bitmap-Objekt und beauftragt dieses, die Datei zu lesen. Daraufhin fügt die Methode die Bitmap an die Bitmap-Liste an und erweitert die Listbox um den Namen der Datei.

Hinweis: Normalerweise wäre das Property *TListBox.Items.Objects* ein geeigneter Ort, zu jedem Dateinamen die zugehörige Bitmap zu speichern. Dazu müssten wir hier statt *Items.Add* die Methode *Items.AddObject* aufrufen (siehe Kapitel 4.1.1). In diesem Fall funktioniert das jedoch nicht, denn der interne Ablauf von *AddObject* führt dazu, dass das im Folgenden noch erklärte *OnMeasureItem*-Ereignis auftritt, bevor die Bitmap in der Liste gespeichert wird. Da wir die Bitmap jedoch schon in der Bearbeitung von *OnMeasureItem* benötigen, wird sie hier in einer separaten Liste gespeichert.

Neue Verzeichnisse wählen

Jedes Mal, wenn der Benutzer das Verzeichnis wechselt, muss die Liste der Bitmaps und der Inhalt der Listbox neu aufgebaut werden. Das *OnChange*-Event der *DirectoryListBox* muss also – anders als bei automatischer Verknüpfung mit einer *TFileListBox* – selbst bearbeitet werden:

```
procedure TForm1.DirectoryListBox1Change(Sender: TObject);
begin
  Dir := DirectoryListBox1.Directory;
  { Verzeichnisname muss mit einem "\" enden,
    zur leichteren Verknüpfung mit dem Dateinamen: }
  if Dir[length(dir)] <> '\' then
    Dir := Dir+'\';
  LoadFiles;
end;
```

Bitmaps zeichnen

Nach dem erfolgreichen Aufbau der Bitmapliste können wir uns nun der Hauptaufgabe zuwenden, dem Zeichnen der Bitmaps. Hierbei spielt die Bitmap nicht mehr die aktive Rolle, d.h. es gibt keine Methode *DrawToCanvas*, die zu den Methoden *SaveToFile*, *CopyToClipboard* usw. passen würde. Statt dessen rufen Sie eine Methode des *Canvas* auf, in dem die Bitmap gezeichnet werden soll, und übergeben dieser die Bitmap als Parameter.

Zur Auswahl stehen die beiden *TCanvas*-Methoden *Draw* und *StretchDraw*, die in diesem Beispielprogramm beide zum Einsatz kommen: Falls die Bitmap nicht breiter ist als die Listbox und nicht höher als die in *MaxHeight* festgelegte maximale Höhe, wird sie mit *Draw* unverzerrt gezeichnet, andernfalls wird sie mit *StretchDraw* in das zur Verfügung stehende Rechteck gepresst unter Missachtung dessen, ob die Seitenverhältnisse noch stimmen. Der entsprechende Code-Ausschnitt sieht wie folgt aus (*Rect* ist das Rechteck, das der Eintrag in der Zeichenfläche der Listbox beansprucht):

```
if (Bitmap.Width > ListBox1.ClientWidth) or
   (Bitmap.Height > MaxHeight) then
  ListBox1.Canvas.StretchDraw(Rect, Bitmap)
else
  ListBox1.Canvas.Draw(Rect.Left, Rect.Top, Bitmap);
```

Bevor wir zur vollständigen Methode kommen, müssen wir uns etwas mit der besitzergezeichneten Listbox beschäftigen.

Besitzergezeichnete TListBox-Komponenten

TListBox ist eine der in Kapitel 4.4.4 erwähnten Komponenten, deren Elemente variable Größe haben können, wenn sie durch den Besitzer gezeichnet werden. Sie können im Property *Style* mit den Stilen *lbOwnerDrawFixed* und *lbOwnerDrawVariable* zwischen beidem wählen. Im letztgenannten Stil muss außer dem Zeichenereignis *OnDrawItem* noch ein zweites Ereignis namens *OnMeasureItem* bearbeitet werden. Dieses stellt Ihnen einen Variablenparameter zur Verfügung, in dem Sie angeben, wie viel Platz ein bestimmtes Element in Anspruch nimmt. Nachdem so jeder einzelne Eintrag abgemessen wird, erhält *OnDrawItem* jeweils die folgenden Parameter:

- den Index des zu zeichnenden Eintrags,
- die Koordinaten des Rechtecks, in dem der Eintrag gezeichnet werden soll,
- als Statusparameter eine Menge, die die Elemente *odSelected*, *odFocused* und *odDisabled* enthalten kann (für »Eintrag ist ausgewählt, fokussiert, deaktiviert«).

Zeichnen einer OwnerDraw-Listbox *R85*

Die Listbox des Beispielprogramms hat den Stil *lbOwnerDrawVariable* (angegeben im Property *Style*), zeichnet also jede Bitmap in ihrer jeweiligen Größe, wobei das selbst gesetzte Limit bei 90 Pixeln Höhe liegt.

Das Ereignis *OnDrawItem* liefert, wie oben beschrieben, die drei Parameter für das zu zeichnende Element (*Index*, verwendbar als Index für die Arrays *Bitmaps* und *TListbox.Items*), das Ausgaberechteck (*Rect*) und den Hervorhebungsstatus (*State*). Letzterer wird in diesem Beispielprogramm allerdings ignoriert. Wie oben schon beschrieben, gibt die Zeichen-Methode die Bitmap mit *Draw* und *StretchDraw* aus; vorher füllt sie den Hintergrund schwarz, und nachher zeichnet sie noch den Dateinamen auf die Bitmap:

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
var
  Bitmap: TBitmap;
  Text: string;
begin
  { zu zeichnen sind: }
  Bitmap := Bitmaps[Index];  { die Bitmap ... }
  Text := ListBox1.Items[Index]; { ... und der Dateiname }
  if Assigned(Bitmap) then
    with ListBox1.Canvas do begin
      { Schrift einstellen }
      Font.Color := clWhite;
      Font.Name := 'Arial';
      Font.Size := 8;
      { Hintergrund schwarz füllen }
      Brush.Style := bsSolid;
      Brush.Color := clBlack;
      FillRect(Rect);
      { Bitmap zeichnen }
      if (Bitmap.Width > ListBox1.ClientWidth) or
         (Bitmap.Height > MaxHeight) then
        StretchDraw(Rect, Bitmap)
      else
        Draw(Rect.Left, Rect.Top, Bitmap);
      { Bitmap beschriften }
      Brush.Style := bsClear; { transparenter Text }
      if Bitmap.Width > ListBox1.ClientWidth div 2 then
```

```
        { große Bitmap -> Text darauf ausgeben }
        TextOut(Rect.Left+10, Rect.Bottom-TextHeight('A'), Text)
      else
        { kleine Bitmap -> Text daneben ausgeben: }
        TextOut(Rect.Left+Bitmap.Width+10, Rect.Top, Text);
      Font.Color := clBlack;
    end;
end;
```

Der Parameter *Rect* wird in vielen der obigen Anweisungen dazu verwendet, die Grafik an der richtigen Position auszugeben; lediglich beim Größenvergleich zieht die Methode auch die Variablen *MaxHeight* und *ListBox1.ClientWidth* zu Hilfe (nur der Einfachheit halber, um Berechnungen wie *Rect.Right-Rect.Left* zu vermeiden).

Hinweis: Da die Listbox des Beispielprogramms wie üblich keinen horizontalen Scrollbar enthält, braucht sie nicht wie das TreeView-Beispiel aus Kapitel 4.4.4 die Scroll-Position zu berücksichtigen. Bei Listboxen *mit* horizontaler Bildlaufleiste stellt sich jedoch das gleiche Problem wie in Kapitel 4.4.4.

Abmessungen selbst gezeichneter Elemente

Auch die Methode für das *OnMeasureItem*-Event holt sich zunächst die in *Index* angegebene Bitmap aus der *Bitmaps*-Liste, um deren Größe feststellen zu können. Sie schreibt dann die erforderliche Höhe in den Variablenparameter *Height*, die selbst gesetzte Höhenbegrenzung *MaxHeight* (90 Pixel) einbeziehend:

```
procedure TForm1.ListBox1MeasureItem(Control: TWinControl;
     Index: Integer; var Height: Integer);
var
  Bitmap: TBitmap;
begin
  Height := 16;
  if Index < Listbox1.Items.Count then begin
    Bitmap := TBitmap(Bitmaps[Index]);
    if Assigned(Bitmap) then
      Height := Bitmap.Height;
    if Height > MaxHeight then Height := MaxHeight;
  end;
end;
```

Die Freigabe der dynamischen Bitmap-Liste *R124*

Da wir die Bitmaps hier nicht in Komponenten der Klasse *TImage* eingebunden haben, müssen wir sie selbst freigeben. Dazu genügt es nicht, die Liste freizugeben, sondern es muss für jedes einzelne Bitmap die Methode *Free* aufgerufen werden. Dies erledigt

die Formularmethode *FreeListItems*. Sie wird nicht nur beim Ende des Programms aufgerufen, sondern jedes Mal, wenn ein neues Verzeichnis ausgewählt wurde und die Liste neu aufgebaut werden muss (siehe *DirectoryListBox1Change*):

```
procedure TForm1.FreeListItems;
begin
  while Bitmaps.Count > 0 do begin
    TBitmap(Bitmaps[0]).Free;
    Bitmaps.Delete(0);
  end;
end;
```

Beim Ende des Programms wird zusätzlich zu dieser Methode auch die Liste *Bitmaps* freigegeben, die in *FormCreate* dynamisch erzeugt wurde:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FreeListItems;
  Bitmaps.Free;
end;
```

Optimierung der Bitmap-Liste *R94*

Ein großer Nachteil der oben gezeigten Bitmap-Lesemethode ist, wie schon gesagt, der enorme Speicherbedarf großer Bitmaps, denn obwohl auch große Bitmaps nur sehr klein dargestellt werden, werden sie komplett in den Speicher geladen. Es ist jedoch nicht schwer, diesen Nachteil zu beheben und sicherzustellen, dass für jede Bitmap nur so viel Speicher verbraucht wird, wie zur Darstellung in der Listbox notwendig ist.

In der verbesserten Programmversion *OwnerDraw2* ist die Methode *LoadFiles* so geändert, dass Bitmaps, die größer als 150x90 Pixel sind, auf diese Maximalgröße verkleinert werden, bevor sie endgültig in der Liste gespeichert werden.

Dazu benötigt die Methode *LoadFiles* eine neue lokale Variable ...

```
var
  SmallBitmap: TBitmap;
```

... und erhält eine neue *repeat*-Schleife, die die bisherige Schleife komplett ersetzt:

```
repeat
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile(Dir+SearchRec.Name);
  if (Bitmap.Height > 90) or
     (Bitmap.Width > 150) then
  begin
    SmallBitmap := TBitmap.Create;
    SmallBitmap.Height := 90;
    SmallBitmap.Width := 150;
    SmallBitmap.Canvas.StretchDraw(Rect(0, 0, 150, 90), Bitmap);
```

```
      Bitmaps.Add(SmallBitmap);
      Bitmap.Free; { die unverkleinerte Bitmap freigeben }
    end else
      Bitmaps.Add(Bitmap); { Bitmap ist schon klein genug }

    ListBox1.Items.Add(LowerCase(SearchRec.Name));
  until FindNext(SearchRec) <> 0;
```

Falls die Maximalgröße nicht überschritten wird, läuft die Methode wie bisher ab. Andernfalls erstellt sie, wie in Kapitel 4.5.2 beschrieben, eine völlig neue zweite Bitmap mit dem Namen *SmallBitmap*. In diese zeichnet sie mit *StretchDraw* eine kleine Version der großen Bitmap. Anschließend wird *SmallBitmap* statt *Bitmap* zur Bitmap-Liste hinzugefügt und *Bitmap* freigegeben. Für alle anderen Methoden des Programms ändert sich nichts.

Hinweis: Das simple *StretchDraw* im obigen Code-Auszug führt dazu, dass Bitmaps mit extremen Seitenverhältnissen (z.B. 1000 Pixel breit, aber nur 16 Pixel hoch) bis zur Unkenntlichkeit verzerrt werden, da sie immer in die 150x90 Pixel große Fläche gepresst werden. Die Programmversion auf der CD-ROM wurde daher noch einmal verbessert – sie behält nun nach Möglichkeit bei der Verkleinerung noch das Größenverhältnis der Bitmap bei.

4.6 Menüs und Aktionsmanagement

Obwohl sich Menüs ganz einfach visuell zusammensetzen und mit Methoden verbinden lassen, gibt es viele Gründe, sich auch zur Laufzeit im Programmcode etwas mit den Menüs zu beschäftigen, zum Beispiel:

- Das einfachste Ziel ist, Menüpunkte zeitweise zu deaktivieren, da sie gerade nicht anwendbar sind, oder sie mit einer Markierung zu versehen (Letzteres geht seit Delphi 6 auch automatisch, wenn Sie das *AutoCheck*-Property aktivieren). Für diese einfachen Aufgaben müssen Sie lediglich die Properties *Checked* und *Enabled* der entsprechenden *TMenuItem*- oder *TAction*-Komponente zur Laufzeit beeinflussen. Beispiele dafür finden Sie in Kapitel 5.2.6.
- Weniger einfach ist es, die Menüstruktur zur Laufzeit zu ändern. Zwar können Sie durch Austausch des Properties *TForm.Menu* gleich das gesamte Menü auswechseln, oft sollen aber nur kleine Erweiterungen vorgenommen werden, beispielsweise mit einer Fensterliste oder einer Dateiliste. Je nach Anwendung sind auch andere Möglichkeiten denkbar, bei denen eine solche Liste innerhalb eines Menüs erheblich benutzerfreundlicher ist als eine eigene Dialogbox und außerdem platzsparender als eine (aufklappbare) Liste innerhalb des Fensters.

- Eine weitere Möglichkeit ist die Veränderung der Tastenkürzel zur Laufzeit, die normalerweise nur auf Wunsch des Anwenders stattfindet. Mit Hilfe der VCL ist dieses Problem wieder sehr schnell zu bewältigen.
- Etwas aus der Reihe fällt das besitzergesteuerte Zeichnen eines Menüs, bei dem Sie die Bildschirmdarstellung des Menüs durch Ihren eigenen Code bestimmen. Dies läuft so ab wie das besitzergesteuerte Zeichnen anderer Komponenten, für welches Sie in den Kapiteln 4.4.4 und 4.5.4 einige Beispiele finden.

Die Kapitel 4.6.2 und 4.6.3 beschreiben, wie Sie die beiden zuletzt genannten Aufgaben mit der VCL bewältigen können. Davor erläutert Kapitel 4.6.1 die allgemeinen Grundlagen, die in der VCL-Unit *Menus* gelegt werden. Kapitel 4.6.4 befasst sich schließlich mit den Aktionslisten alias *TActionList*-Komponenten und Kapitel 4.6.5 mit dem erweiterten Aktionslistenkonzept von Delphi 6 Professional.

Popup-Menüs sind ein weiteres Menü-Thema, bei dem Sie es normalerweise jedoch weniger mit der Unit *Menus* zu tun bekommen (siehe beispielsweise Kapitel 5.2.6 und 5.8.3).

OOP mit Menüs

Die VCL sieht sowohl Menüs als auch einzelne Menüpunkte als eigene Objekte an und gestattet so zur Laufzeit, Menüeigenschaften wie aktiviert, deaktiviert, markiert etc. auf eine einfache Weise zu verändern. Auch der Aufbau von dynamischen Menüs zur Laufzeit ist relativ unkompliziert, denn die Menüklassen machen sich die indizierten Properties von Object Pascal zunutze, so dass von der kargen API-Schnittstelle von Windows nichts mehr zu sehen bleibt.

Das Windows-API selbst sieht übrigens nur Menüs (Menüzeilen und einzelne Popup-Menüs) als Objekte an, die einzelnen Menüpunkte verfügen unter Windows nicht über ein Handle, sondern lediglich über eine Kennzahl. Ein kleiner Nachteil der voll objektorientierten VCL ist, dass für jeden Menüpunkt eine Variable in die Deklaration des Formulars eingefügt wird, was bei großen Menüs etwas unübersichtlich werden kann.

4.6.1 Die Unit Menus

Menüs unterscheiden sich insofern stark von den anderen visuellen Komponenten, als die Menüklassen nicht von *TControl*, sondern direkt von *TComponent* abgeleitet sind. Die beiden Menütypen – Hauptmenü und Popup-Menü (*TMainMenu* und *TPopupMenu*) – haben die gemeinsame Basisklasse *TMenu*. Die Klasse für die einzelnen Menüpunkte, *TMenuItem*, ist unabhängig davon und direkt abgeleitet von *TComponent*.

Ein wichtiger Funktionsbereich der Unit *Menus* befindet sich außerhalb dieser Klassen in Form einer Funktionssammlung, die das dynamische Konstruieren von Menüs und

die Definition von Tastenkürzeln noch weiter vereinfacht, als das mit den Menüklassen allein möglich wäre (so ersetzt ein einziger Funktionsaufruf einen Konstruktoraufruf und mehrere Zuweisungen an Properties).

TMenuItem

Bei der Arbeit mit Objektinspektor und Menü-Designer sieht eine *MenuItem*-Komponente noch sehr harmlos aus: Der Objektinspektor zeigt nur die Properties, die für die Gestaltung und die Funktion des einzelnen Menüpunkts notwendig sind. Diese Properties von *TMenuItem* sind bereits in Kapitel 1.4.7 erklärt, so dass wir hier gleich unter diese Oberfläche gehen können.

Dort zeigt sich, dass die Klasse *TMenuItem* weit mehr ist als nur ein Container für die einzelnen Menüattribute wie Text, Aktivierung, Markierung etc. Alle *MenuItems*, die zu einer Menükomponente gehören, sind miteinander in einer baumartigen Struktur verknüpft (siehe Abbildung 4.6).

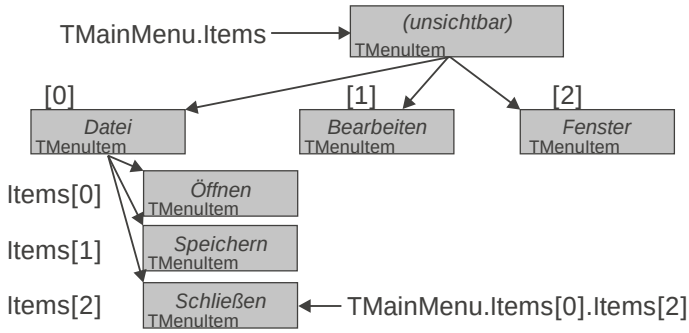


Abbildung 4.6: Aufbau der Menüstruktur mit Komponenten

Die Liste der Menüpunkte

In einem von 0 an indizierten Property namens *Items* speichert ein *TMenuItem*-Objekt alle Menüpunkte (wieder als *TMenuItem*-Objekte), die ihm untergeordnet sind. So enthält beispielsweise die *TMenuItem*-Komponente für den Hauptmenüpunkt DATEI einer normalen Anwendung die Menüpunkt-Komponenten für ÖFFNEN, SPEICHERN, SPEICHERN UNTER... usw. Wenn diese Menüpunkte ihrerseits keine weiteren Unterpunkte besitzen, dann bleibt ihr *Items*-Property leer.

In umgekehrter Richtung sind alle *MenuItems* über das Property *Parent* verknüpft. Es weist zu jedem Menüpunkt den übergeordneten Menüpunkt aus. Die Wurzel des Menübaumes wird von einem einzelnen *TMenuItem*-Objekt gebildet, dessen *Parent*-

Property *nil* ist. Es beinhaltet in seinem *Items*-Property alle Menüpunkte der obersten Ebene, die der Benutzer am Bildschirm sehen kann, ist aber selbst für den Benutzer unsichtbar.

Viele weitere Methoden und ein Property helfen bei der Verwaltung der *Items*-Liste. Sie stimmen in Namen und Funktion mit den entsprechenden Methoden der Klasse *TList* überein:

Element	**Bedeutung**
Property Count	Zahl der Menüpunkte
Insert(Index, Item)	fügt *Item* an Position *Index* in die Liste ein.
Delete(Index)	löscht den Menüpunkt an Position *Index*.
IndexOf(Item)	stellt fest, ob sich *Item* bereits in der Liste befindet, und gibt die Position zurück.
Add(Item)	hängt *Item* an das Ende der Liste an.
Remove(Item)	löscht *Item* aus der Liste.

Diese Funktionen können Sie beliebig zur Laufzeit des Programms aufrufen, um die Struktur des Menüs anzupassen. Wir kommen in Kapitel 4.6.2 auf diese Funktionen zurück, wenn es um die dynamische Erzeugung eines Menüs der zuletzt geöffneten Dateien geht.

TMenu, TPopupMenu und TMainMenu

Dadurch, dass die gesamte Menüstruktur bereits in einem *TMenuItem*-Baum gespeichert ist, wird die Klasse *TMenu* stark entlastet: Sie benötigt lediglich ein Property *Items*, das die unsichtbare Wurzel des Menübaums als *TMenuItem*-Komponente enthält (zur Erinnerung: im Objektinspektor können Sie über dieses Property den Menü-Designer aufrufen). Die beiden Klassen für Haupt- und Popup-Menüs fügen dem noch einige weitere Funktionen hinzu: *TMainMenu* erweitert *TMenu* um Funktionalität zum Verschmelzen von Menüs (siehe Kapitel 5.7.3) und für OLE (nur intern von der VCL verwendet); *TPopupMenu* enthält einige Properties, die das Verhalten des Popup-Menüs steuern (z.B. das Property *AutoPopup*, siehe Kapitel 5.2.6).

Als Beispiel für die Verwendung des *Items*-Properties kann hier der Tastenkürzel-Manager aus Kapitel 4.6.3 dienen, der die Punkte eines Menüs in einer ListBox aufführt. Die folgenden Zeilen hängen den Text (*Caption*) aller Menüpunkte einer *TMainMenu*-Komponente an die Liste einer *TListBox*-Komponente an. Die *TMainMenu*-Komponente ist in *FMenu* angegeben:

```
for i := 0 to FMenu.Items.Count-1 do
  ListBox1.Items.Add(FMenu.Items[i].Caption);
```

Wie in diesem Beispiel benötigen Sie die *TMainMenu*-Komponente oft nur, um an das *TMenuItem*-Objekt *Items* zu kommen. Insofern sind diese drei Zeilen eher Beispiel für die Verwendung von *TMenuItem* als für den Umgang mit *TMainMenu*.

Die Hilfsfunktionen *R22*

Wenn Sie ein Menü mit Hilfe der von *TMenuItem* bereitgestellten Methoden und Properties dynamisch zur Laufzeit aufbauen wollten, müssten Sie in etwa die folgenden Schritte unternehmen:

```
var
  MenuItem, MainMenu: TMenuItem;
begin
  MainMenu := TMenuItem.Create(...);
  MenuItem := TMenuItem.Create(...);
  MenuItem.Caption := 'Menutext';
  MainMenu.Add(MenuItem);
  ...
```

Dies wäre letztlich komplizierter, als wenn Sie das Menü mit den Windows-API-Funktionen zusammengestellt hätten. Einige Hilfsfunktionen der VCL vereinfachen die Sache jedoch deutlich, wenn ihre große Parameterzahl auch etwas unübersichtlich ist. Die folgende Tabelle sortiert diese Funktion nach dem Objekt, das sie erzeugen:

Objekt	**hergestellt von**	**Parameter**
TMainMenu	NewMenu	Owner: TComponent; Name: String; Items: array of TMenuItem
TPopupMenu	NewPopupMenu	Owner, Name, Alignment, AutoPopup, Items
TMenuItem mit Unterpunkten	NewSubMenu	Caption: String; helpCtx: Word; Name: String; Items: array of TMenuItem
TMenuItem ohne Unterpunkte	NewItem	Caption, ShortCut, Checked, Enabled, OnClick, helpCtx, Name
TMenuItem als Trennlinie	NewLine	(keine)

Die Bedeutung der Parameter stimmt mit denen der Properties von *TMenuItem* überein. Zugegebenermaßen sieht diese Tabelle auf den ersten Blick noch nicht nach einer Vereinfachung aus. Erst in einem Beispielprogramm zeigt sich der Nutzen dieser Funktionen.

Viele Funktionen erwarten als letzten Parameter ein offenes Array von Menüpunkten, das Sie in Object Pascal direkt aus geschachtelten Funktionsaufrufen angeben können. Beispielsweise können Sie einem Formular wie folgt ein neues Hauptmenü zuweisen (der Code ist dem Beispielprojekt *HLTest* (Demo für *THistoryList*) entnommen):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { Unsichtbare Wurzel des Menüs: }
  Menu := NewMenu(self, 'unsichtbarer Text', [
    { Liste der Hauptmenüpunkte (hier nur der Punkt "Demo"): }
    NewSubMenu('&Demo', 0, 'NoName', [
      { Liste der Menüpunkte des "Demo"-Menüs: }
      NewItem('&Tastenkürzel...', 0, False, True,
              P1Click, 0, ''),
      NewItem('&Zum Symbol verkleinern', 0, False, True,
              P2Click, 0, ''),
      NewItem('&Farbe wechseln', 0, False, True,
              P3Click, 0, '')
    ]) { Ende von NewSubMenu "Datei" }
    { hier könnten weitere Hauptmenüpunkte folgen... }
  ]); { Ende von NewMenu "Hauptmenü" }
end;
```

Der Text des Hauptmenüs ist am Bildschirm nicht sichtbar, wohl aber der einzige Menüpunkt des Hauptmenüs (DEMO) und seine drei Unterpunkte. Durch den dynamischen Aufbau der Menüpunkt-Arrays mit der Object-Pascal-Schreibweise *[...]* (siehe *Offene Arrays* in Kapitel 2.5.3) genügt für jeden Menüpunkt eine einzige Anweisung.

Die Struktur des Menüs kontrollieren Sie mit der Position dieser Aufrufe. Wenn Sie beispielsweise den Menüpunkt FARBE WECHSELN als Hauptmenüpunkt neben DEMO sehen wollen, verschieben Sie den entsprechenden *NewItem*-Aufruf einfach eine Schachtelungsebene nach oben hinter die beiden nächsten geschlossenen Klammern (eine runde und eine eckige) und passen die Kommasetzung entsprechend an. Um Trennlinien einzufügen, verwenden Sie statt *NewItem* einfach *NewLine*.

Das obige Beispiel verknüpft jeden Menüpunkt mit einer Methode des Formulars (*P1Click*, *P2Click*, *P3Click*). Falls Sie eine solche nicht schon vom Objektinspektor aus erstellt haben, müssen Sie sie vollständig von Hand erstellen. Die folgende Methode gehört zum obigen Beispiel und dem Menüpunkt ZUM SYMBOL VERKLEINERN (für Weiteres siehe *HLTest*).

```
procedure TForm1.P2Click(Sender: TObject);
begin
  Application.Minimize;
end;
```

Anders als in diesem Beispiel werden bei einer dynamischen Menüpunktliste oft alle Menüpunkte durch eine einzige Methode bearbeitet. Entweder muss diese Methode die verschiedenen Punkte über das *Tag*-Property unterscheiden, oder sie richtet sich nach dem Inhalt des *Caption*-Properties. Letzteres ist im Beispiel von Kapitel 4.6.2 der Fall.

Tastenkürzel

Ein besonderes Highlight der Unit *Menus* ist das Management der Tastenkürzel. Zwar können Sie auch diese bereits bequem im Objektinspektor festlegen, jedoch werden Tastaturkürzel noch häufiger zur Laufzeit angepasst als Menüs. Auch hier gibt die Delphi-IDE selbst wieder ein gutes Beispiel: Sie bietet im Umgebungsdialog vier verschiedene Tastaturbelegungen an. Wenn Sie hier eine andere Belegung auswählen, werden die geänderten Tastenkürzel automatisch auch im Menü angezeigt.

Damit reizt Delphi jedoch die Fähigkeiten der eigenen VCL noch keineswegs aus. Die VCL zwingt Sie nicht dazu, die gesamte Tastaturbelegung auf einmal auszuwechseln, sondern erlaubt Ihnen, das Tastenkürzel jedes einzelnen Menüpunkts zu ändern. Dazu benötigen Sie noch nicht einmal Tastencodes, sondern nur einen String, der diesen Code angibt, z.B. *'Strg+F1'*.

Das Property für das Tastenkürzel ist ja bereits im Objektinspektor ersichtlich: Jede *TMenuItem*-Komponente verfügt über ein *ShortCut*-Property, das Sie auch zur Laufzeit verändern können. Die VCL sorgt dann automatisch dafür, dass das angegebene Kürzel im Menü erscheint und dass es als Tastenkürzel funktioniert.

ShortCut hat jedoch den Typ *TShortCut*, bei dem es sich um einen Zahlencode handelt. Durch die Funktionen *TextToShortCut* können Sie diesen Code aus einem einfach zu verstehenden String berechnen lassen. Die Funktion *ShortCut* generiert den *TShortCut*-Wert aus einem Tastencode und einem *TShiftState*-Parameter, der den Status verschiedener umschaltbarer Tasten angibt. Die folgende Tabelle erklärt die Parameter und listet die beiden Umkehrfunktionen auf, die einen *TShortCut*-Code wieder zurück in einen lesbaren String oder in Tastencode und *ShiftState*-Angabe umwandeln:

Funktion	**Beschreibung**
function ShortCut(Key: Word; Shift: TShiftState): TShortCut;	liefert den Code für die in *Key* angegebene Taste, wenn diese mit den in *Shift* angegebenen Zusatztasten gedrückt wird.
procedure ShortCutToKey(ShortCut; var Key; var Shift);	liefert den Code, der dem *Text* entspricht. Gültige Parameter sind beispielsweise: 'F1', 'Strg+A' oder 'Alt+0'.
function ShortCutToText(ShortCut: TShortCut): String;	wandelt einen Tastencode in einen String um.
function TextToShortCut(Text: String): TShortCut;	wandelt einen Tastencode zurück in die Parameter der Funktion *ShortCut*.

4.6.2 Dynamische Menüerweiterungen

Dieses Kapitel gibt ein praktisches Anwendungsbeispiel für die im letzten Kapitel beschriebenen Hilfsfunktionen und für die Menüpunkteliste in *TMenuItem*. Dabei geht es um die dynamische Menüerweiterung der Klasse *THistoryList*, die Sie in der Unit *HList* auf der CD-ROM finden.

Die dynamische Erweiterung von Menüs ist wahrscheinlich am häufigsten notwendig, um Listen von Dateien oder Fenstern an bestehende Menüs anzuhängen. So finden Sie in Delphis Dateimenü eine Liste der zuletzt geladenen Projekte und im Fenstermenü einer MDI-Anwendung eine Liste der offenen Fenster. Während Sie für die MDI-Kindfensterliste auf die VCL vertrauen können, solange Sie das Property *WindowMenu* wie in Kapitel 5.7.4 beschrieben setzen, müssen Sie das Dateimenü selbst erweitern.

Erweitern des Dateimenüs

Um ein Dateimenü zu erweitern, benötigen Sie zuerst das *TMenuItem*-Objekt, das für das Dateimenü steht. Solange dieses Dateimenü sich immer an derselben Stelle befindet, können Sie einen festen Ausdruck angeben, um das gewünschte *TMenuItem*-Objekt zu erhalten. Falls sich das Dateimenü z.B. ganz links befindet, lautet dieser: *Menu.Items.Items[0]*. *Menu* ist dabei das Hauptmenü-Property des Formulars, *Items* ist die unsichtbare Wurzel des Menübaumes und in *Items.Items* sind schließlich die Hauptmenüpunkte gespeichert. Da die Liste *Items* in der Klasse *TMenuItem* als Standard-Array-Property deklariert ist, erhalten Sie das Dateimenü auch mit *Menu.Items[0]*. Den so erhaltenen Menüpunkt können Sie nun um die Dateinamen erweitern, indem Sie mit seiner *Add*-Methode neue *TMenuItem*-Objekte anfügen. Die dynamische Dateiliste des TreeDesigners sehen Sie in Abbildung 4.7.

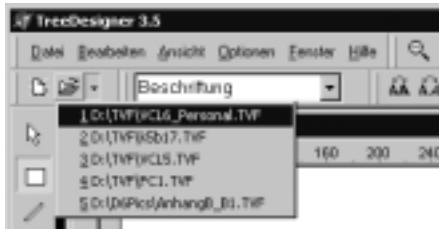


Abbildung 4.7: Die Datei-History-Liste des TreeDesigners kann über das Datei-Menü und über den Öffnen-Schalter der Symbolleiste aufgerufen werden.

Dateilisten im TreeDesigner

Beim TreeDesigner genügt es nicht, die Dateiliste an das Dateimenü des Hauptfensters anzuhängen, weil dieses beim Verschmelzen des Menüs mit einem Kindfenster-Menü (siehe Kapitel 5.7.3) durch das Dateimenü dieses Kindfensters ersetzt wird. Der Tree-

Designer muss daher die Dateiliste jedes Mal, wenn ein anderes Kindfenster aktiviert wird, mit dessen Dateimenü verknüpfen. Erst wenn einmal alle Kindfenster geschlossen sind, hängt er die Liste wieder an das Dateimenü des Hauptfensters an. Zuständig hierfür sind die Methoden *TDocumentForm.MdiActivate* und *TDocumentForm.FormClose*, im Folgenden sind diese Details jedoch ohne Bedeutung.

THistoryList

Die Klasse *THistoryList* aus Kapitel 4.1.2 verfügt über die Fähigkeit, die komplette von ihr verwaltete Stringliste an ein Menü anzuhängen. Wie schon erwähnt, ist dies meistens nur für die Liste der zuletzt geladenen Dateien notwendig. Bei der Verwendung einer *HistoryList* müssen Sie im Wesentlichen nur das *MenuItem*-Property dieser Liste setzen, die *THistoryList* hängt dann automatisch alle Strings an den angebenenen Menüeintrag an und aktualisiert das Menü bei jeder Änderung der History-Liste (auf die Details dieser Verwendung kommen wir später noch einmal zurück).

Da viele Anwendungen eine Dateiliste nicht nur im Hauptmenü, sondern auch als Popup-Menü für einen Schalter einer Symbolleiste anzeigen, verfügt *THistoryList* auch über das Property *PopupMenu*. Wenn es gesetzt ist, schreibt *THistoryList* die Dateinamen zusätzlich in dieses Popup-Menü (dieses muss vorher allerdings leer sein, da *THistoryList* es komplett neu aufbaut).

Zum Verständnis der Menümethoden benötigen Sie die folgenden Informationen über *THistoryList*:

```
type
  { Event, das bei Anwahl eines der automatisch zum Menü
    hinzugefügten Einträge ausgelöst wird: }
  TListMenuEvent = procedure(Sender: TObject; MenuText: string)
                     of object;

  THistoryList = class(TStringList)
  { Stringliste, die sich selbst an ein Menü anhängen kann und
    sich in die Registry/eine INI-Datei speichern und von dort
    wieder lesen kann }
  private
    FMenuItem: TMenuItem;
    FPopupMenu: TPopupMenu;
    MenuSizeBefore: Integer; { Speichert die vorherige
                  Menügröße zur späteren Rekonstruktion }
    FOnAutoItemClick: TListMenuEvent;

    procedure SetMenuItem(MenuItem: TMenuItem);
    procedure SetPopupMenu(PopupMenu : TPopupMenu);
    procedure AddMenuItems; // ruft AddToMenuItem und AddToPopupMenu auf
      procedure AddToMenuItem; // Liste nur an den Hauptmenüpunkt anhängen
      procedure AddToPopupMenu; // Liste nur in das Popup-Menü schreiben
    procedure RemoveMenuItems; // ruft beide Remove...-Methoden auf
```

```
      procedure RemoveFromMenuItem; // entfernt die Liste aus FMenuItem
      procedure RemoveFromPopupMenu; // entfernt die Liste aus FPopupMenu
    procedure DoAutoItemClick(Sender: TObject);
       { empfängt die OnClick-Ereignisse }
  public
    { Funktionsbereich Menü: sobald "Menu" einmal gesetzt ist,
      passt die Liste das Menü bei jeder Veränderung an. }
    property MenuItem: TMenuItem read FMenuItem write SetMenuItem;
    property PopupMenu : TPopupMenu read FPopupMenu write SetPopupMenu;
    property OnAutoItemClick: TListMenuEvent read FOnAutoItemClick
                                          write FOnAutoItemClick;
    ...
```

Bei der Menüerweiterung spielen vier Datenelemente dieser Klasse eine wichtige Rolle:

- *Strings*, die von *TStringList* geerbte Stringliste, die die Dateinamen enthält (oder die anderen Strings, die an das Menü angehängt werden sollen). Wie diese Liste verwaltet wird, lesen Sie in Kapitel 4.1.2; in diesem Kapitel können wir diese Stringliste als fest vorgegeben betrachten.
- Das Menü, an das die Strings dieser Liste angehängt werden sollen, speichert *THistoryList* im Datenfeld *FMenuItem*.
- Für das gleichzeitig verwendbare Popup-Menü gibt es entsprechend eine *FPopupMenu*-Variable.
- *MenuSizeBefore* speichert die Zahl der Menüpunkte, bevor die Liste an *FMenuItem* angehängt wird, damit die Liste später wieder entfernt und aktualisiert werden kann. Die Zahl der Menüpunkte des Popup-Menüs wird nicht gespeichert, da dieses Menü wie erwähnt immer neu aufgebaut wird, also gar keine anderen Punkte enthalten soll.

Wie Sie der obigen Deklaration entnehmen können, wird *FMenuItem* von außen über das Property *MenuItem* angesprochen. Jedes Mal, wenn *MenuItem* von außen gesetzt wird, wird die Methode *SetMenuItem* aufgerufen. *SetMenuItem* löscht zuerst mit *RemoveFromMenuItem* alle zuletzt angefügten Menüpunkte, speichert die neue Zahl der Menüpunkte in *MenuSizeBefore* und fügt die derzeitige Liste mit *AddToMenuItem* zum neuen Menü hinzu:

```
procedure THistoryList.SetMenuItem(MenuItem: TMenuItem);
begin
  { ein eventuell schon verändertes Menü in seinen alten
    Zustand zurückversetzen: }
  if Assigned(FMenuItem) then
    RemoveFromMenuItem;
  FMenuItem := MenuItem; { Property-zugehörige Variable setzen }
  MenuSizeBefore := MenuItem.Count; { bisherige Menügröße speichern }
  AddToMenuItem; { ab sofort hält THistoryList dieses Menü aktuell }
end;
```

Menüpunkte hinzufügen und löschen *R65*

Wir sehen uns nun die von *SetMenu* aufgerufenen Methoden *AddToMenuItem* und *RemoveFromMenuItem* an. *AddToMenuItem* kann davon ausgehen, dass das Menü noch in seinem Ursprungszustand ist, dass also alle eventuell früher schon angefügten Menüpunkte mit *RemoveFromMenuItem* wieder entfernt worden sind.

Hinweis: Dies wird dadurch sichergestellt, dass alle Aufrufer von *AddToMenuItem* oder *AddMenuItems* vorher *RemoveFromMenuItem* aufrufen. Die Aufrufer dieser Methoden sind *SetMenu* (siehe oben) und *AddString*(Kapitel 4.1.2), von außen können diese Methoden nicht aufgerufen werden, da sie privat sind.

AddToMenuItem hängt zuerst eine Trennlinie, erzeugt durch die Funktion *NewLine*, an die vorhandenen Menüpunkte an. Danach durchläuft sie in einer Schleife alle Strings der Liste und erzeugt mit der Funktion *NewItem* für jeden String einen neuen Menüpunkt. Zum Anfügen der Trennlinie und der Dateinamen verwendet sie die *TMenuItem*-Methode *Add*:

```
procedure THistoryList.AddToMenuItem;
var
  i: Integer;
begin
  if Assigned(FMenuItem) then begin
    FMenuItem.Add(NewLine); { mit einem Separator abtrennen }
    for i := 0 to Count-1 do
      { Der Menütext wird mit der komfortablen Format-Funktion
        zusammengesetzt: }
      FMenuItem.Add(NewItem(Format('&%d %s', [i+1, Strings[i]]), 0,
                            False, True, DoAutoItemClick, 0, ''));
  end;
end;
```

AddToMenuItem nummeriert die neuen Menüpunkte durch, wobei es mit der Ziffer 1 beginnt. Sie übergibt der Methode *NewItem* einen Menütext, der sich zusammensetzt aus dem Zeichen »&« (für die Unterstreichung), aus der Nummer des Eintrags, aus einem Leerzeichen und schließlich aus dem Dateinamen (zur Funktion *Format* siehe Kapitel 2.8.4).

RemoveItems löscht die Einträge wieder. Sie löscht so lange den letzten Eintrag der *TMenuItem*-Liste, bis die Liste dadurch so klein geworden ist wie beim ursprünglichen Menü (*MenuSizeBefore*).

```
procedure THistoryList.RemoveFromMenuItem;
begin
  { alle Menüeinträge ab der gespeicherten Position
    MenuSizeBefore wieder entfernen: }
  if Assigned(FMenuItem) then
```

```
    while FMenuItem.Count > MenuSizeBefore do
      FMenuItem.Delete(FMenuItem.Count-1);
end;
```

Diese beiden Methoden werden jedes Mal aufgerufen, wenn die History-Liste verändert wird (siehe Methode *THistoryList.AddString* in Kapitel 4.1.2).

Hinweis: Die Methoden *AddToPopupMenu* und *RemoveFromPopupMenu* funktionieren ähnlich und sind daher nicht abgedruckt.

Verknüpfen der OnClick-Ereignisse

NewItem erwartet im vierten Parameter eine Methode, die bei jedem *OnClick*-Ereignis des neuen Menüpunkts aufgerufen werden soll. Die obige Methode *AddToMenuItem* übergibt in diesem Parameter jedes Mal die Methode *DoAutoItemClick*. Diese wird also jedes Mal aufgerufen, wenn zur Laufzeit einer der hinzugefügten Strings bzw. Dateinamen aus dem Menü ausgewählt wird.

Wie in der allgemeinen Beschreibung schon erwähnt, steht diese Methode nun vor dem Problem, zwischen den einzelnen Menüpunkten zu unterscheiden. Sie benötigt dazu lediglich das *Caption*-Property des Menüpunkts, der im Parameter *Sender* übergeben wird:

```
procedure THistoryList.DoAutoItemClick(Sender: TObject);
var
  Text: string;
  Index: Integer;
begin
  if Assigned (FOnAutoItemClick) then begin
    { Caption beinhaltet den Menüpunkt-Text: }
    Text := (Sender as TMenuItem).Caption;
    { Text sieht so aus:"&12 MenuText"
      das "&" muss übersprungen werden, dann folgt bis
      zum Leerzeichen der Index, der gewählt wurde. }
    Index := StrToInt(Copy(Text, 2, Pos(' ', Text)-2));
    { Index - 1 ergibt den Index in der internen Stringliste: }
    if Index > 0 then
      FOnAutoItemClick(FMenuItem, Strings[Index-1]);
  end;
end;
```

DoAutoItemClick findet zuerst die Nummer des Menüpunkts heraus, die von *AddToMenuItem* vor den Text geschrieben wurde (mit Hilfe der Standardfunktionen *Copy*, *Pos* und *StrToInt*). Mit Hilfe dieser Nummer findet sie in ihrer eigenen Liste schließlich den zugehörigen String (bzw. Dateinamen).

Hinweis: Natürlich hätte die Methode den Dateinamen auch direkt im Property *Caption* finden können, allerdings wurde hier zuerst der Index gesucht in der Annahme, dass er sich vielleicht später einmal sinnvoll verwerten lässt.

Was schließlich mit dem angeklickten String oder Dateinamen geschehen soll, kann das *THistoryList*-Objekt nicht wissen, dafür muss der Benutzer dieses Objekts sorgen. *THistoryList* stellt diesem dafür das Event *OnAutoItemClick* zur Verfügung. Innerhalb der Methode *DoAutoItemClick* wird dieses Menü über den Methodenzeiger *FOnAutoItemClick* ausgelöst; mehr über das Auslösen von eigenen Events finden Sie in Kapitel 6.3.3.

Verwendung von THistoryList *R46*

Wenn Sie *THistoryList* in eigenen Programmen verwenden wollen, müssen Sie die folgenden vier hier beschriebenen Schritte nachvollziehen:

- 1.Initialisierung und Laden der History-Liste, z.B. beim *OnCreate*-Event
- 2.Schreiben einer Methode für die automatischen Menüpunkte, also für das Ereignis *OnAutoItemClick*
- 3.Erweitern der History-Liste an den geeigneten Stellen
- 4.Speichern der History-Liste, z.B. im *OnDestroy*-Event

Der TreeDesigner aus Kapitel 5 demonstriert die Anwendung von *THistoryList* zur Speicherung einer Liste der zuletzt geladenen Dateien. Die genannten vier Schritte lassen sich in der Unit für das MDI-Hauptformular *MdiForm* wiederfinden und sind im Folgenden einzeln besprochen.

Initialisieren und Laden der History-Liste

In der *OnCreate*-Methode des TreeDesigner-Hauptformulars geschehen die folgenden unter Punkt 1 erwähnten Dinge:

```
{ Auszug aus MainFormCreate: }
IniPath := 'Software\TreeDesigner\';
FileList.UseRegistry := True;
FileList := THistoryList.Create;
FileList.LoadFromIni(IniPath+'TREEDSGN.INI', 'FILES');
FileList.PopupMenu := RecentFilesPopup;
FileList.MenuItem := TLM_File;
FileList.OnAutoItemClick := AutoItemClick;
```

FileList, ein *THistoryList*-Objekt, wird wie üblich per Konstruktoraufruf erzeugt. Dann werden die Namen der zuletzt benutzten Dateien, die beim letzten Schließen des Programms gespeichert wurden, wieder geladen. Die Methode *LoadFromIni* erwartet hierzu

im ersten Parameter den Namen eines Registry-Schlüssels bzw. einer INI-Datei und im zweiten den Namen des Abschnitts, unter dem die Dateinamen gespeichert werden sollen. Der TreeDesigner verwendet unter 32-Bit-Windows grundsätzlich die Registry, setzt also das *UseRegistry*-Property von *THistoryList* auf *True* (der Umgang von *THistoryList* mit der Windows-Registry ist in Kapitel 4.2.4 ausführlich beschrieben).

Als Nächstes setzt die Methode die Properties, die die zu erweiternden Menüs angeben. *Datei1* ist der Name des *TMenuItem*-Objekts für den Menüpunkt DATEI (um diesen Namen im Objektinspektor zu sehen, müssen Sie den Menü-Editor aufrufen). *RecentFilePopup* ist der Name einer *TPopupMenu*-Komponente.

Die letzte Zeile im obigen Codeausschnitt setzt das Property für das Event *OnAutoItemClick*, das von der Methode *DoAutoItemClick* aufgerufen wird, jedes Mal, wenn einer der automatisch erzeugten Menüpunkte aufgerufen wird.

Schreiben einer Methode für OnAutoItemClick

Die Methode des Formulars für dieses Ereignis heißt *AutoItemClick*. Wie aus der Deklaration von *TListMenuEvent* auf Seite 532 ersichtlich, ist der erste Parameter der übliche *Sender*, im zweiten Parameter befindet sich der Menütext (ohne die vorangehende Ziffer).

AutoItemClick weiß, dass es sich bei diesem Menütext nur um einen Dateinamen handeln kann, und lädt diesen mit der hier nicht weiter interessanten Methode *LoadFromFile* in ein Fenster, das mit der hier ebenfalls uninteressanten Methode *NewWindow* geöffnet wird. Punkt 2 der obigen Aufzählung sieht damit im TreeDesigner wie folgt aus:

```
procedure TMainForm.AutoItemClick(Sender: TObject; FileName: string);
begin
  NewWindow.LoadFromFile(FileName);
end;
```

Wie die Dateinamen in die Liste kommen

Wir sind nun beim dritten Schritt der obigen Aufzählung angekommen: beim Erweitern der Dateiliste an den geeigneten Stellen. Hierfür stellt *THistoryList* die Methode *AddString* zur Verfügung. Sie können diese aufrufen, wenn eine Datei unter neuem Namen gespeichert wird oder wenn eine neue Datei geöffnet wird. Im TreeDesigner ist die Methode *MDIOpenClick* für das Öffnen der Datei zuständig:

```
procedure TMainForm.MDIOpenClick(Sender: TObject);
var
  NewChild: TDocumentForm;
begin
  if OpenDialog1.Execute then begin
    NewChild := NewWindow;
    NewChild.LoadFromFile(OpenDialog1.FileName);
```

```
      { Datei-History-Liste erweitern: }
      FileList.AddString(OpenDialog1.FileName);
    end;
  end;
```

AddString erweitert die History-Liste um den neuen Dateinamen und aktualisiert das Menü, indem sie die Methoden *AddMenuItems* und *RemoveMenuItems* aufruft. Weitere Informationen zu *THistoryList* und zur Methode *AddStrings* finden Sie in Kapitel 4.1.2.

Genauso können Sie die History-Liste erweitern, wenn eine Datei unter neuem Namen gespeichert wird (im TreeDesigner wird das Speichern vom Dokumentfenster übernommen).

Speichern in einer INI-Datei

Als letzter Schritt bleibt nun noch das »*Speichern der History-Liste, z.B. im OnDestroy-Event*«, denn ein wesentlicher Vorteil der History-Liste soll ja sein, dass der Anwender auch auf Dateien schnell zugreifen kann, die er in früheren Sitzungen geladen hat. Als Gegenstück zum *LoadFromIni*-Aufruf in *FormCreate* findet daher in der Methode zum *OnDestroy*-Event ein Aufruf von *SaveToIni* statt:

```
procedure TMainForm.FormDestroy(Sender: TObject);
begin
  FileList.SaveToIni(IniPath+'TREEDSGN.INI', 'FILES');
  FileList.Free;
end;
```

Der Aufruf von *Free*, der den manuellen *Create*-Aufruf in *FormCreate* wieder aufhebt, sollte nicht fehlen, denn da *FileList* keine Komponente ist, wird sie bekanntlich nicht automatisch von der VCL gelöscht.

4.6.3 Ein dynamischer Tastenkürzeleditor

Mit den in Kapitel 4.6.1 beschriebenen Funktionen können wir leicht einen Tastenkürzeleditor schreiben, der in alle Anwendungen eingebunden werden kann und der dort die Tastenkürzel von beliebigen Menüs modifizieren kann. Ein solcher Editor sollte natürlich in der Lage sein, die Menükonfiguration in einer Datei zu speichern.

Auf der CD-ROM finden Sie den Tastenkürzeleditor als Formular in der Unit *HKeyForm* und als Komponente *THotkeyManager* in der Unit *HKeyMan* (siehe Kapitel 6.7); eine Komponente der Klasse *THotkeyManager* ist im Dokumentformular des *TreeDesigners* aus Kapitel 5 eingebunden, zur Laufzeit rufen Sie den Editor dort über das Menü OPTIONEN | TASTENKÜRZEL-EDITOR... auf.

Da die Methoden des Tastenkürzeleditors mit ähnlichen Mitteln arbeiten wie die Methoden zur dynamischen Menüerweiterung aus dem letzten Kapitel, ist hier nur ein kleiner Teil des Quelltextes abgedruckt.

Tastenkürzel für Aktionslisten

Ebenfalls nicht abgedruckt sind die Programmteile, die den Editor befähigen, auch die Tastenkürzel einer Aktionsliste zum Editieren freizugeben. Ob ein Menü oder eine Aktionsliste editiert wird, entscheidet sich daran, welches Property von *THotkeyEditor* Sie setzen: Bei Zuweisung an das Property *Menu* wird das Property *ActionList* automatisch gelöscht und umgekehrt. Wie die Abbildung zeigt, verwendet der Editor in beiden Fällen denselben Fensteraufbau, lediglich die Beschriftungen sind unterschiedlich.

Ein *ActionList*-Property finden Sie natürlich auch in der Komponente *THotkeyManager*. Wenn Sie alle Aktionen Ihrer Anwendung in einer Aktionsliste verwalten, kann *THotkeyManager* leicht zum Editieren aller Tastenkürzel einer Anwendung dienen. Mit den Aktionslisten wird sich Kapitel 4.6.4 beschäftigen.

Bedienung

Abbildung 4.8 zeigt eine aktive Version des Editors. Seine erste Liste enthält schon nach dem Öffnen des Fensters die Punkte des Hauptmenüs (bzw. die Kategorien der Aktionsliste). Wählen Sie einen Eintrag dieser Liste aus, so füllt der Editor die zweite Liste mit den zugehörigen Untermenüpunkten (bzw. den Aktionen der gewählten Kategorie), wobei er die bereits aktiven Tastenkürzel in Klammern anzeigt. Im Editierfeld unter der ersten Liste geben Sie das Tastenkürzel ein, indem Sie einfach die entsprechenden Tasten drücken.

Abbildung 4.8: Der Tastenkürzeleditor in HKeyForm

Das Dateimenü des Editors enthält die beiden Punkte *Laden* und *Speichern*, mit denen Sie manuell verschiedene Dateien mit Tastenkürzeln verwalten können; normalerweise sollte jedoch die in Kapitel 6.7 beschriebene automatische Speicherung der Tastaturbelegung ausreichen.

Einbindung des Tastenkürzeleditors in eigene Programme

Kapitel 6.7 beschreibt, wie Sie den Tastenkürzeleditor über eine Komponente zur Entwurfszeit in das Formular einbinden. Auch ohne diese Hüllkomponente können Sie den Editor leicht in eigenen Programmen verwenden. Nachdem Sie die Unit *HKeyForm* optional in Ihr Projekt (PROJEKT | DEM PROJEKT HINZUFÜGEN) und die Unit in die *uses*-Anwendung der passenden Unit eingefügt haben, können Sie den Editor wie in der folgenden Methode aufrufen.

```
procedure TDocumentForm.Tastenkrzel1Click(Sender: TObject);
begin
  HotkeyEditor.Menu := Menu; { "Menu" ist das Formular-Property }
{ oder, für das Editieren der Aktionen einer Aktionsliste:
    HotkeyEditor.ActionList := ActionList1; }
  HotkeyEditor.ShowModal;
end;
```

Diese Zeilen setzen voraus, dass das Formular des Tastenkürzeleditors in den Projektoptionen als automatisch zu erzeugen angegeben ist. Mit einer dynamischen Erzeugung könnten Sie während der Zeit, in der der Editor nicht angezeigt wird, Ressourcen sparen, siehe dazu Kapitel 3.4.4.

Nach der Ausführung von *ShowModal* werden die neuen Tastenkürzel im Menü (bzw. bei entsprechender Verwendung einer Aktionsliste in allen Menüs) angezeigt und können sofort verwendet werden. Um die Tastenkürzel automatisch zu speichern, müssten Sie nun noch die Methoden *SaveToFile* und *LoadFromFile* an einer geeigneten Stelle aufrufen (z.B. bei *OnCreate* und *OnDestroy* oder direkt vor und hinter dem Aufruf von *ShowModal*, falls Sie den Tastenkürzeleditor auf diese Weise aufrufen). Ein Beispiel für den Aufruf der Speichern- und Lademethoden finden Sie in der Hüllkomponente für diesen Editor in Kapitel 6.7.

Die THotkey-Komponente

Im Formular des Tastenkürzeleditors verrichtet ein *THotkey*-Eingabefeld nützliche Dienste. Es gibt die Bezeichnungen der Tastenkombination aus, die Sie drücken, während das Editierfeld den Tastaturfokus besitzt, und zwar immer nur die zuletzt eingegebene Tastenkombination. Wenn Sie beispielsweise Strg+Einfg drücken, zeigt das Feld zunächst »Strg +« an, solange Sie nur Strg drücken. Wenn Sie dann zusätzlich Einfg drücken, ändert sich die Anzeige in »Strg + Num 0« .

Ihre Anwendung erhält das eingegebene Tastenkürzel über das Property *Hotkey* in Form eines *TShortCut*-Wertes zurück. Der Tastenkürzeleditor kann dieses direkt dem *ShortCut*-Property eines Menüpunkts zuweisen (ohne Umweg über die Hilfsfunktion *TextToShortCut*, siehe Methode *ZuweisenClick* unten). Das Property *Modifiers* enthält zusätzlich noch eine Menge von Flags, die angeben, welche der »Zusatztasten« Alt, Strg und Shift zum Tastenkürzel gehören (diese Information ist auch schon in einem *TShortCut*-Wert und damit im Property *Hotkey* enthalten).

Erwähnenswert ist schließlich das Property *InvalidKeys*, in dem Sie einzelne oder bestimmte Kombinationen von Zusatztasten verbieten können (z.B. *hcShiftCtrl*, wenn das Tastenkürzel nicht genau die beiden Zusatztasten Shift und Strg enthalten darf). Der Tastenkürzeleditor benötigt jedoch lediglich das *Hotkey*-Property.

Implementierung

Wir wenden uns noch einigen Ausschnitten aus dem Code zu. In diesem kommen nun die in Kapitel 4.6.1 behandelten Tastenkürzelfunktionen zum Einsatz. Beim Aktualisieren der zweiten Liste verwendet *UpdateSubItems* die Funktion *ShortCutToText*, um das Tastenkürzel in eine lesbare Form umzuwandeln (*SubMenu* ist eine Variable des Formulars, sie zeigt immer auf das in der ersten Liste gewählte Menü):

```
procedure THotkeyEditor.UpdateSubItems;
{ Baut "ListBox2" neu auf und füllt sie mit den
  Punkten des aktuellen "SubMenu" }
var
  i: Integer;
  Index: Integer;
begin
  { gewählten Listeneintrag merken: }
  Index := ListBox2.ItemIndex;
  ListBox2.Clear;
  for i := 0 to SubMenu.Count-1 do begin
    ListBox2.Items.Add(
      SubMenu[i].Caption+' '
       +ShortCutToText(SubMenu[i].ShortCut)+')');
  end;
  { bisher gewählten Listeneintrag wiederherstellen: }
  ListBox2.ItemIndex := Index;
end;
```

Falls der Tastenkürzeleditor keine *THotkey*-Komponente, sondern eine Kombobox zur Eingabe des Tastenkürzels enthält (dies ist in der Delphi-1-Version des Editors der Fall), muss er die als String angegebene aktuelle Auswahl der Kombobox in einen *TShortCut*-Wert umrechnen. Hier kann sich die VCL-Methode *TextToShortCut* nützlich machen:

```
procedure THotkeyEditor.ComboBox1Change(Sender: TObject);
begin
  if Assigned(Selected) then begin
    Selected.ShortCut := TextToShortCut(ComboBox1.Text);
    UpdateSubItems;
  end;
end;
```

Dabei ist *Selected* eine Variable des Formulars, die immer auf den in der zweiten Liste gewählten Eintrag weist.

Bei Eingaben in die *THotkey*-Komponente ist eine solche Umwandlung nicht notwendig, da das *ShortCut*-Property dieser Komponente den gewünschten Code bereits enthält. Allerdings hat *THotkey* weder ein *OnChange*- noch *OnKey...*-Ereignis. Da aber das *KeyPreview*-Property des Editorformulars auf *True* gesetzt wurde, kann der Editor trotzdem beim *OnKeyUp*-Ereignis des Formulars auf jede Eingabe in diese Komponente reagieren (siehe Quelltext auf der CD-ROM).

Speichern der Tastenkürzel

Wie in Kapitel 4.3 angekündigt, gibt es nun ein Beispiel für die Verwendung der Methoden von *TWriter* und *TReader* im Zusammenhang mit einem *TStream*-Objekt, insbesondere dem Schreiben und Lesen von Listen. Untersucht werden die Methoden des Tastenkürzeleditors, die das Speichern und das Laden der Tastenkürzel durchführen.

Zuerst zur Methode für das Speichern. *SaveToFile* schreibt einfach für jeden Menüpunkt, der ein Tastenkürzel hat, drei Zahlen in die Datei: den Index des Hauptmenüpunkts, zu dem der Menüpunkt gehört, den Index des Menüpunkts innerhalb des Untermenüs und den Zahlencode des Tastenkürzels. Geschachtelte Untermenüs werden also – wie beim Editieren auch – ignoriert. Auch hier müssen wieder alle Punkte eines *Items*-Properties in einer Schleife durchlaufen werden, dieses Mal sogar in zwei geschachtelten Schleifen.

```
procedure THotkeyEditor.SaveToFile(FileName: string);
var
  MainIndex, SubIndex: Integer;
  SubMenu: TMenuItem;
  CurrentItem: TMenuItem;
  S: TStream;
  W: TWriter;
begin
  S := TFileStream.Create(FileName, fmCreate);
  W := TWriter.Create(S, 2000);
  W.WriteListBegin;
  { für alle Hauptmenüpunkte: }
  for MainIndex := 0 to FMenu.Items.Count-1 do begin
    SubMenu := FMenu.Items[MainIndex];
```

```
      { für alle Untermenüpunkte: }
      for SubIndex := 0 to SubMenu.Count-1 do begin
        CurrentItem := SubMenu.Items[SubIndex];
        { nur speichern, wenn ein Tastenkürzel existiert: }
        if CurrentItem.ShortCut <> 0 then begin
          W.WriteInteger(MainIndex);
          W.WriteInteger(SubIndex);
          W.WriteInteger(CurrentItem.ShortCut);
        end;
      end;
    end;
    W.WriteListEnd;
    W.Free;
    S.Free;
  end;
```

Da die *TWriter*-Methoden bereits in Kapitel 4.3.4 erläutert wurden, können wir uns sofort dem Laden der Datei zuwenden.

Lesen der Tastenkürzeldatei

Die Methode zum Lesen kann keine interessanten neuen Aktionen in Sachen Menüs mehr durchführen, dient hier also in erster Linie als Fortsetzung des *TReader* / *TWriter*-Beispiels und wird hier nur soweit dargestellt, wie es die Arbeit des *TReader*-Objekts betrifft. Dieses muss die Daten genauso aus der Datei lesen, wie sie in *SaveFile* vom *Writer* gespeichert wurden.

Eine Besonderheit ist, dass der Benutzer aus Versehen eine Datei laden kann, die für ein ganz anderes Menü gedacht war. Hierauf bereitet sich *LoadFromFile* mit einer Exception-Abfrage vor. Anstatt jeden aus der Datei geladenen Menüindex (*MainIndex* und *SubIndex*) daraufhin zu überprüfen, ob es ihn wirklich gibt, probiert die Methode einfach aus, ob die Verwendung der Menüpunkt-Indizes eine Exception hervorruft (das Schlüsselwort *try* ist hier also wörtlich zu nehmen):

```
// Auszug aus THotkeyEditor.LoadFromFile:
// [von SaveToFile abweichende Variablen:
//    R: TReader; Errors: Boolean; ShortCut: Integer;]
R := TReader.Create(S, 2000);
R.ReadListBegin;
while not R.EndOfList do begin
  MainIndex := R.ReadInteger;
  SubIndex := R.ReadInteger;
  ShortCut := R.ReadInteger;
  try
    SubMenu := FMenu.Items[MainIndex];
    CurrentItem := SubMenu.Items[SubIndex];
    CurrentItem.ShortCut := ShortCut;
  except
    on EListError do Errors := True;
```

```
    end;
  end;
  R.ReadListEnd;
  R.Free;
```

Falls es in einem Fall zu einer *EListError*-Exception kommt, setzt *LoadFromFile* das Flag *Error*, damit sie am Schluss eine warnende Dialogbox anzeigen kann. Alle anderen Exceptions werden nicht behandelt und führen zu einem Abbruch der Methode.

Natürlich führt die Verwendung der *TWriter*- und *TReader*-Methoden nicht gerade zu übermäßiger Effizienz; da es aber bei den kleinen Tastenkürzeldateien auch so nicht zu Wartezeiten kommen sollte, ist dieses Problem eher theoretischer Natur.

4.6.4 Befehlslisten mit TActionList

Viele moderne Anwendungen stimmen darin überein, dass sie Menübefehle und Aktionsschalter in Symbolleisten gleich behandeln – insbesondere in beiden dieselben Bildsymbole verwenden, so dass der Benutzer erkennen kann, dass Menüpunkt und Schalter, die dieselbe Funktion verrichten, eine Einheit bilden. Spätestens anlässlich der Implementierung einer solchen Oberfläche drängt es sich auch für den Entwickler auf, Menüpunkte und zugehörige Schalter gemeinsam zu behandeln. Nun kann man in Delphi zwar einem Schalter-*OnClick*-Ereignis dieselbe Methode zuweisen wie einem Menüpunkt-*OnClick*-Ereignis. Aber damit ist es ja nicht getan, denn folgende Dinge müssen Sie in diesem Fall immer noch getrennt festlegen, einmal für den Schalter und einmal für den Menüpunkt:

- die statischen Daten, die sich zur Laufzeit normalerweise nicht ändern: die Beschriftung (*Caption*), der Bildindex (*ImageIndex*), der Hinweistext (*Hint*) und der Hilfekontext (*HelpContext*)
- die dynamischen Daten, die sich zur Laufzeit in vielen Fällen ändern können: der Markierungszustand (*Checked*), der Aktivierungszustand (*Enabled*) und der Sichtbarkeitszustand (*Visible*), vielleicht auch das Tastenkürzel (*ShortCut*).

Unter Verwendung von Aktionslisten (verfügbar ab Delphi 4) brauchen Sie all diese Daten jeweils nur ein einziges Mal festzulegen, und – was noch besser ist – wenn Sie diese Daten zur Laufzeit ändern, wirkt sich diese Änderung sowohl auf den Schalter als auch auf den Menüpunkt aus.

TAction

Das zugrunde liegende Konzept besteht darin, dass die Aktion, die von beiden gleichermaßen ausgeführt werden soll, in einem neuen Objekt, einem Aktionsobjekt der Klasse *TAction* zusammengefasst wird.

TAction verfügt über alle in obiger Aufzählung genannten Properties und natürlich auch über ein *OnExecute*-Ereignis, das mit der Methode verknüpft wird, die die Aktion ausführt. Wenn Sie ein *TAction*-Objekt mit einem Menüpunkt oder einem Schalter verknüpfen, übernehmen Menüpunkt bzw. Schalter diese Daten einfach. Um die Verknüpfung herzustellen, brauchen Sie lediglich das *Action*-Property des Schalters bzw. des Menüpunkts auf das *TAction*-Objekt zu setzen.

Die Komponente, die zum Auslösen einer Aktion verwendet wird, also z.B. ein Menüpunkt, wird als *Client* der Aktion bezeichnet. Welche Properties von *TAction* in den Client der Aktion übernommen werden, bestimmen die *TActionLink*-Klassen der VCL, um die Sie sich normalerweise nicht zu kümmern brauchen. Es kann jedenfalls sein, dass ein Property von *TAction* in ein verschieden benanntes Property des Clients kopiert wird (so übernimmt *TToolButton.Down* beispielsweise den Wert von *TAction.Checked*).

TActionList

Die Komponentenklasse *TActionList* hat drei wichtige Aufgaben:

- Offensichtlich liegt der organisatorische Zweck darin, die häufig in großen Scharen auftretenden Aktionen von Programmen in einer gemeinsamen Liste zu verwalten. Einen Editor für diese Liste erhalten Sie mit einem Doppelkick auf eine *TActionList*-Komponente (Abbildung 4.9). Zur besseren Übersicht können Sie Aktionen auch in Kategorien aufteilen. Jedes *TAction*-Objekt verfügt hierzu über ein *Category*-Property, das der Aktionslisten-Editor verwendet, um die Aktionen in Kategorien zu sortieren.
- *TActionList* bietet auch den einzigen Weg, ein *TAction*-Objekt zur Entwurfszeit zu erzeugen, denn *TAction* selbst ist nicht in der Komponentenpalette vertreten. Im Aktionslisten-Editor können Sie mit Einfg bzw. mit dem entsprechenden Menüpunkt oder Schalter neue Aktionsobjekte erzeugen, deren Properties Sie dann über den Objektinspektor einstellen können.
- Schließlich speichert eine *TActionList* im Property *Images* einen Verweis auf die Bilderliste, auf die sich die *ImageIndex*-Angaben aller in ihr enthaltenen *TAction*-Objekte beziehen. Dies ist auch schon das einzige Property dieser Klasse – abgesehen von den obligatorischen *Name*- und *Tag*-Angaben.

Es ist übrigens nicht notwendig, die *TMainMenu*- oder *TPopupMenu*-Komponenten mit den *TActionList*-Objekten zu verknüpfen, sondern es genügt, wenn die einzelnen Menüpunkte mit *TAction*-Objekten verknüpft sind. Diese dürfen theoretisch auch aus verschiedenen Aktionslisten stammen.

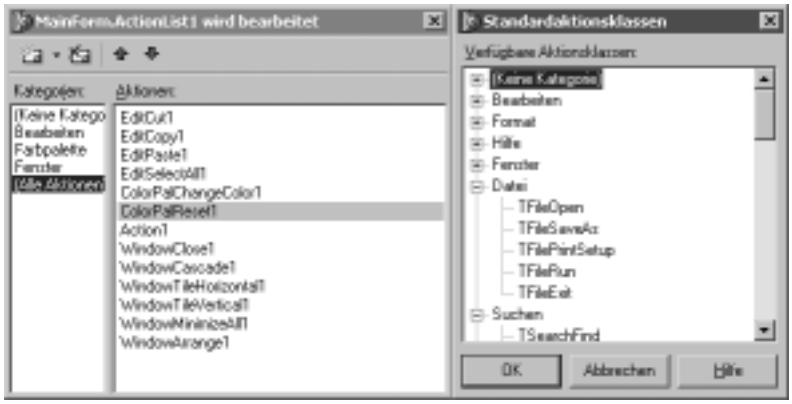


Abbildung 4.9: Zusammenstellen eigener Aktionslisten (links) aus den mit der VCL mitgelieferten Standardaktionen (rechts). Die beiden ColorPal...-Aktionen sind in der Komponentenbibliothek dieses Buchs definiert (Kategorie »Farbpalette«).

Praktischer Einsatz *R I I*

In der Praxis sieht die Verwendung der Aktionslisten so aus, dass Sie für alle Befehle, die der Benutzer Ihrer Anwendung ausführen soll, eine eigene Aktion in einer *TAction-List*-Komponente definieren. Bei umfangreichen Anwendungen bietet es sich an, diesen Aktionen verschiedene Kategorien (*Category*-Property) zuzuweisen.

Statt die *OnClick*-Ereignisse von Schaltern und Menüpunkten zu bearbeiten, schreiben Sie eine Methode für das *OnExecute*-Ereignis der Aktion. Auch alle schon genannten weiteren Parameter wie etwa Tastenkürzel und Bildsymbol-Index legen Sie in der Aktion fest. Bei Schaltern und Menüpunkten müssen Sie dann als Einziges noch das *Action*-Property setzen.

Beispiele zur grundsätzlichen Anwendung dieses Aktionskonzepts finden Sie sowohl im TreeDesigner aus Kapitel 5 (für die Befehle des Ansichtsmenüs) als auch in der von Kylix bereitgestellten Projektschablone für eine *MDI-Anwendung* (DATEI | NEU / PROJEKTE / MDI-ANWENDUNG).

Beispiele zur grundsätzliche Anwendung dieses Aktionskonzepts finden Sie sowohl im TreeDesigner aus Kapitel 5 (für die Befehle des Ansichtsmenüs) als auch in Delphis Projektschablone für eine Anwendung mit dem Windows-Logo (DATEI | NEU..., PROJEKTE, WIN95-LOGO ANWENDUNG).

Ereignisse von TAction *R2 I*

Die Klasse *TAction* verfügt noch über zwei interessante Ereignisse, die Sie im Objektinspektor mit Methoden verknüpfen können:

- *OnUpdate* wird während des Wartezustandes der Anwendung aufgerufen, um Ihnen die Gelegenheit zu geben, die Aktivierbarkeit der Aktion zu überprüfen und das *Enabled*-Property entsprechend zu setzen (ggf. auch das *Checked*-Property). Bei Aktionen, die sich auf ein Editierfeld beziehen, können Sie z.B. die Aktivierbarkeit der Aktion von der Frage abhängig machen, ob im Editierfeld Text selektiert ist:

  ```
  procedure TForm1.EditieraktionUpdate(Sender: TObject);
  begin
    Editieraktion.Enabled := Edit1.SelLength>0;
  end;
  ```

  Mit »Wartezustand« ist übrigens derselbe Zustand gemeint, in dem auch die *OnIdle*-Ereignisse auftreten und der in diesem Zusammenhang in Kapitel 4.7.1 untersucht wird.

- *OnHint* wird aufgerufen, bevor ein Hinweistext der Aktion angezeigt wird. Sie können darin bestimmen, ob und welcher Hinweis angezeigt werden soll (standardmäßig wird der im *Hint*-Property angegebene Hinweis angezeigt).

Standardaktionen

Das Konzept der Aktionen ist aber mit den bisher beschriebenen Möglichkeiten noch nicht ausgereizt, sondern es erlaubt auch die Definition von vordefinierten Aktionen. Das sind kurz gesagt Aktionen, zu denen Sie keine *OnExecute*-Methode mehr schreiben müssen, da sie »sich selbst ausführen« können.

Zur technischen Sicht der Standardaktionen kommen wir in Kapitel 6.6.7, wo erläutert wird, wie Sie eigene Standardaktionen definieren und sogar an andere Benutzer weitergeben können. An dieser Stelle soll es genügen, auf die mit Delphi mitgelieferten Standardaktionen hinzuweisen. Rufen Sie aus dem Kontextmenü des Aktionslisten-Editors den Befehl NEUE STANDARD-AKTION auf, um eine Liste der verfügbaren Aktionen zu erhalten und daraus zu wählen. Delphi wartet mit drei Kategorien von einfachen Standardaktionen auf: Fensteraktionen für MDI-Anwendungen (siehe Kapitel 5.7.4), die typischen Zwischenablageoperationen von Editierfeldern (diese werden immer auf das Editierfeld angewendet, das gerade den Tastaturfokus besitzt) sowie die von *TDBNavigator* gewohnten Navigationsbefehle für Datenbanken (in den Properties dieser Aktionen können Sie eine *TDataSource*-Komponente, auf die sich die Aktionen beziehen sollen, fest vorgeben).

In Delphi 6 wurde die Menge der verfügbaren Standardaktionen stark erweitert, und zwar durch die Kategorien *Format* (spezialisiert auf *TRichEdit*-Komponenten), *Hilfe*, *Datei*, *Suchen*, *Tab* (zur Steuerung eines *TTabControls*), *Liste* (Aktionen für die aktuelle Auswahl in einem Listen-Control), *Dialog* (Ausführen verschiedener Standard-Dialoge) und *Internet* (z.B. »Gehe zu URL«). Wir kommen darauf nach der Besprechung des Themas der Zielkomponenten noch einmal zurück.

Zielkomponenten

Bei den Standardaktionen ist auffällig, dass sie sich meistens auf eine andere Komponente beziehen: So benötigen beispielsweise Editieroperationen ein Editierfeld und Datenbankaktionen eine Datenquelle (*TDataSource*). Auch selbst geschriebene Aktionen sind oft von solchen *Zielkomponenten* abhängig, jedoch sind diese leicht ausfindig zu machen, da Sie sie einfach im Programmcode für das *OnExecute*-Ereignis mit Namen ansprechen können. Da die Aktionsobjekte der VCL aber in einem *beliebigen* Formular eingesetzt werden können, muss die CLX sich ihre Zielkomponenten auf eine kompliziertere Weise suchen. Der Ablauf dieser Suche ist wie folgt:

- Zuerst stellt die VCL fest, auf welches Formular sich die Aktion bezieht: falls vorhanden, wird das aktive Formular verwendet (angegeben im *ActiveForm*-Property des globalen *Screen*-Objekts), ansonsten das Hauptformular der Anwendung (*Application.MainForm*).
- Innerhalb dieses Formulars testet sie zuerst, ob die fokussierte Komponente ein passendes Ziel für die Aktion ist (*TForm.ActiveControl*).
- Ist das nicht der Fall, probiert sie, ob sich das Formular selbst als Zielkomponente eignet.
- Wenn auch dies nicht zutrifft, probiert sie alle anderen Komponenten des Formulars (*TForm.Components*) schrittweise durch, bis sich eventuell eine davon als passendes Ziel erweist.

Die Überprüfung, ob eine Komponente ein passendes Ziel für eine Aktion ist, wird vom Aktions-Objekt selbst vorgenommen. Die Standard-Editieraktionen akzeptieren beispielsweise nur fokussierte Editierfelder, so dass die Suche nach einer Zielkomponente hier entweder im zweiten Schritt oder gar nicht erfolgreich beendet werden kann. In Kapitel 6.6.7 wird gezeigt, wie dies in selbst definierten Standardaktionen in einer *HandlesTarget*-Methode geschieht.

Hinweis: Wie schon bei der Nachrichtenverarbeitung haben Sie auch beim Aufruf von Aktionen die Möglichkeit, auf globaler Ebene in die Verarbeitung einzugreifen. Noch bevor die eigentliche Aktion ausgeführt wird, erzeugt die VCL nämlich ein *OnExecute*-Ereignis der Aktionsliste und, soweit die zugehörige Bearbeitungsmethode es nicht unterbindet, danach noch ein *OnActionExecute*-Ereignis des Applikationsobjekts.

Delphi-6-Standardaktionen

Nicht alle neuen Aktionen von Delphi 6 können so vollautomatisch funktionieren wie die Standardaktionen, die sich Editierfeld, MDI-Kindfenster und Datenmengen selbst

suchen. Der Nutzen dieser Standardaktionen besteht darin, dass sie einen meistens gleich ablaufenden, von der konkreten Anwendung unabhängigen Aufgabenteil automatisch durchführen. Für die Implementierung des speziellen, anwendungsabhängigen Teils müssen Sie selbst sorgen. Damit Sie diese Implementierung bequem einbauen können, stellen die Standardaktionen neue Ereignisse bereit.

So besteht etwa die automatisch durchgeführte Arbeit einer Datei-Öffnen-Standardaktion aus dem Aufruf eines Dateidialogs. Sie sparen dadurch das manuelle Einfügen einer *TOpenDialog*-Komponente in das Formular und den Aufruf dieser Komponente mit *Execute*. Offen bleibt, was geschieht, wenn der Benutzer nun tatsächlich eine Datei gewählt hat und den Dialog mit *Ok* schließt. Sie müssen für diesen Fall zumindest das Ereignis *OnAccept* der Aktionskomponente bearbeiten. Zusätzlich bietet die Aktionskomponente noch die Ereignisse *OnCancel* (Benutzer hat *Abbruch* gedrückt) und *BeforeExecute* (zu Beginn der Aktionsausführung, bevor der Dialog aufgerufen wird). Letztlich haben Sie über diese drei Ereignisse also die gleichen Möglichkeiten wie bei einer Bearbeitung eines DATEI-ÖFFNEN-Menüpunkts ohne die Standardaktion. Sogar den von der Aktionskomponente automatisch verwendeten Standarddialog können Sie noch verändern: Alle seine Properties stehen im Objektinspektor als Unterproperties der Aktion (z.B. *TFileOpen.Dialog.FileName*) zur Verfügung.

4.6.5 Die neuen Aktionsmanager-Komponenten

Ab der Professional-Version bringt Delphi 6 im Bereich der Aktionslisten einen umfangreichen neuen Funktionsbereich mit, der sich zur Laufzeit durch seinen vollautomatischen Ablauf auszeichnet: Die vom Entwickler über Menüs und Toolbars verteilten Aktionen lassen sich nun vom Benutzer über einen ANPASSEN-Dialog umverteilen, ergänzen oder löschen, ähnlich wie in der Delphi-IDE, wenn Sie dort aus dem lokalen Menü der Symbolleisten den Punkt ANPASSEN... aufrufen (diese Änderungen zur Laufzeit werden auch automatisch gespeichert, wenn Sie ein entsprechendes *FileName*-Property setzen). Mehr noch als in der Delphi-IDE können Sie dem Benutzer sogar erlauben, das Hauptmenü zu verändern. Optional werden die Aktionen in diesem Menü auch noch nach Häufigkeit ihrer Benutzung aufgeführt. Zusammen mit den schon in früheren Delphi-Versionen vorhandenen Funktionen zum An- und Abdocken von Toolbars erhalten Sie so ohne eigene Programmierung eine Oberfläche, die dem modernsten Office-Standard entspricht (eventuell neue, dem Autor unbekannte von Microsoft neu eingeführte Spielereien in Office XP seien hierbei einmal außer Acht gelassen).

Um eine solche konfigurierbare Oberfläche in ein Formular einzubauen, benötigen Sie drei Komponenten der Seite ZUSÄTZLICH: Als zentrale, nicht-visuelle Verwaltungskomponente ein Exemplar von *TActionManager* und zur Anzeige der Aktionen im Formular eine oder mehrere der Komponenten *TActionToolBar* und *TActionMainMenuBar*. Die vierte Aktionsmanager-bezogene Komponente *TCustomizeDlg* brauchen Sie nicht

manuell in das Formular zu übernehmen, wenn Sie den Menüpunkt TOOLBARS ANPASSEN... mit Hilfe der von Borland vordefinierten Standardaktion implementieren.

Erweiterung einer Anwendung mit Aktionslisten

TActionManager dient als Container für alle Aktionen des Formulars. Das Konzept der Aktion hat sich gegenüber früheren Delphi-Versionen nicht geändert und wurde bereits in Kapitel 4.6.4 beschrieben. Wichtigste Elemente einer Aktion sind also das *Caption*-Property zur Beschriftung von Button bzw. Menüpunkt und das *OnExecute*-Ereignis für die Implementierung der Aktion. Das in Kapitel 4.6.4 beschriebene Konzept der Aktionsliste wird durch den Aktionsmanager erweitert: Da er selbst eine Aktionsliste mit Kategorieeinteilung enthält (siehe Abbildung 4.6.4), benötigen Sie bei Verwendung des Managers keine einzelnen *TActionList*-Komponenten mehr.

Zur leichteren Erweiterung bestehender Anwendung mit Aktionslisten hat Borland jedoch eine Möglichkeit vorgesehen, den Aktionsmanager mit Aktionslisten zu verknüpfen: Rufen Sie im Objektinspektor für *TActionManager.LinkedActionLists* den Property-Editor auf, der zunächst aus einer leeren Liste besteht. Dieser Liste fügen Sie nun für jede Aktionsliste, die Sie mit dem Manager verknüpfen wollen, einen neuen Eintrag hinzu. Für jeden dieser Einträge wählen Sie schließlich im Objektinspektor unter *ActionList* die zu verknüpfende Aktionsliste (vollständig ausgeschrieben ist dies dann das Property *TActionManager.LinkedActionLists[i].ActionList*).

Der Aktionsmanager-Dialog

Der größte Teil der Einstellungen des Aktionsmanagers wird in einer Variation des Editorfensters vorgenommen, das auch für die Benutzereinstellungen zur Laufzeit verwendet werden kann. Die Entwurfszeitversion dieses Fensters wird über einen Doppelklick auf die Aktionsmanager-Komponente im Formular aufgerufen; sie ist in Abbildung 4.10 gezeigt. Über den nur zur Entwurfszeit sichtbaren Hinzufügen-Schalter auf der Seite AKTIONEN erstellen Sie wie bei herkömmlichen Aktionslisten neue Aktionen oder fügen Standardaktionen ein (siehe Kapitel 4.6.4). Auch die Properties der Aktionen werden wie bei den Aktionslisten im Objektinspektor eingestellt.

Hinweise: Im Folgenden wird dieses Fenster auch als »Einstellungsdialog« oder »Aktionsmanager-Dialog« bezeichnet. Es handelt sich jedoch nicht um einen modalen Dialog, Sie können also zu anderen Fenstern wechseln, ohne das Aktionsmanager-Fenster zu schließen.

Das *Category*-Property einer neuen Aktion müssen Sie nicht immer im Objektinspektor einstellen. Es wird automatisch auf die Kategorie eingestellt, die in der Kategorieliste ausgewählt ist (Ausnahme: Wenn Sie eine Standardaktion einfügen, wird diese der für diese Aktion vordefinierten Kategorie zugeordnet).

Verknüpfung zu Aktionsleisten und Menüleisten

Wenn Sie die Aktionen Ihrer Anwendung definiert haben, geht es darum, sie auf die Aktionsleisten zu verteilen. Dafür müssen diese Aktionsleisten zunächst in das Formular eingezeichnet und so mit dem Aktionsmanager verknüpft werden, dass sie im Einstellungsdialog auf der Seite SYMBOLLEISTEN erscheinen (Abb. 4.11). Um diese Schritte durchzuführen, gibt es zwei alternative Möglichkeiten:

- Mit dem Schalter NEU... fügen Sie eine *TActionToolbar* in das Formular, die gleichzeitig mit dem Aktionsmanager verknüpft wird, also wie in der Abbildung gezeigt im Einstellungsdialog erscheint. In der dem Autor vorliegenden Delphi-Version war es nicht möglich, mit dem Schalter NEU... auch *TActionMainMenuBar*-Komponenten zu erzeugen, obwohl die drei Punkte hinter dem NEU... eine Auswahlmöglichkeit zwischen mehreren Komponenten suggerieren.

Abbildung 4.10: Die in den Aktionsmanager integrierte Aktionsliste wird auf der Dialogseite Aktionen editiert, ihre Kategorien werden in der Objekthierarchie als dem Aktionsmanager direkt untergeordnet dargestellt und die Properties der einzelnen Aktionen werden wie gehabt im Objektinspektor eingestellt.

- Alternativ können Sie die Leiste auch manuell in das Formular einfügen (*TActionMainMenuBar* konnte im Test *nur* auf diese Weise eingefügt werden). Die Verknüpfung mit dem Aktionsmanager muss dann ebenfalls manuell hergestellt werden. Dazu müssen Sie dem *ActionBars*-Objekt (in der Objekthierarchie von Abbildung 4.11 als Unterelement von *ActionManager1* zu sehen) einen neuen Eintrag hinzufügen. Dies können Sie entweder über das Property *ActionManager1.ActionBars* im Objektinspektor oder über das Kontextmenü des Eintrags *ActionBars* im Objekthierarchie-Fenster. Sobald der neue Eintrag erstellt ist, erscheint er in der Objekthierarchie als *TActionBarItem*. Sie können diesen Eintrag in der Objekthierarchie selektieren und im Objektinspektor sein *ActionBar*-Property auf die gewünschte Leiste setzen. Danach wird der *TActionBarItem* in der Objekthierarchie so dargestellt, wie in Abbildung 4.11 gezeigt: als *ActionBar -> ActionMainMenuBar1*.

Unabhängig davon, ob Sie den zuerst beschriebenen automatischen oder den beim Lesen womöglich kompliziert anmutenden zweiten Weg wählen – das Ergebnis ist immer eine Objekthierarchie wie in der Abbildung gezeigt: Alle vom Aktionsmanager verwalteten Leisten erscheinen als Untereinträge von *ActionBars*. Ein wichtiges Detail dabei ist, dass es nicht die Leisten selbst sind, die dem Aktionsmanager untergeordnet sind: Die Leisten selbst finden sich in Abbildung 4.11 unter den Namen *ActionMainMenuBar1* und *ActionToolbar1* am Ende des Objekthierarchie-Fensters (in diesem Beispiel sind sie einer *TControlBar*-Komponente untergeordnet). Die dem Aktionsmanager untergeordneten Einträge enthalten also nur eine *Referenz* auf die Aktionsleisten-Komponenten.

Unterhalb des Referenzknotens finden sich in der Objekthierarchie zwei weitere Einträge, welche die durch die verknüpfte Leiste auslösbaren Aktionen betreffen. Hier erhalten wir jedoch eine kleine Pause vom verschachtelten Objekthierarchie-Fenster, denn zur Verteilung der Aktionen auf die Leisten wird es nicht benötigt.

Verteilen der Aktionen

Um nun die Aktionsleisten mit Aktionen zu versehen, ziehen Sie die Aktionen einfach vom Einstellungsdialog (Seite AKTIONEN, siehe Abbildung 4.10) auf die Leisten im Formular. Folgende Drag&Drop-Operationen kommen in Frage:

- Aktions-Toolbars nehmen *einzelne Aktionen* auf, die durch einen Schalter ausgelöst werden. Je nach den Einstellungen im Dialog werden die Schalter mit Beschriftung dargestellt oder nur mit einem Icon (hierfür müssen natürlich die Properties *TActionManager.Images* für die Bilderliste und *TAction.ImageIndex* für den Bildindex gesetzt sein). Ziehen Sie also einzelne Aktionen vom Einstellungsdialog auf die Toolbar.

Abbildung 4.11: Auch die Verknüpfungen des Aktionsmanagers zu den Aktionsleisten sind sowohl im Dialog als auch in der Objekthierarchie (unterhalb des ActionBars-Eintrags) zu finden.

- Aktions-*Menü*leisten machen nur dann Sinn, wenn Sie eine *ganze Kategorie* von Aktionen aus dem Dialog auf die Leiste ziehen. Die Menüleiste erhält dann ein neues Menü mit dem Namen der Kategorie, unter dem die einzelnen Aktionen dieser Kategorie aufgeführt werden.
- Um eine Aktion wieder von den Leisten zu entfernen, ziehen Sie den Schalter bzw. Menüpunkt einfach mit der Maus von der Leiste an eine an der ganzen Sache »unbeteiligte« Position am Bildschirm (vom Desktop-Hintergrund wollen wir hier nicht sprechen, denn der dürfte in dieser Situation wahrscheinlich mehrlagig mit Fenstern bedeckt sein ...).

Mit den gleichen Handgriffen kann der Benutzer übrigens später zur Laufzeit die Konfiguration der Leisten ändern.

Delphi aktualisiert bei diesen Drag&Drop-Operationen automatisch den *Items*-Teilbaum der *ActionBar* im Objekthierarchie-Fenster (siehe Abbildung 4.12). Da die Properties der *Items*-Untereinträge automatisch verwaltet bzw. aus den entsprechenden Einstellungen der Aktionskomponenten übernommen werden, müssen Sie im *Items*-Unterbaum selten manuell eingreifen.

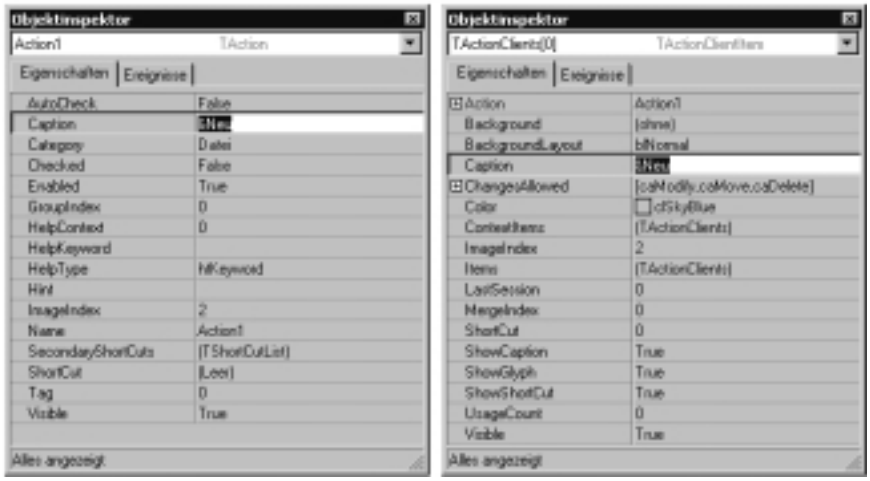


Abbildung 4.12: Die Aktionslisteneinträge (TAction, in der Objekthierarchie aus Abbildung 4.10 expandiert dargestellt) machen den festen Grundvorrat an Aktionen einer Anwendung aus und sind von den dynamischen Client-Daten (TActionClientItem, siehe Objekthierarchie in Abbildung 4.11) zu unterscheiden.

Kontextmenüs mit Aktionen

Offen geblieben ist noch der Eintrag *ContextItems* unterhalb der *ActionBar*-Einträge in der Objekthierarchie. Dieser dient der Angabe von Aktionen, die über das Kontextmenü der Leiste aufgerufen werden können. Wieder haben Sie zwei Möglichkeiten, neue Einträge unter *ContextItems* hinzuzufügen:

- Wenn in der Objekthierarchie der *ActionBar*-Eintrag selektiert ist, finden Sie das Property *ContextItems* im Objektinspektor und können darüber einen Mini-Editor aufrufen.
- Direkt von der Objekthierarchie aus können Sie Einträge über das Kontextmenü des *ActionBar*-Eintrags hinzufügen.

Beachten Sie, dass es sich bei den neuen Einträgen wieder nur um eine Referenz handelt. Die Aktionen selbst verbleiben in ihren Aktionslisten bzw. in den Kategorien des Aktionsmanagers. Die neuen Einträge unter *ContextItems* haben die Klasse *TActionClientItem* und werden über ihr Property *Action* mit der gewünschten Aktion verknüpft.

Konfiguration zur Laufzeit

Neben den Aktionsleisten finden Sie in der Komponentenpalette auch die Komponente *TCustomizeDlg*, die den Aktionsmanager-Dialog zur Laufzeit zugänglich macht. Jedoch brauchen Sie diese Komponente nicht direkt zu verwenden, denn was liegt bei Verwendung des Aktionsmanagers näher, als den Aufruf des Einstellungsdialogs selbst als Aktion zu definieren? Entsprechend wird Delphi bereits mit einer Standardaktion ausgeliefert, die genau diesen Dialog aufruft. Sie finden diese Aktion, wenn Sie in der Aktionsliste des Aktionsmanagers das Kontextmenü aufrufen und NEUE STANDARDAKTION... wählen in der Kategorie *Tools* und unter dem Namen *TCustomizeActionBars*.

Wenn Sie die neue Standardaktion in Ihre Aktionsliste übernommen haben, müssen Sie sie nur noch auf eine der Aktionsleisten schieben. Wenn die Aktion dann zur Laufzeit aufgerufen wird, wird der Einstellungsdialog automatisch geöffnet.

Natürlich macht die Änderung der Einstellungen zur Laufzeit nur dann Sinn, wenn die Änderungen beim Programmende auch gespeichert und beim nächsten Start automatisch wiederhergestellt werden. Borland hat das nicht vergessen und den Aktionsmanager daher mit dem Property *FileName* versehen. Wenn Sie hier einen Dateinamen angeben, wird sowohl das Speichern als auch das Wiederherstellen der Konfiguration automatisch übernommen.

Hintergrund: Sie können sich die Speicherung der Konfiguration grob so vorstellen, dass all das gespeichert wird, was im Objekthierarchie-Fenster als *Items*-Teilbäume unter den *ActionBar*-Einträgen zu sehen ist. Beim Neustart des Programms haben die aus der Datei geladenen Einstellungen Vorrang vor den Einstellungen, die Sie zur Entwurfszeit vorgenommen haben. Wenn eine solche Konfigurationsdatei bereits erstellt wurde, hat dies zur Folge, dass Änderungen, die Sie zur Entwurfszeit vornehmen, zur Laufzeit nicht sichtbar werden. Um sie sichtbar zu machen, löschen Sie entweder die Konfigurationsdatei oder setzen das Property *TActionManager.FileName* zurück.

Zusammenfassung der Aktionsmanager-Einstellungen

Abschließend seien hier die entscheidenden der schon besprochenen Einstellungen zusammengefasst. Die meisten Einstellungen tauchen im Objektinspektor als Properties von *TActionManager* auf (auch wenn sie nicht immer über den Objektinspektor editiert werden müssen):

- *ActionBars*: Liste der Verknüpfungen zu Aktionsleisten (*TActionMainMenubar* und *TActionToolbar*); kann auch über das Objekthierarchie-Fenster editiert werden; wenn Sie eine Toolbar über den Einstellungsdialog einfügen, wird die Verknüpfung automatisch aufgebaut.

- *FileName*: Datei zur automatischen Speicherung der Einstellungen (siehe oben).
- *Images*: Wie schon bei den Aktionslisten in Kapitel 4.6.4 wird über dieses Property eine Verbindung zu der *TImageList*-Komponente aufgebaut, auf die sich die Bilderindizes aller Aktionen beziehen.
- *LinkedActionLists*: Liste der in den Aktionsmanager eingebundenen *TActionList*-Komponenten (zur Erweiterung bestehender Anwendungen nützlich; auch diese Liste kann über das Objekthierarchie-Fenster erweitert werden).

Der grundlegenste Inhalt des Aktionsmanagers, die Liste der Aktionen, ist *nicht* über den Objektinspektor zugänglich, sondern nur über den Aktionsmanager-Dialog.

Ein Beispiel

Als Beispiel soll hier eine spezielle Version des TreeDesigners aus Kapitel 5 dienen, bei der das Hauptmenü durch eine *TActionMenuBar* ersetzt wurde und die sowieso schon vorhandenen Werkzeugleisten durch *TActionToolBars* ergänzt wurden. Eine komplette Umstellung des TreeDesigners auf die neuen Aktionsmanager-Komponenten wurde aus vielerlei Gründen nicht durchgeführt:

- In der MDI-Version des TreeDesigners werden die meisten Funktionen vom Dokumentformular bereitgestellt. Ein Aktionsmanager auf Hauptfensterebene wäre zwar möglich, würde aber für die meisten Aktionen neuen Code erfordern, in dem das gerade aktuelle Dokumentformular aufgerufen wird. Daher wurde das Aktionsmanager-Konzept lediglich an der SDI-Version des TreeDesigners getestet.
- Die SDI-Version des TreeDesigners soll kompatibel mit der Personal-Edition von Delphi und mit Kylix sein – der Aktionsmanager ist diesen beiden jedoch unbekannt. Daher finden Sie auf der CD eine separate SDI-Aktionsmanager-Version des TreeDesigners unter dem Projektnamen *TreeDesignerSDI_AM*.
- Die bestehenden Toolbars des TreeDesigners enthalten auch Eingabeelemente wie Editierfelder und einen TrackBar. Diese sind in *TActionToolBar* nicht vorgesehen, daher werden die bestehenden Toolbars weiter verwendet und durch neue Exemplare von *TActionToolBar* ergänzt.

Abbildung 4.13 zeigt das Hauptformular der speziellen TreeDesigner-Version zur Entwurfszeit. Als Beispiel für die Verwendung des Aktionsmanagers seien nun die Schritte kurz dargestellt, die im TreeDesigner durchgeführt wurden. Da die Details zum Umgang mit dem Aktionsmanager bereits oben beschrieben wurden, dient dies als eine weitere Zusammenfassung, diesmal aus praktischer Hinsicht:

- 1. Einfügen einer *TActionManager*-Komponente in das Formular und Verbinden mit der schon bestehenden TreeDesigner-Aktionsliste (diese enthält lediglich die Aktionen des Menüs ANSICHT und wird über das Property *TActionManager.LinkedActionLists* mit dem Aktionsmanager verbunden), Verbinden mit der schon bestehenden Bilderliste (*TActionManager.Images*).

- 2. Definition der Aktionen. Hierzu wurden die Einträge des normalen TreeDesigner-Hauptmenüs in Aktionen »übersetzt«. Dazu mussten für jeden Menüpunkt eine neue Aktionskomponente definiert und deren Properties *Caption*, *ImageIndex* auf die entsprechenden Werte des Menüpunkts gesetzt werden. Das *OnExecute*-Ereignis der Aktionen wurde mit der *OnClick*-Methode des entsprechenden Menüpunkts verknüpft. Als Kategorienamen wurden die Namen der Hauptmenüpunkte verwendet (DATEI, BEARBEITEN etc.). Außerdem wurde für die Konfiguration der Leisten zur Laufzeit mit NEUE STANDARDAKTION... eine *Customize-Toolbar*-Standardaktion hinzugefügt und zur Demonstration noch eine der in Kapitel 6.6.7 selbst definierten Farbpaletten-Standardaktionen.

- 3. Es wurden, wie oben unter *Verknüpfung zu Aktionsleisten und Menüleisten* beschrieben, zwei *TActionToolbars* automatisch und eine *TActionMainMenuBar* manuell mit dem Aktionsmanager verknüpft. Um die *MainMenuBar* zu füllen, wurden die einzelnen Kategorien mit der Maus auf die Komponente gezogen. Die erste *ActionToolbar* wurde mit einer zufälligen Auswahl an Aktionen gefüllt, die zweite wurde leergelassen und unsichtbar gemacht (sie soll dem Benutzer zur Laufzeit als optional zuschaltbarer Speicher für seine Lieblingsaktionen dienen, ähnlich der *Individuell*-Toolbar der Delphi-IDE).

- 4. Um die aus der externen Aktionsliste stammenden Aktionen in die neue Menüleiste aufzunehmen, wird zunächst in der aufklappbaren Liste des Aktionsmanagers (siehe den Eintrag *(Alle Aktionen)* in Abbildung 4.10) die externe Aktionsliste (hier *Ansichtsaktionen*) gewählt, so dass die von ihr definierten Aktionskategorien in der Liste erscheinen. Von dort können diese dann wie jede im Aktionsmanager definierte Kategorie auf die *ActionMainMenuBar* gezogen werden. Im Beispiel wurde noch die Menübeschriftung geändert, die automatisch auf den Kategorienamen gesetzt wird. Im Beispiel soll statt dem Kategorienamen ANSICHTSAKTIONEN der Menütext ANSICHT angezeigt werden. Natürlich würde es genügen, den Kategorienamen auf *Ansicht* zu ändern, aber Sie haben auch noch eine andere Möglichkeit, die Menütexte anzupassen: Wählen Sie den für die Kategorie erstellten *TClientActionItem*-Eintrag in der Objekthierarchie (unter dem *Items*-Knoten) und setzen Sie sein *Caption* auf den gewünschten Menütext.

- 5. Wenn Sie über den Aktionsmanager-Dialog neue *ActionToolbars* erzeugen, werden diese per Voreinstellung direkt dem Formular als Kindkomponenten untergeordnet. Um sie in die *ControlBar* des TreeDesigner-Dokumentfensters zu ver-

schieben, wurde die glorreiche Drag&Drop-Funktion des Objekthierarchie-Fensters verwendet. Für die *ControlBar* wurden das Property *AutoSize* eingeschaltet und *RowSize* auf einen passenden Wert eingestellt. Für die einzelnen Aktionsleisten wurden vor allem das Property *MinWidth* angepasst, damit die Leisten beim Verschieben innerhalb der ControlBar nicht so klein werden, dass ihr Inhalt nicht mehr genutzt werden kann. Rein optische Wirkung hatte schließlich die Modifikation des Properties *EdgeBorders* (ebenfalls jeweils für jede Aktionsleiste).

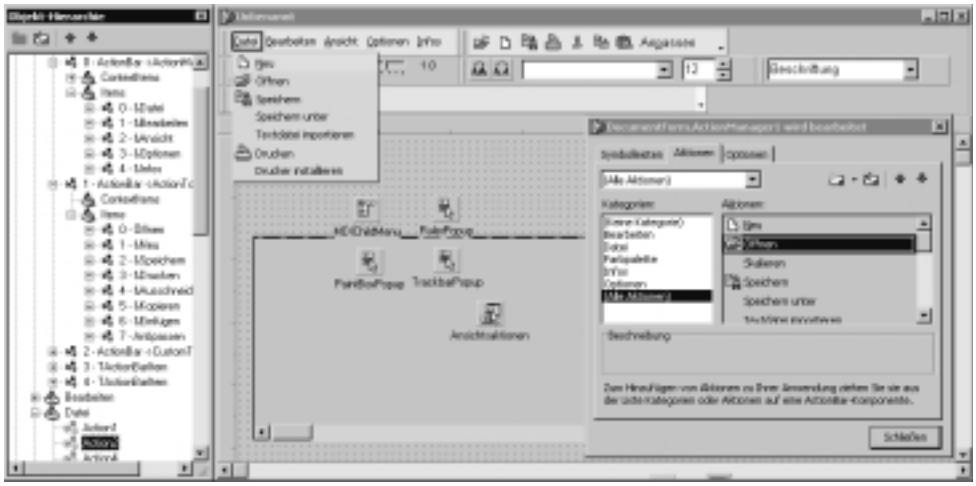


Abbildung 4.13: Das Dokumentformular des SDI-TreeDesigners mit drei neuen Aktionsleisten. Um hier zur Entwurfszeit alle relevanten Editorfenster, das entworfene Formular und das Hauptfenster der Delphi-IDE sehen zu können, ist ein großer Bildschirm erforderlich.

4.7 Threads

Threads sind Programmabschnitte, die innerhalb einer Anwendung parallel ablaufen können. Dieses Kapitel zeigt anhand eines interaktiven Beispielprogramms die Unterschiede zwischen zwei verschiedenen Multitasking-Typen, wichtige Eigenschaften der Threads und wie diese synchronisiert werden, so dass beispielsweise mehrere Threads auf dieselben Variablen zugreifen können. Weitere Beispielprogramme gibt es zur Synchronisation von Threads und zur Verwendung einer dynamischen Anzahl von Threads.

4.7.1 Multitasking-Typen

In einem Betriebssystem, das lediglich kooperatives Multitasking beherrscht, dürfen Anwendungen nicht ununterbrochen arbeiten, sondern müssen hin und wieder die

Kontrolle an das Betriebssystem zurückgeben, so dass dieses weitere Eingaben des Benutzers an die anderen Anwendungen weiterleiten und dort bearbeiten lassen kann.

Beim preemptiven Multitasking kann das Betriebssystem die gerade laufende Anwendung jederzeit unterbrechen, was dem Anwendungsprogrammierer einen erheblichen Vorteil bringt, da er sich nicht mehr um die Aufteilung der Rechenzeit kümmern muss. Auch der Anwender kann großen Nutzen aus dieser Form des Multitasking ziehen, da seine Arbeitsoberfläche durch einzelne fehlgeleitete Programme nicht mehr ohne weiteres blockiert werden kann.

Ziel des ersten Beispielprogramms *ThreadDemo1* (Abbildung 4.14) ist ein Vergleich zwischen preemptivem Multitasking durch Threads und kooperativem Multitasking durch die Nutzung von *TApplication.OnIdle* oder *TApplication.ProcessMessages*.

Kooperatives Multitasking

Wie oft eine Anwendung beim kooperativen Multitasking die Kontrolle an das Betriebssystem zurückgeben muss, damit der Benutzer vernünftig weiterarbeiten kann, können Sie z.B. beim Eingeben von Texten daran ablesen, wie schnell Sie erwarten, dass die Zeichen auf dem Bildschirm erscheinen, gemessen vom Zeitpunkt, an dem Sie die Taste gedrückt haben. Eine Viertelsekunde Reaktionszeit ist hier für einen Computer schon ziemlich träge, setzt aber voraus, dass eine eventuell im Hintergrund arbeitende Anwendung ihre Berechnung mindestens vier Mal pro Sekunde unterbricht.

Im einfachsten Fall wird das kooperative Multitasking dadurch sichergestellt, dass eine Anwendung jede Eingabe des Benutzers in Sekundenbruchteilen vollständig bearbeiten kann. Handelt es sich z.B. um eine Delphi-Anwendung, bei der die Bearbeitung innerhalb einer Ereignisbearbeitungsmethode stattfindet, gelangt die Kontrolle automatisch an das Betriebssystem zurück.

Wenn es einmal etwas länger dauert, verwandeln viele Anwendungen den Mauszeiger für kurze Zeit in eine Sanduhr, die dem Benutzer signalisiert, dass er sich etwas gedulden muss. Hat die Anwendung die Aufgabe erledigt, erhält der Anwender seinen gewohnten Mauszeiger zurück und kann sich neue Herausforderungen für seine Software ausdenken. Das kooperative Multitasking ist in diesem Fall darauf beschränkt, dass Windows so lange beliebig zwischen den Anwendungen wechseln kann, wie keine neuen Ereignisse zu bearbeiten sind.

Damit eine Anwendung auch bei kooperativem Multitasking länger dauernde Aufgaben erledigen kann, ohne das System zu blockieren, bieten sich mit der VCL zwei Möglichkeiten an:

- stückweises Ausführen der Aufgabe bei bestimmten regelmäßigen Ereignissen, z.B. bei *OnIdle*- oder *OnTimer*-Ereignissen oder

- komplettes Ausführen der Aufgabe mit freiwilliger Unterbrechung und Rückgabe der Kontrolle an Windows durch Aufruf von *Application.ProcessMessages*.

TApplication.OnIdle

Eine Anwendung kann die »Zeit zwischen den Ereignissen« gezielt nutzen, um kleinere Arbeiten zu verrichten. Die VCL definiert für eine solche ereignislose Phase das Ereignis *TApplication.OnIdle*, das Sie nicht über den Objektinspektor, sondern nur im Programmcode mit einer Methode verknüpfen können. Das Ereignis *OnIdle* ist nicht auf eine Windows-Nachricht zurückzuführen, sondern wird alleine von der VCL erzeugt, wenn diese feststellt, dass die Nachrichtenwarteschlange der Anwendung leer ist.

Da auch eine Methode für *OnIdle* nur wenig Zeit in Anspruch nehmen sollte (damit neue Eingaben des Benutzers sofort bearbeitet werden können), wird *OnIdle* meistens dazu verwendet, zeitlich unkritische Routineaufgaben zu erfüllen, wie z.B. die Anzeige einer Uhrzeit in der Statuszeile zu aktualisieren. Eine *Idle*-Methode kann jedoch auch länger dauernde Aufgaben durchführen, wenn sie bei jedem Durchlauf, also bei jedem *OnIdle*-Event, nur ein kleines Stück weiter rechnet und sich zwischen den Ereignissen den bisherigen Fortschritt merkt.

TTimer als Alternative

Eine Alternative zum Abfangen des *OnIdle*-Ereignisses ist in vielen Fällen die Verwendung einer *TTimer*-Komponente, die *OnTimer*-Ereignisse erzeugt. Diese *OnTimer*-Ereignisse sind erheblich regelmäßiger als *OnIdle*-Ereignisse. Ein Timer hat außerdem den Vorteil, dass er automatisch ausgelöst wird und die Anwendung nicht ständig abfragen muss, ob Nachrichten zur Bearbeitung anliegen (diese Abfrage wird von der VCL durchgeführt, bevor diese ein *OnIdle*-Ereignis erzeugt).

Zur Verwendung eines Timers nehmen Sie sich eine *TTimer*-Komponente von der Palettenseite *System*, stellen ihr *Intervall*-Property auf die Zeit in Millisekunden, die zwischen zwei *OnTimer*-Ereignissen vergehen soll, und schreiben eine Methode für das *OnTimer*-Ereignis. Um den Timer zeitweise zu deaktivieren, verwenden Sie das Property *Enabled*. Ein Beispiel für die Verwendung von *TTimer* geben die Beispielprogramme aus dem ersten Kapitel und das Programm *SyncDemo* aus Kapitel 4.7.3.

Das Multitasking-Demoprogramm

Wenn Sie das Beispielprogramm *ThreadDemo1* aus Abbildung 4.14 ablaufen lassen, erhalten Sie einen Eindruck davon, wie regelmäßig *OnIdle*-Ereignisse auftreten und wie im Vergleich dazu ein Thread mit der Berechnung vorankommen würde.

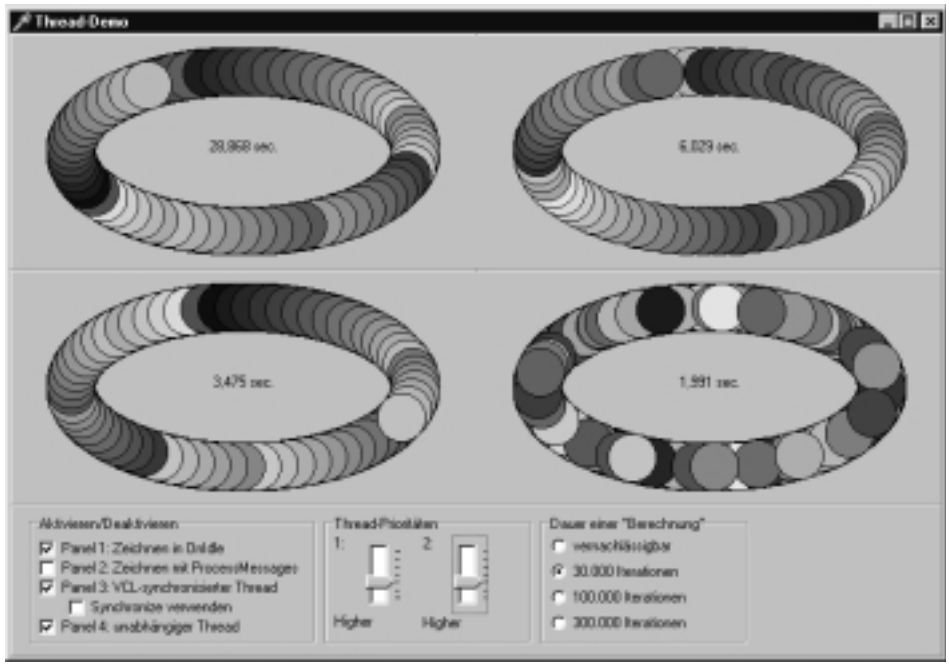


Abbildung 4.14: Die vier Bereiche dieses Fensters ermöglichen einen Vergleich zwischen kooperativem und preemptivem Multitasking

Das Fenster des Programms wird durch *TPanel*-Komponenten in vier Bereiche geteilt, die jeweils durch eine *TPaintBox*-Komponente abgedeckt werden. Der linke obere Bereich ist für das *OnIdle*-Ereignis reserviert: Die *OnIdle*-Methode zeichnet bei jedem Aufruf einen farbig gefüllten Kreis, wobei dieser – über mehrere Ereignisse betrachtet – auf einer Ellipsenbahn über die Fläche des Panels bewegt wird. Die Zahl in der Mitte dieser Ellipsenbahn gibt die für die letzte »Runde« ungefähr benötigte Zeit an.

Das Panel rechts oben wird auf eine später beschriebene Weise gefüllt, die beiden unteren Panels werden durch Threads gezeichnet.

Unwichtige Arbeit wird in einer Hilfsunit versteckt

Das Beispielprogramm wurde so entworfen, dass der Code zum Thread-Management streng von allem restlichen Code getrennt ist. Der »restliche Code« ist dafür zuständig,

- eine Berechnung für die verschiedenen Threads (bzw. für *OnIdle*) zu simulieren,
- den Fortschritt der Berechnung durch die Ausgabe der Grafik am Bildschirm sichtbar zu machen

- und die Geschwindigkeit des Fortschritts zu messen sowie am Bildschirm auszugeben.

Damit wir uns hier ganz auf das Thread-Management konzentrieren können, wurde alles, was mit den obigen Aufgaben zusammenhängt, in die Unit *GlobalProcs* verlagert. Zum Verständnis des Programms brauchen Sie nur das Interface dieser Unit zu kennen:

```
unit GlobalProcs;
interface
uses SysUtils, ExtCtrls, Graphics;
type
  TProgress = object
    // keine öffentlichen Variablen
  end;

procedure InitProgress(var Progress: TProgress);
procedure SimulateWork(var Progress: TProgress);
procedure ShowProgress(PaintBox: TPaintBox; const Progress: TProgress);

var
// zwei Optionen, die zur Laufzeit angepasst werden können
// (s. Abbildung 4.14, "Dauer einer Berechnung"
//  und "Synchronize verwenden"):
  Iterations: integer;
  UseSynchronize: Boolean=False;

implementation
```

Für jeden seiner vier Fensterbereiche wird das Beispielprogramm ein Objekt des oben deklarierten Typs *TProgress* erstellen, das vom jeweiligen Thread bzw. von *OnIdle* verwaltet wird. Um die Berechnung zu simulieren und die Grafik auszugeben, rufen *OnIdle* und die Threads einfach nur *SimulateWork* und *ShowProgress* auf und übergeben ihnen ihr *Progress*-Objekt, in dem alle intern wichtigen Daten enthalten sind.

Hinweis: Für *TProgress* wurde das Schlüsselwort *object* verwendet, damit *TProgress* von außen so leicht zu verwenden ist wie ein *record* (es sind keine Konstruktor- oder Destruktoraufrufe erforderlich), aber gleichzeitig den Zugriffsschutz einer *class* genießt (alle Elemente von *TProgress* sind als *private* deklariert und werden nur innerhalb der Unit *GlobalProcs* angesprochen).

Implementation von OnIdle *R146*

In der Methode für das *OnCreate*-Event des Formulars wird das *OnIdle*-Ereignis des globalen Anwendungsobjekts der VCL wie folgt mit einer Methode verknüpft:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // Variablen initialisieren, die in FormIdle benutzt werden:
  InitProgress(IdleProgress);
  Application.OnIdle := FormIdle; // Ereignis und Methode verbinden
```

Um den bisherigen »Fortschritt« zu speichern, ist in der Formularklasse die Formularvariable *IdleProgress* deklariert. Sofern die *OnIdle*-Bearbeitung nicht über den Markierungsschalter *Panel1Active* abgestellt wurde, wird *IdleProgress* wie angekündigt einfach nur an die Prozeduren der Unit *GlobalProcs* weitergegeben:

```
procedure TForm1.FormIdle(Sender: TObject; var Done: Boolean);
begin
  // nur, wenn der Markierungsschalter nicht ausgeschaltet ist:
  if Panel1Active.Checked then begin
    GlobalProcs.SimulateWork(IdleProgress);
    GlobalProcs.ShowProgress(PaintBox1, IdleProgress);
    Done := False;
  end;
  // else Done := True; :: Done ist per Voreinstellung True.
end;
```

OnIdle in der Praxis

Wenn Ihr Rechner durch andere Prozesse nicht ausgelastet ist, wird die *OnIdle*-Methode sehr schnell mit dem Zeichnen der Kreise vorankommen, und zwar zunächst einmal erheblich schneller als der im linken unteren Bereich arbeitende Thread. Allerdings kann es bei der Bewegung des Kreises sehr schnell zu Stockungen und Pausen kommen, nämlich immer dann, wenn es in der Nachrichtenschlange der Anwendung zu einem kleinen Stau kommt, die Bearbeitung einer einzelnen Nachricht etwas länger dauert oder das System aus anderem Grund für eine Weile vollständig ausgelastet ist. Solange beispielsweise das Systemmenü des Fensters aufgeklappt ist, tut sich im linken oberen Fensterbereich überhaupt nichts, während der Thread seine Zeichnung ungehindert fortsetzt.

Das Programm zeigt außerdem, was passiert, wenn ein Durchlauf der *OnIdle*-Methode zu lange dauert: Erhöhen Sie z.B. die *»Dauer einer Berechnung«* auf das Maximum, so wird die Reaktionszeit des Systems sehr träge. Wenn beispielsweise der Mauszeiger den Fensterrand berührt, dauert es meistens eine Weile, bis er wie üblich sein Aussehen ändert. (Unter 32-Bit-Windows wird die Reaktionszeit innerhalb der *anderen* Anwendungen jedoch nicht beeinflusst, denn *zwischen* den 32-Bit-Anwendungen führt sogar Windows 95 preemptives Multitasking durch.)

Wenn Sie die *OnIdle*-Bearbeitung über den entsprechenden Schalter des Formulars abschalten und nur noch die beiden Threads laufen lassen, ist dagegen keine derartige Trägheit festzustellen, obwohl die Threads genau dieselbe »Berechnung« durchführen wie die *OnIdle*-Methode.

Hinweis: Bei der Veränderung der *»Dauer einer Berechnung«* passt das Beispielprogramm den Wert der globalen Variablen *Iterations* in der Unit *GlobalProcs* an. Dieser wirkt sich lediglich auf die Zahl der Rechenoperationen aus, die die Methode *SimulateWork* für *OnIdle* und alle vier Fensterbereiche gleichermaßen durchführt.

TApplication.ProcessMessages

Während die Methode für *OnIdle* davon abhängig ist, dass nicht zu viele andere Ereignisse anfallen, verhält es sich bei der zweiten Möglichkeit des kooperativen Multitasking umgekehrt: Mit einem Aufruf von *TApplication.ProcessMessages* haben Sie es in der Hand, wann Ereignisse bearbeitet werden dürfen; wenn Sie dies zu selten zulassen, blockieren Sie Ihre eigene Anwendung – die anderen Anwendungen sind aufgrund des preemptiven Multitaskings von Windows hiervon unabhängig.

Ein Beispiel für einen sinnvollen Einsatz von *ProcessMessages* finden Sie im Beispielprogramm *FileDB* in Kapitel 7.3.7. In *ThreadDemo1* wird *ProcessMessages* innerhalb der Methode *Panel2Proc* aufgerufen. *Panel2Proc* tut dasselbe wie die eben gezeigte Methode *FormIdle*, kehrt aber nicht nach jedem einzelnen Schritt zum Aufrufer zurück, sondern ruft nach jedem Arbeitsschritt *ProcessMessages* auf:

```
procedure TForm1.Panel2Proc;
var
  Progress: Integer;
begin
  InitProgress(Progress);
  // Wiederholen, solange der Schalter Panel2Active eingeschaltet ist:
  while Panel2Active.Checked do begin
    Application.ProcessMessages;
    GlobalProcs.SimulateWork(Progress);
    GlobalProcs.ShowProgress(PaintBox2, Progress);
  end;
end;

// Aufruf von Panel2Proc beim Anklicken des Markierungsschalters:
procedure TForm1.Panel2ActiveClick(Sender: TObject);
begin
  if Panel2Active.Checked then
    Panel2Proc;
end;
```

Interessant ist hierbei, dass *Panel2Proc* erst dann beendet wird, wenn Sie den Markierungsschalter ausschalten und damit *Panel2Active.Checked* wieder auf *False* setzen. Die Ereignisbearbeitungsmethode *Panel2ActiveClick* kehrt also zunächst einmal überhaupt nicht zu ihrem Aufrufer zurück, und trotzdem können Sie dem Programm andere Eingaben senden, die es bereitwillig verarbeitet. Dies liegt am Aufruf von *Application.ProcessMessages*, zu dessen Laufzeit alle zwischenzeitlich angefallenen Nachrichten bearbeitet werden, also quasi geschachtelt in die Bearbeitung von *Panel2ActiveClick*.

Ein Aspekt ist hier jedoch wichtig: Das Fenster kann auf normalem Wege erst dann geschlossen werden, wenn alle Nachrichten abgearbeitet sind. Daher muss die Prozedur *Panel2Proc* gestoppt werden, wenn der Benutzer versucht, das Fenster zu schließen. Im Falle eines *OnClose*-Ereignisses wird daher folgende Aktion durchgeführt:

```
procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  Panel2Active.Checked := False; // Simuliert das Ausschalten
    // des Schalters, so dass eine eventuell aktive Panel2Proc
    // und somit auch eine eventuell noch nicht abgeschlossene
    // Bearbeitung des OnClick-Ereignisses für den Schalter
    // Panel2Active beendet wird.
end;
```

ProcessMessages in der Praxis

Die Grafik im Fensterviertel rechts oben wird ähnlich schnell und unkontinuierlich erneuert wie die Grafik der *OnIdle*-Methode. Auch im Falle länger dauernder Berechnungen verhält sich *Panel2Proc* wie *FormIdle*. Vorteil von *Panel2Proc* ist jedoch, dass sie sich den Fortschritt in den lokalen Variablen *Progress* und *Color* merken kann und erst dann verlassen werden muss, wenn die simulierte Berechnung beendet ist.

Eine interessante Beobachtung ist darüber hinaus, dass *FormIdle* kein einziges Mal mehr aufgerufen wird, sobald Sie *Panel2Proc* gestartet haben, denn *Application.ProcessMessages* arbeitet nur die inzwischen angefallenen Nachrichten ab und kehrt dann sofort zurück zu *Panel2Proc*. Für ein *OnIdle*-Event bleibt so überhaupt keine Zeit mehr.

Unter Windows 95 und NT sind weder *ProcessMessages* noch *OnIdle* notwendig, um andere Anwendungen zum Zuge kommen zu lassen, jedoch können Sie auf beide Arten erreichen, dass Ihre eigene Anwendung gleichzeitig rechnen und weitere Nachrichten bearbeiten kann, ohne dass Sie dazu einen neuen Thread starten müssen.

Preemptives Multitasking

Mussten wir uns bisher darum kümmern, eine längere Berechnung ständig zu unterbrechen, sei es durch *Application.ProcessMessages* oder durch ein möglichst schnelles Verlassen der *OnIdle*-Bearbeitung, nimmt uns beim preemptiven Multitasking das Betriebssystem diese Aufgabe vollständig ab. Es kann das Programm zwar genau genommen nur bei einem Interrupt unterbrechen, aber schon der alte PC-Timer-Interrupt wird ca. 18-mal pro Sekunde ausgelöst (theoretisch könnte das Betriebssystem eine Anwendung natürlich auch nach jeder einzelnen Anweisung unterbrechen, indem es wie ein Debugger Haltepunkte setzt).

Für Sie als Entwickler bedeutet das, dass Sie sich weitaus weniger Gedanken über die Beteiligung anderer Programme an der Rechenzeit machen müssen. Preemptives Mul-

titasking bedeutet also nicht nur mehr Komfort für den Anwender, sondern auch für den Softwareentwickler. Die Tatsache, dass das Betriebssystem die gesamte Verantwortung über die Rechenzeit übernimmt, braucht jedoch keinen Programmierer in die Verantwortungslosigkeit zu treiben, denn auch Threads können kompliziert werden, wenn sie zu mehreren innerhalb eines Prozesses (einer Anwendung) auftreten, auf gemeinsame Daten zugreifen und synchronisiert werden wollen. Hier stellt sich dann eventuell eine Vielzahl an neuen Herausforderungen, in die man die durch das preemptive Multitasking eingesparte Entwicklungszeit wieder neu investieren kann.

Hinweis: Unter Windows 95/98 ist das preemptive Multitasking auf 32-Bit-Anwendungen eingeschränkt. Eine alte 16-Bit-Anwendung wird also bei ihrer Nachrichtenbearbeitung nicht preemptiv unterbrochen. Dies fällt erst dann auf, wenn sie einmal längere Zeit braucht, um eine Nachricht zu bearbeiten, und zwischendurch nicht z.B. *ProcessMessages* aufruft, denn dann kann auch Windows 95/98 nicht mehr zu anderen Anwendungen wechseln.

4.7.2 Threads in der VCL

Wenn Sie Ihre Anwendung in mehrere Threads aufteilen, kann sie gleichzeitig Benutzereingaben verarbeiten und längere Berechnungen durchführen. Nach dem Start der Anwendung läuft diese in einem *primären Thread* (oder Haupt-Thread), und nur dieser primäre Thread kann die Nachrichten des Anwenders bearbeiten. Um eine längere Berechnung durchführen zu können, ohne dass die Anwendung blockiert wird, kann der primäre Thread weitere Threads starten. Ein solcher zusätzlicher Thread genügt in vielen Fällen schon.

Die Klasse TThread

Nach der langen Vorrede des letzten Abschnitts kommen wir nun zur Sache und erstellen ein *TThread*-Objekt. Dafür machen wir zunächst vom Angebot der Delphi-IDE Gebrauch, eine Gerüst-Unit für einen neuen Thread anzulegen. Sie können Thread-Klassen natürlich auch selbst anlegen und müssen diese nicht jeweils in einer eigenen Unit unterbringen.

Wenn Sie also mit DATEI | NEU die Objektablage aufrufen, das mit »Thread-Objekt« beschriftete Symbol wählen und in der folgenden Dialogbox einen Namen für die neue Thread-Klasse eingeben, erhalten Sie eine neue Seite im Editor mit einer neuen Unit, die wiederum eine neue Klasse enthält. Für das Beispielprogramm *ThreadDemo1* ist der Klassenname *TDemoThread*; das Gerüst sieht wie folgt aus:

```
unit Unit2;
interface
```

```
uses
  Classes;

type
  TDemoThread = class(TThread)
  private
    { Private-Deklarationen }
  protected
    procedure Execute; override;
  end;

implementation

... Kommentar zu "SYNCHRONIZE" wurde hier gekürzt ...

{ TDemoThread }

procedure TDemoThread.Execute;
begin
  { Place thread code here }
end;

end.
```

Die Methode Execute

Im Implementationsteil der neuen Unit befindet sich ein hier nicht abgedruckter Hinweis zum Aufruf der VCL aus diesem Thread (hierzu kommen wir später) und der bis auf einen weiteren Hinweis leere Rumpf der Methode *Execute*.

Diese Methode ist auch bereits das Wichtigste, was ein Thread braucht; sie enthält sozusagen das »Hauptprogramm« des Threads: Der Thread beginnt mit der ersten Anweisung in *Execute* parallel zum bisher laufenden Programm zu arbeiten und wird beendet, sobald die Ausführung dieser Methode abgeschlossen ist.

Um nun wie im letzten Abschnitt die kreisende Scheibe zu zeichnen, kann die *Execute*-Methode wie folgt implementiert werden:

```
procedure TDemoThread.Execute;
begin
  while not Terminated do begin
    GlobalProcs.SimulateWork(Progress);
    Synchronize(Update);
  end;
end;
```

Hierbei fallen zwei Dinge auf: Viele Threads, die ein bestimmtes Ziel zu erreichen haben, enden »freiwillig«, wenn die *Execute*-Methode abgelaufen ist (z.B. die Threads aus Kapitel 4.7.4). Der oben gezeigte Thread kann theoretisch unendlich lange laufen, bis er von außen durch den Aufruf von *Terminate* beendet wird. In diesem Fall wird das *Terminated*-Property gesetzt, das oben nach jedem Rechenschritt überprüft wird.

Außerdem wurde der bisher bekannte *ShowProgress*-Aufruf durch den Aufruf von *Synchronize(Update)* ersetzt, zu dem wir gleich kommen werden.

Hinweis: Tatsächlich besteht der durch *TThread* definierte Thread nicht aus der *Execute*-Methode, sondern aus der VCL-Funktion *ThreadProc*, die aber sofort *Execute* aufruft. Nachdem diese abgearbeitet ist, erzeugt *ThreadProc* das Ereignis *OnTerminate* des Threads und gibt das *TThread*-Objekt frei, falls Sie dessen Property *FreeOnTerminate* auf *True* gesetzt haben.

Synchronize

Der Aufruf von *Synchronize* ist erforderlich, da die bereits erwähnte Methode *ShowProgress* intern VCL-Objekte benutzt, um Grafik in der PaintBox auszugeben. Die VCL kann jedoch immer nur in einem Thread gleichzeitig aktiv sein: Wenn beispielsweise der Thread *TDemoThread* versuchen würde, in die PaintBox zu zeichnen, während gleichzeitig der Haupt-Thread diese PaintBox gerade selbst verwendet, könnte es zu unvorhersehbaren Zwischenfällen kommen.

Um derartige Konflikte einfach vermeiden zu können, definiert *TThread* die Methode *Synchronize*. Diese wartet ab, bis sich die VCL eventueller anderer Aufgaben entledigt hat, und führt erst dann die Methode aus, die Sie ihr als Parameter übergeben. Sie brauchen nun, um die VCL aus einem Thread heraus aufzurufen, nur alle zusammengehörenden VCL-Aufrufe in eine parameterlose Methode zu packen und diese an *Synchronize* weiterzugeben. Die oben an *Synchronize* übergebene Methode *Update* enthält nur den schon bekannten *ShowProgress*-Aufruf:

```
procedure TDemoThread.Update;
begin
  GlobalProcs.ShowProgress(PaintBox, Progress, Color);
end;
```

Der Schalter *Synchronize verwenden* in Abbildung 4.14 deutet bereits an, dass Sie das hier beschriebene Verhalten des Programms zur Laufzeit ausschalten können. Dies hängt mit einer weiteren Eigenschaft der VCL zusammen, die im Abschnitt *Lock und Unlock für die Grafikausgabe* beschrieben wird.

Hinweis zu den VCL-Interna: Das Warten der Methode *Synchronize* sieht intern nicht so aus, wie man es nach der obigen Beschreibung erwarten könnte. *Synchronize* besteht also nicht aus einer Rechenzeit verschwendenden Schleife, in der beispielsweise ein Flag abgefragt wird. Statt dessen basiert *Synchronize* darauf, dass sich die VCL selbst eine Nachricht in den Haupt-Thread sendet. Da die VCL die eingehenden Nachrichten schön der Reihenfolge nach aus ihrem »Briefkasten« entnimmt, kann sie davon ausgehen, dass sie zu dem Zeitpunkt, an dem sie die selbst gesendete Nachricht von *Synchronize* erhält, nicht mitten in der Bearbeitung einer anderen Nachricht steckt. Die Bearbeitung der selbst gesendeten Nachricht besteht nun darin, die an *Synchronize* übergebene Methode aufzurufen. Diese Methode wird also als Teil des Haupt-Threads aufgerufen und nicht als Teil des Threads, zu dessen *TThread*-Objekt sie gehört.

Der TThread-Konstruktor

Es wird nun Zeit, den restlichen Quelltext für *TDemoThread* nachzuliefern; es handelt sich um die Erweiterung des automatisch erstellten Klassengerüsts, das die Variablen enthält, die in den oben abgedruckten Methoden bereits verwendet wurden, sowie um den Konstruktor *Create*:

```
type
  TDemoThread = class(TThread)
  protected
    PaintBox: TPaintBox;
    Progress: Integer;

    procedure Execute; override;
    procedure Update;
  public
    constructor Create(PB: TPaintBox);
  end;

constructor TDemoThread.Create(PB: TPaintBox);
begin
  inherited Create(True);
  PaintBox := PB;
  InitProgress(Progress);
  Resume;
end;
```

Create muss natürlich als *public* deklariert sein, damit er von außen aufgerufen werden kann. Seine Aufgabe ist es in diesem Fall, eine *TPaintBox*-Komponente entgegenzunehmen, in die der Thread später seine Grafik zeichnen soll. *Create* hebt diese PaintBox-Referenz in der Variablen *PaintBox* auf, die von der oben gezeigten Methode *Update* verwendet wird.

Der von *TThread* vordefinierte Konstruktor erwartet einen booleschen Parameter, der besagt, ob der Thread zunächst einmal angehalten werden soll (im Falle von *True*). *TDemoThread.Create* ruft diesen geerbten Konstruktor immer mit dem Parameter *True* auf, was bedeutet, dass der Thread nicht sofort gestartet wird, sondern erst, wenn die Methode *Resume* aufgerufen wird. *Resume* sollte in diesem Fall erst nach abgeschlossener Initialisierung aufgerufen werden.

Starten eines Threads

Sehen wir uns nun an, wie man ein *TThread*-Objekt vom Haupt-Thread aus steuern kann. Als Erstes muss das *TThread*-Objekt wie jedes andere Objekt, das nicht schon im Formularentwurf eingebaut wurde oder anderweitig vordefiniert ist, über den Konstruktoraufruf konstruiert werden. Das Beispielprogramm tut dies in der Methode für das *OnCreate*-Ereignis:

```
// Auszug aus der Klassendeklaration von TThreadDemo1Form:
  public
    Thread1: TDemoThread;

procedure TThreadDemo1Form.FormCreate(Sender: TObject);
begin
  ... // Initialisierung für OnIdle, siehe Kapitel 4.7.1
  // der Thread soll in der PaintBox3 (links unten) zeichnen:
  Thread1 := TDemoThread.Create(PaintBox3);
```

Da *TDemoThread* den *Create*-Konstruktor von *TThread* überschreibt, entfällt hier der boolesche Parameter, der normalerweise angibt, ob der *Thread* nicht sofort gestartet werden soll.

Manipulation eines Threads

Sobald der Thread mit *Create* erzeugt ist, können Sie ihn auf verschiedene Weise manipulieren. Wie üblich stellt die Klasse *TThread* zu diesem Zweck Methoden, Properties und Events, in diesem Fall allerdings nur ein einziges, zur Verfügung. Diese Klassenelemente sind in der folgenden Tabelle zunächst einmal zusammengefasst:

TThread-Element	**Aufgabe**
constructor Create (CreateSuspended: Boolean);	erzeugt das Objekt; falls *CreateSuspended True* ist, wird der Thread nicht sofort gestartet – dies kann z.B. mit *Resume* nachgeholt werden.
function WaitFor: Integer;	wartet, bis der Thread beendet ist, und liefert den Rückgabewert des Threads als Ergebnis zurück. Das Warten des aktuellen Threads besteht darin, dass seine Ausführung angehalten wird.

TThread-Element	**Aufgabe**
procedure Resume;	nimmt die Ausführung des Threads wieder auf, falls der Thread noch nicht gestartet oder mit *Suspend* oder *Suspended:=True* angehalten wurde. Der Thread wird allerdings erst dann fortgesetzt, wenn *Resume* genauso oft aufgerufen wurde wie *Suspend*.
procedure Suspend;	hält die Ausführung des Threads an und erhöht einen internen Zähler um eins, so dass mehrere Aufrufe von *Suspend* mehrere Aufrufe von *Resume* erfordern, bis der Thread fortgesetzt wird.
procedure Terminate;	setzt das interne *TThread*-Property *Terminated*. Damit der Thread tatsächlich beendet wird, müssen Sie dieses Flag in der *Execute*-Methode abfragen und die Methode verlassen, wenn es den Wert *True* erhält.
property FreeOnTerminate: Boolean;	Wenn Sie dieses (per Voreinstellung auf *False* gesetzte) Flag auf *True* setzen, gibt die VCL das Thread-Objekt automatisch mit einem Aufruf von *Free* frei, sobald der Thread beendet ist. (Vorsicht: Eventuell auf das Thread-Objekt weisende Zeiger werden natürlich nicht auf *nil* gesetzt.)
property Handle: THandle;	kann nur gelesen werden und enthält einen Bezeichner für den Thread, der für einige direkte Win32-Aufrufe benötigt wird. In einigen Fällen wird auch der Bezeichner im Property *ThreadId* verwendet.
property Priority: TThreadPriority;	Hier können Sie durch einen der Werte *tpIdle*, *tpLowest*, *tpLower*, *tpNormal*, *tpHigher*, *tpHighest* und *tpTimeCritical* angeben, mit welcher Priorität der Thread vom Betriebssystem Rechenzeit zugeteilt bekommt.
property Suspended: Boolean;	Setzen des Wertes *True* bewirkt den Aufruf von *Suspend*, falls der Thread noch nicht angehalten wurde. Umgekehrt führt ein Setzen des Wertes *False* zum Aufruf der Methode *Resume*, falls der Thread nicht bereits läuft.
property ThreadID: THandle;	kann nur gelesen werden, siehe Property *Handle*.
property OnTerminate: TNotifyEvent;	Dies ist das einzige Event dieser Klasse; es wird ausgelöst, sobald die Methode *Execute* beendet wurde und kurz bevor das Objekt gelöscht wird (falls *FreeOnTerminate=True*). Die Bearbeitungsmethode für dieses Event hat das einfache Format *»xxxTerminate(Sender: TObject);«*.

Im Beispielprogramm können Sie die wichtigsten Eigenschaften des Threads zur Laufzeit verändern (so wie auch die Eigenschaften des zweiten Threads, der im nächsten Kapitel besprochen werden wird): Das Formular enthält zwei *TTrackBar*-Komponenten zur Einstellung der Priorität und zwei Markierungsschalter, mit denen Sie die Threads anhalten können. Bei Betätigung von TrackBar (*TrackBar1*) und Markierungsschalter

(*Panel3Active*) für den ersten Thread werden die folgenden beiden Ereignisse ausgelöst, die auf die Properties zurückgreifen, die in der Tabelle beschrieben wurden:

```
procedure TThreadDemo1Form.TrackBar1Change(Sender: TObject);
begin
  Thread1.Priority := TThreadPriority(TrackBar1.Position);
  UpdatePriorityCaptions; // Methode des Formulars
end;

procedure TThreadDemo1Form.Panel3ActiveClick(Sender: TObject);
begin
  Thread1.Suspended := not Panel3Active.Checked;
end;
```

Das Laufzeitverhalten im Vergleich zu OnIdle/ProcessMessages

Wie schon im Abschnitt 4.7.1 angedeutet, ist das Verhalten des Threads im Beispielprogramm gegenüber den beiden kooperativen »Konkurrenten« durch eine besondere Gleichmäßigkeit charakterisiert. Selbst wenn Sie das Systemmenü aufklappen oder das Fenster verschieben, zeichnet der Thread munter weiter. Erst wenn Sie eine 16-Bit-Anwendung in den Vordergrund holen und diese das System mit einer längeren Berechnung blockiert, wird auch der Thread unterbrochen, und das auch nur, solange Sie kein Windows NT verwenden.

Vielleicht finden Sie es verwunderlich, dass der Thread erst bei sehr hoher Priorität mit der Zeichengeschwindigkeit von *OnIdle* mithalten kann. Dies liegt jedoch nicht daran, dass die Verwendung von *OnIdle* zu einem Geschwindigkeitsvorteil führt, und es bedeutet auch nicht, dass Sie einem Thread eine sehr hohe Priorität geben müssen, damit dieser effektiv läuft. Der Geschwindigkeitsnachteil des Threads ist vielmehr darin begründet, dass der Thread sich ständig über den Aufruf von *Synchronize* mit dem Haupt-Thread synchronisiert. Die eigentliche Berechnung nimmt, verglichen mit der Wartezeit der *Synchronize*-Methode, so gut wie keine Zeit in Anspruch. Diese Benachteiligung ändert sich erst, wenn Sie die »Dauer der Berechnung« erhöhen – oder wenn Sie den Nachteil des *Synchronize* gänzlich beseitigen.

Lock und Unlock für die Grafikausgabe

Seit Delphi 3 können Sie in Threads, die die VCL nur zur Grafikausgabe benötigen, auf die Verwendung der *Synchronize*-Methode verzichten und statt dessen die neuen *TCanvas*-Methoden *Lock* und *Unlock* aufrufen. Mit *Lock* sperren Sie die Zeichenfläche so lange vor dem Zugriff durch andere Threads, bis Sie *Unlock* aufrufen (Voraussetzung dafür ist, dass alle anderen Threads ebenfalls *Lock* aufrufen, bevor sie auf die Zeichenfläche zugreifen). Um andere Bereiche der VCL von Threads aus aufzurufen, benötigen Sie auch in Delphi 6 noch die *Synchronize*-Methode.

Da die Beispielthreads von der VCL nur die Klasse *TCanvas* benötigen, können Sie den Unterschied gleich an diesem Beispielprogramm ausprobieren: Ohne *Synchronize*, aber mit *Lock/Unlock* arbeitet der erste Thread des Beispielprogramms erheblich schneller, was nicht verwundert, da *Lock* nur so lange wartet, bis kein anderer Thread mehr die Zeichenfläche verwendet, während *Synchronize* so lange warten muss, bis überhaupt keine Aufrufe der VCL mehr laufen.

Wenn das Feld *Synchronize verwenden* im Beispielprogramm nicht markiert ist, ruft *TDemoThread.Execute* statt *Synchronize(Update)* einfach *Update* auf. Dafür muss dann aber die Grafikausgabe aller Threads durch Aufrufe von *Lock* und *Unlock* abgesichert werden. In vereinfachter Fassung sieht die Grafikausgabe der Unit *GlobalProcs* dann wie folgt aus:

```
Canvas.Lock;
Canvas.Brush.Color:=...
Canvas.Ellipse(...);
Canvas.Unlock;
```

Der nächste Abschnitt zeigt, wie die Berechnung noch effektiver vorangetrieben werden kann; in diesem Zusammenhang wird auch endlich das letzte Fensterviertel des Beispielprogramms gefüllt.

4.7.3 Mehrere Threads und deren Synchronisation

Wenn eine Anwendung aus mehreren Threads besteht, so müssen sich diese viele Ressourcen der Anwendung teilen, wodurch Konflikte entstehen können. Dieses Kapitel beschreibt ein zweites Beispielprogramm, das die Häufigkeit dieser auf den ersten Blick selten auftretenden Konflikte demonstriert und zeigt, wie derartige Konflikte durch die Verwendung von Synchronisationsobjekten vermieden werden können. Wir beginnen jedoch mit einem einfachen Beispiel für ein Programm mit zwei zusätzlichen Threads, bei dem noch keine weitere Synchronisation erforderlich ist.

Verbesserung von ThreadDemo1 durch einen zweiten Thread *R92*

Und zwar geht es ein letztes Mal um das Beispielprogramm des letzten Kapitels, dessen Fenster noch ein leer stehendes Viertel zu beklagen hat. Im letzten Kapitels haben wir festgestellt, dass der Thread durch das ständige Warten auf die Synchronisation der Grafikausgabe oder auf *TCanvas.Lock/Unlock* stark gebremst wurde und unter Umständen sogar noch erheblich langsamer als die *OnIdle*-Methode war. Es gibt mehrere Möglichkeiten, diese Bremse zu lösen:

- Sie könnten im Thread gänzlich darauf verzichten, die Bildschirmanzeige unter Verwendung der VCL zu aktualisieren. Der Thread müsste dann kein *Synchronize* aufrufen und könnte immer so schnell arbeiten, wie es der Prozessor und die anderen Threads erlauben. Um dennoch eine Rückmeldung zu erhalten, wie schnell der

Thread vorankommt, könnte der Haupt-Thread in gewissen Abständen – z.B. im *OnIdle*- oder in einem Timer-Event – einen Fortschrittszähler (in diesem Fall die Variable *TDemoThread.Progress*) abfragen und den Fortschritt in einer der Komponenten oder Zeichenflächen anzeigen.

- Das Beispielprogramm geht einen ähnlichen Weg: Es lagert die Aktualisierung des Bildschirms für das rechte untere Fensterviertel in einen weiteren Hilfs-Thread aus. Dieser muss dann zwar mit *Synchronize* auf die VCL warten, hält damit aber den zweiten Thread nicht auf. Während der Hilfs-Thread also noch darauf wartet, den letzten Fortschritt am Bildschirm ausgeben zu dürfen, kann der eigentliche Rechen-Thread in der Zwischenzeit weiterrechnen.

Die Unit *Thread2* enthält die beiden genannten Threads, wobei der Rechen-Thread *TDemoThread2* und der Hilfs-Thread *TUpdateThread* heißt. Gegenüber dem im letzten Kapitel gezeigten Thread *TDemoThread* wird die *while*-Anweisung der Methode *Execute* des neuen Threads um den bremsenden Aufruf von *Synchronize(Update)* verkürzt, so dass nur noch übrig bleibt:

```
procedure TDemoThread2.Execute;
begin
  while not Terminated do
    GlobalProcs.SimulateWork(Progress);
end;
```

Ein weiterer Thread wiederholt nun ununterbrochen den *Synchronize*-Aufruf, der aus dem ersten Thread entfernt wurde; die Variable *ThreadToUpdate* wurde zuvor auf diesen ersten Thread gesetzt:

```
procedure TUpdateThread.Execute;
begin
  while not Terminated do
    Synchronize(Update);
end;

procedure TUpdateThread.Update;
begin
  GlobalProcs.ShowProgress(ThreadToUpdate.PaintBox,
                           ThreadToUpdate.Progress);
end;
```

Was bleibt, ist das Starten beider Threads in der *OnCreate*-Methode des Formulars. Der *Thread2* erhält nun die vierte PaintBox als Arbeitsgebiet zugewiesen (allerdings wird diese PaintBox nur im Thread *UpdateThread* benutzt); *UpdateThread* wird anschließend mit der niedrigsten Priorität *tpIdle* gestartet, damit er nicht unnötig viel Rechenzeit in Anspruch nimmt:

```
procedure TThreadDemo1Form.OnCreate(Sender: TObject);
begin
```

```
...
Thread2 := TDemoThread2.Create(PaintBox4);
UpdateThread := TUpdateThread.Create(Thread2);
UpdateThread.Priority := tpIdle;
```

Ein Programmstart zeigt, dass *Thread2* bei gleicher Priorität tatsächlich erheblich schneller vorankommt als der erste Thread und auch die beiden kooperativen Methoden weit übertrifft (dies liegt zu einem guten Teil natürlich wieder daran, dass die eigentliche Berechnung kaum ins Gewicht fällt, wodurch die kooperativen Methoden natürlich erheblich benachteiligt werden). Allerdings ist das Gespann aus den beiden neuen Threads weniger gut für optische Effekte geeignet, da das Aktualisieren der Bildschirmanzeige in *UpdateThread* zu selten stattfindet. *TDemoThread2* macht zwischen zwei Zeichenvorgängen so große Fortschritte, dass keine zusammenhängende Ellipse mehr gezeichnet wird, sondern die einzelnen Kreise vielmehr an zufällig wirkenden Positionen erscheinen. (Auch hier lohnt es sich wieder, den Parameter »Dauer der Berechnung« zu erhöhen.)

TThread.WaitFor

Bevor wir zu einem Beispiel für den gleichzeitigen Datenzugriff in mehreren Threads kommen, sei hier auf die einfachste Art der Thread-Synchronisation hingewiesen, für die nur eine Methode aufgerufen werden braucht: Mit *TThread.WaitFor* synchronisieren Sie den Thread, aus dem Sie *WaitFor* aufrufen, auf das Ende des Threads, dessen *WaitFor*-Methode aufgerufen wird. *WaitFor* besteht nicht etwa aus einer Warteschleife, sondern bewirkt, dass der aufrufende Thread so lange angehalten wird, bis der andere Thread beendet ist.

WaitFor ist damit eine besonders einfache Möglichkeit sicherzustellen, dass nur einer von zwei Threads gleichzeitig aktiv ist, allerdings bewirkt sie, dass die Threads nacheinander ablaufen, was eigentlich nicht das Ziel mehrerer Threads sein sollte.

Gleichzeitige Datenzugriffe

Manchmal müssen mehrere Threads auf gemeinsame Daten zugreifen. Da die Variablen der Anwendung allen Threads innerhalb der Anwendung zugänglich sind, ist dies technisch kein Problem. Für die Funktion der Anwendung bedeutet ein gleichzeitiger mehrfacher Zugriff jedoch eine Gefahr:

- Wenn mehrere Threads auf die gleichen Daten schreibend zugreifen, kann es sehr schnell zur Zerstörung der Daten kommen. So kann es zum Beispiel kaum funktionieren, dass zwei Threads die gleiche Liste gleichzeitig in aufsteigender und in absteigender Reihenfolge sortieren.

- Auch wenn nur einer von mehreren Threads die Daten verändert, kann es die anderen Threads ins Chaos stürzen, wenn sie die Daten während einer solchen Veränderung lesen und die Veränderung gerade in einem Zwischenstadium befindlich ist, in dem die Daten keinen oder einen falschen Sinn ergeben. (Ein solches Zwischenstadium ist nur bei Variablen denkbar, die vom Prozessor nicht in einem Schritt gesetzt werden. Es kommt natürlich auch darauf an, dass der Compiler die entsprechenden Anweisungen erzeugt und ein *Word* nicht beispielsweise als zwei Einzelbytes beschreibt.)

Wie man dieses Problem auf einfache Weise vermeiden kann, demonstriert die VCL: Man verbietet einfach, dass mehrere Threads, die auf die gleichen Daten zugreifen wollen, gleichzeitig laufen. Nichts anderes ist die Aufforderung zur Verwendung der *Synchronize*-Methode, die sicherstellt, dass die VCL immer nur einmal gleichzeitig aktiv ist und folglich auch immer nur einmal auf ihre Daten zugreifen kann.

Hinweis: Wenn eine Funktion oder eine ganze Bibliothek mit sich selbst in Konflikt gerät, falls sie in mehreren Threads gleichzeitig aktiv ist, bezeichnet man diese Funktion bzw. Bibliothek als *nicht wiedereintrittsfähig* (non reentrant). Die Wiedereintrittsfähigkeit einer Funktion lässt sich beispielsweise dadurch erreichen, dass die Funktion nur lokale Variablen verwendet und nur Funktionen aufruft, die ihrerseits wiedereintrittsfähig sind.

Ein Beispielprogramm mit gleichzeitigem Datenzugriff

Das Beispielprogramm *SyncDemo* aus Abbildung 4.15 startet zwei Paare von Threads, wobei die Threads innerhalb eines Paares auf dieselben Daten zugreifen müssen und nur das zweite Paar richtig synchronisiert ist. Im Memofeld des Fensters wird angezeigt, wie häufig es in den beiden Thread-Paaren zu Zugriffskonflikten kommt.

Sehen wir uns zunächst die Aufgaben der Threads an:

- Jeder Thread soll in ständiger Wiederholung das gesamte Array *Bytes* aus der unten gezeigten Datenstruktur bearbeiten.
- Dabei füllt es die einzelnen Array-Elemente mit Zahlen und setzt danach alle Elemente des Arrays wieder auf Null.
- Nur wenn alle Array-Elemente auf Null gesetzt sind, darf ein anderer Thread auf das Array zugreifen. Diese frei erfundene Regel soll für das Beispielprogramm alle Zustände des Arrays, in denen nicht alle Elemente Null sind, als »Zwischenzustände« definieren, während derer nur ein Thread Zugriff auf das Array haben darf.

Das Daten-Array ist im falsch synchronisierten Thread wie folgt definiert:

```
const
  DataCount = 100;
type
  TThreadData = record
    InUseFlag: Boolean;
    Bytes: array[1..DataCount] of Byte;
  end;
```

Das *InUseFlag* soll speichern, ob das Array gerade von einem der beiden Threads verwendet wird. Bevor ein Thread das Array *Bytes* verändert, soll er warten, bis *InUseFlag* auf *False* gesetzt wird, bis also der andere Thread seine Bearbeitung des Arrays beendet hat. Bevor er das Array selbst verändert, setzt er seinerseits *InUseFlag* auf *False*, damit der zweite Thread warten muss.

Der eigentliche Fehler in der Synchronisation besteht nun nicht darin, dass beide Threads gleichzeitig auf *Bytes* zugreifen, sondern dass sie beide gleichzeitig das *InUseFlag* modifizieren (als Folge davon greifen sie dann auch gleichzeitig auf das Array zu):

```
procedure TSyncThread.Falsch; // einer von vielen falschen Wegen
var
  CollisionDetected: Boolean;
  i: Integer;
begin
  // Data weist hier auf eine von beiden Threads benutzte
  // TThreadData-Struktur
  repeat
    // Warten, bis der Datenblock nicht mehr verändert wird:
    while Data^.InUseFlag do ;
    // Datenblock reservieren:
    Data^.InUseFlag := True;
    inc(IterationCount); // nur für die Statistik im Fenster
    // Überprüfen, ob sich der Datenblock tatsächlich im
    // Ursprungszustand (gefüllt mit 0-Bytes) befindet:
    CollisionDetected := False;
    for i := 1 to DataCount do
      if Data^.Bytes[i] <> 0 then
        CollisionDetected := True;
    if CollisionDetected then // Kollision
      inc(CollisionCount); // für die Statistik im Fenster
    // Die eigentliche Operation, die immer nur ein
    // Thread gleichzeitig ausführen soll, und zwar
    // den Datenblock zu verändern, folgt hier:
    for i := 1 to DataCount do
      Data^.Bytes[i] := i;
    // Datenblock in den definierten Ausgangszustand
    // zurückversetzen:
    for i := 1 to DataCount do
      Data^.Bytes[i] := 0;
```

```
      // Datenblock freigeben:
      Data^.InUseFlag := False;
    until False;
  end;
```

Bevor diese Methode den eigentlichen Datenzugriff durchführt, überprüft sie, ob die Synchronisation über das *InUseFlag* funktioniert hat. Dazu testet sie, ob alle Elemente des Arrays Null sind (dieser Test findet zwar nicht alle Fälle, in denen beide Threads gleichzeitig auf das Array zugreifen, aber doch so viele, dass es für eine Demonstration ausreicht). Falls die Methode feststellt, dass das Array nicht vorschriftsmäßig mit Nullen gefüllt ist, erhöht sie den Zähler *CollisionCount*.

Hinweis: Der Zähler *CollisionCount* wird durch Methoden, die hier nicht abgedruckt sind, im Programmfenster angezeigt, siehe Abbildung 4.15. Die Aktualisierung des Fensters findet übrigens zur Abwechslung einmal mit Hilfe einer *Timer*-Komponente statt.

Bei Ablauf des Programms zeigt sich, dass diese Art der Synchronisation tatsächlich nicht immer funktioniert: In seltenen Fällen stellen beide Threads fest, dass sie auf ein Array zugreifen, das nicht vollständig mit Nullen gefüllt ist, und erhöhen daher den Zähler *CollisionCount*. Zwar tritt dieser Fall weniger als einmal alle 10000 Durchläufe auf; da moderne Prozessoren jedoch mit diesen Durchläufen sehr schnell sind, treten die Fehler immer noch schneller auf, als der Anwender bei einer echten Anwendung Fehlermeldungsfenster schließen könnte.

Abbildung 4.15: Zugriffskonflikte im Beispielprogramm SyncDemo

Die Kollisionsursache

Die Ursache des Fehlers liegt darin, dass das Betriebssystem die beiden Threads auch zwischen den beiden folgenden Zeilen unterbrechen kann:

```
while Data^.InUseFlag do ;
Data^.InUseFlag := True;
```

Angenommen, Thread 1 stellt fest, dass *InUseFlag = False*. Er beendet die *while*-Schleife und will das Flag nun setzen, wird aber vorher vom System unterbrochen, das nun zum Thread 2 wechselt. Auch dieser führt die obige *while*-Anweisung aus und stellt fest, dass das Flag gelöscht ist (da Thread 1 es noch nicht setzen konnte). Nun setzt Thread 2 das Flag, und wenn das Betriebssystem wieder zu Thread 1 wechselt, tut dieser dasselbe. Beide Threads führen nun den Code aus, der eigentlich nur einmal gleichzeitig aktiv sein sollte.

Es genügt also nicht, dass der Prozessor das *InUseFlag* in jeweils einem Schritt setzen oder lesen kann, er müsste sogar in der Lage sein, Lese- und Schreibzugriff zusammen in einem nicht unterbrechbaren Schritt auszuführen, damit es nicht zu einem Konflikt kommen könnte.

Interlocked-Funktionen des Windows-API

Das Win32-API bietet Ihnen verschiedene Funktionen mit dem Namensanfang »Interlocked« an, die zwei dieser oder ähnlicher Elementaroperationen zu einer nicht unterbrechbaren Einheit zusammenfassen. Der obige Programmauszug könnte beispielsweise durch die kollisionssichere Schleife *while InterlockedExchange (Data^.InUseFlag, 1)=1 do;* ersetzt werden (*InUseFlag* müsste dazu als *Integer* deklariert und zur Freigabe auf 0 statt auf *False* gesetzt werden).

Es gibt jedoch wesentlich elegantere Möglichkeiten, bei der der wartende Prozess die Prozessorzeit nicht in einer Schleife verschwenden muss, sondern ganz unterbrochen wird, bis das erwartete Ereignis eintritt. Die Lösung mit *InterlockedExchange* könnte außerdem nicht garantieren, dass ein einmal laufender Thread jemals die Kontrolle an den wartenden Thread abgibt. Auch diese missliche Situation kann bei der Verwendung von Synchronisationsobjekten vermieden werden.

Synchronisation mit TCriticalSection R125

Dieses Beispiel sollte ausreichend Motivation liefern, die von der Delphi-Laufzeitbibliothek bereitgestellten Synchronisationsobjekte zu verwenden. Dazu ersetzen wir zunächst das gescheiterte *InUseFlag* durch die Variable *CritSec*, die grob gesehen dieselbe Funktion hat wie das *InUseFlag*:

```
uses SyncObjs, ... ; // << SyncObjs = Unit von TCriticalSection

type
  TThreadData = record
    CritSec: TCriticalSection;
    Bytes: array[1..DataCount] of byte;
  end;
```

Bei der Initialisierung der Threads (im *OnCreate*-Event des Formulars) wird diese Variable initialisiert (genauere Erläuterungen folgen später):

```
SharedData2.CritSec := TCriticalSection.Create;
```

Der bisherige direkte Zugriff auf das *InUseFlag* wird durch zwei Methodenaufrufe von *CritSec* ersetzt:

- Ein Aufruf von *Enter* ersetzt die Abfrage des *InUseFlag* und das anschließende Setzen dieses Flags.
- Der Aufruf von *Leave* tritt an die Stelle der Zuweisung *InUseFlag := False*.

Die geänderten Teile der Methode sehen nun wie folgt aus:

```
procedure TSyncThread.Richtig; // einer von vielen richtigen Wegen
var
  CollisionDetected: Boolean;
  i: Integer;
begin
  repeat
    // Warten, bis der Datenblock nicht mehr verändert wird
    // und gleichzeitig reservieren:
    Data^.CritSec.Enter;
    inc(IterationCount);
    ... wie oben ...
    for i := 1 to DataCount do
      Data^.Bytes[i] := 0;
    // Sperre freigeben:
    Data^.CritSec.Leave;
  until False;
end;
```

Zur Funktionsweise von TCriticalSection

TCriticalSection symbolisiert, wie der Name schon andeutet, einen kritischen Programmabschnitt; kritisch deshalb, weil er nur einmal gleichzeitig aktiv sein darf. Die Methoden *Enter* und *Leave* symbolisieren das Betreten und das Verlassen dieses kritischen Bereichs. Die Verantwortung von *TCriticalSection* liegt nun darin, jeweils nur einem Aufrufer von *Enter* gleichzeitig den Zugang zu diesem Programmabschnitt zu gestatten. Wenn also zwei Programmteile nahezu gleichzeitig *Enter* aufrufen, dann erhält der Teil, der zuerst zum Aufruf gekommen ist, den Zuschlag. Die *Enter*-Methode kehrt zum Aufrufer

zurück und dieser kann nun den kritischen Programmteil abarbeiten. Der zweite Aufrufer von *Enter* muss warten, und zwar dadurch, dass Enter *nicht* zurückkehrt, bis der kritische Abschnitt wieder frei ist, bis also der erste Programmteil die *Leave*-Methode aufgerufen hat.

TCriticalSection ist damit quasi ein Türsteher, der darauf achtet, dass sich immer nur eine Person in einem bestimmten Raum aufhält. Die Überprüfung, ob der Raum bereits belegt ist, obliegt dabei ausschließlich dem Türsteher. Die Besucher kommen einfach mit der Aufforderung »Enter« und bekommen sie mal sofort erfüllt, mal erst nach einer Wartezeit.

Zu Anfang wurde die Klasse *TCriticalSection* als »symbolisch« beschrieben. Damit war gemeint, dass ein Objekt dieser Klasse nicht selbst der kritische Programmabschnitt ist, dass es also keine wirkliche Kontrolle über diesen Programmabschnitt hat. Die tatsächliche Verantwortung liegt nämlich beim Programmierer, der die Methoden *Enter* und *Leave* korrekt aufrufen muss. Vergisst er etwa den Aufruf von *Enter*, wird *TCriticalSection* machtlos. Wichtig ist auch, dass pro kritischem Programmabschnitt nur ein *TCriticalSection*-Objekt angelegt wird. Für das Beispielprogramm bedeutet das: ein *CriticalSection*-Objekt pro Datenstruktur, auf die nur einmal gleichzeitig zugegriffen werden darf. Falsch wäre es dagegen, das *CriticalSection*-Objekt als Variable der Thread-Klasse anzulegen, denn dann gäbe es ja ein *CriticalSection*-Objekt pro Thread und die *Enter*- und *Leave*-Aufrufe wären vollkommen belanglos.

Es kann auch ohne weiteres sein, dass mehrere Progammabschnitte sich eine *CriticalSection* teilen. Das Beispielprogramm etwa besitzt einen Schalter ANHALTEN, mit dem der Benutzer einen der beiden Threads anhalten kann. Hier ist es sehr wichtig, dass der Thread nicht angehalten wird, während er sich im kritischen Bereich befindet. Denn dann würde die *Leave*-Methode nicht aufgerufen und auch der andere Thread könnte den kritischen Bereich nicht betreten, er wäre durch das Anhalten des ersten Threads blockiert.

Dies kann dadurch vermieden werden, dass die Bearbeitungsmethode für den ANHALTEN-Schalter vor dem Anhalten des Threads selbst die *Enter*-Methode aufruft. Nach dem *Enter*-Aufruf kann sie sicher sein, dass sich keiner der beiden Threads im kritischen Bereich befindet. In dieser Situation kann nun der vom Benutzer gewählte Thread angehalten werden. Danach wird natürlich sofort *Leave* aufgerufen, denn mittlerweile wartet vielleicht schon der nicht angehaltene Thread vor der Schwelle zum kritischen Bereich:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ThreadSelection.ItemIndex>1 then begin
    SharedData2.CritSec.Enter;
    T[ThreadSelection.ItemIndex].Suspend;
```

```
      SharedData2.CritSec.Leave;
    end;
  end;
```

Dieses Listing dient nur als weiteres Beispiel für die Verwendung von *Enter* / *Leave*. Tatsächlich hält das Beispielprogramm die Threads nicht mit der *Suspend*-Methode an, sondern auf eine im weiteren Kapitelverlauf beschriebene Weise.

TEvent

Eine zweite Synchronisationsklasse der Unit *SyncObjs* ist *TEvent*. Sie kapselt die entsprechenden Synchronisationsfunktionen des Betriebssystems (Event-Funktionen unter Windows, Semaphoren-Objekte unter Linux). Ein *TEvent*-Objekt besitzt zwei Zustände, die in den Methoden als »set« und »reset« bezeichnet werden, also quasi »gesetzt / ungesetzt« oder »signalisiert / nicht signalisiert« bedeuten. *TEvent* wird durch zwei Methoden gesteuert:

- *WaitFor* ähnelt der Methode *TCriticalSection.Enter*, denn beide warten sie auf ein bestimmtes Ereignis. Bei *TEvent* besteht dieses Ereignis darin, dass der Zustand des Objekts auf »gesetzt« geschaltet wird. Sobald dies passiert, kehrt *WaitFor* zum Aufrufer zurück, allerdings schaltet es vorher noch den Zustand wieder auf »ungesetzt« .
- *SetEvent* »setzt« das Event, gibt also einer eventuell mit *WaitFor* wartenden Methode das Signal, das sie jetzt weiterarbeiten darf.

Hinweis: *TEvent* besitzt zwar noch eine Methode *ResetEvent*, die jedoch unter Linux nichts tut, da das Event vor dem Ende der *WaitFor*-Methode automatisch zurückgesetzt wird.

Beim nächsten Thema findet sich auch eine Gelegenheit für die Anwendung von *TEvent*.

Eine neue Suspend-Methode

Das Beispielprogramm *SyncDemo* verwendet *TEvent*, um eine Funktion zum Anhalten der beiden Threads zu implementieren, die ohne die *Suspend*-Methode auskommt, denn diese hat zwei Nachteile:

- Sie ist nicht besonders gut portabel, da sie unter Kylix intern nur mit einem Trick funktioniert, der von der Implementierung des Linux-Kernels abhängig ist.
- Das simple Anhalten eines Threads kann zum Blockieren anderer Threads führen, nämlich wenn der angehaltene Thread über Systemressourcen verfügt, auf deren Freigabe gerade ein anderer Thread wartet.

Die Fenster der Beispielprogramme *ThreadDemo1* und *SyncDemo* verfügen daher zwar über Schalter, mit denen die Threads angehalten werden können, aber sie verwenden nicht die *Suspend* / *Resume*-Methoden von *TThread*, sondern die neue Unit *SuspendableThread*, die Sie auf der CD finden.

Die Grundüberlegung dabei ist, dass der zweite Nachteil der *Suspend*-Methode, das drohende Blockieren anderer Threads, nur dadurch vermieden werden kann, dass der angehaltene Thread kooperiert, indem er sich selbst anhält, sobald er keine wichtigen Systemressourcen mehr innehat. Sich selbst anzuhalten, darin besteht nämlich keine Schwierigkeit, und dafür bedarf es auch nicht der vordefinierten Methode *TThread.Suspend*. In den zurückliegenden Abschnitten wurden ja bereits zwei unterschiedliche Konstrukte vorgestellt (*CriticalSection* und *Event*), durch die ein Thread zum Warten verpflichtet wird.

Für das Anhalten von Threads eignen sich die *Events* besonders gut. Denn das Anhalten eines Threads kann verstanden werden als »Warten auf das Fortsetzen-Event«. Jeder Thread, der mit Hilfe eines *TEvent*-Objekts angehalten werden soll, benötigt daher ein eigenes *TEvent*-Objekt. Die Unit *SuspendableThread* definiert daher gleich eine neue Thread-Klasse *TSuspendableThread*, die von *TThread* abgeleitet ist und die private Variable *ResumeEvent* einführt. *TSuspendableThread* besitzt auch eine neue *Resume*-Methode, die das »Fortsetzen-Event« setzt, also quasi die Ampel auf Grün schaltet:

```
procedure TSuspendableThread.Resume2;
begin
  ResumeEvent.SetEvent;
end;
```

Nicht ganz so einfach ist die Implementation der neuen *Suspend*-Methode. Sie soll ja den Thread nicht abrupt anhalten, sondern ihm Gelegenheit lassen, alle wichtigen Systemressourcen freizugeben, auf die andere Threads warten könnten.

Dies wird in der Klasse *TSuspendableThread* nun dadurch realisiert, dass *Suspend2* lediglich eine private Variable namens *FSuspendRequest* setzt, aber den Thread nicht anhält. Der eigentliche Thread-Code (die Methode *Execute* und alle von ihr aufgerufenen Methoden) ist dafür verantwortlich, bei passenden Gelegenheiten *FSuspendRequest* abzufragen und sich selbst anzuhalten, falls *FSuspendRequest* den Wert *True* hat. Zum Anhalten wird die folgende Anweisung verwendet, die so lange wartet, bis die oben gezeigte *Resume2*-Methode aufgerufen wird:

```
ResumeEvent.WaitFor($FFFFFFFF);
```

Das folgende Listing zeigt die komplette Unit *SuspendableThread*, die den bereits gezeigten entscheidenden Anweisungen noch weitere Verwaltungstätigkeiten hinzuzufügen hat, etwa das Initialisieren des Objekts über die Klasse *TSimpleThread* (eben-

falls in *SyncObjs* deklariert und ein Nachfahre von *TEvent*, der besonders einfach zu verwenden ist), die Freigabe des *TEvent*-Objekts im Destruktor und die Bereitstellung eines Properties *Suspended2* als Alternative zum *TThread*-Property *Suspended*:

```
unit SuspendableThread;

interface

uses Classes, SyncObjs;

type
  TSuspendableThread = class(TThread)
  private
    ResumeEvent: TEvent;
    FSuspended2: Boolean;
    FSuspendRequest: Boolean;
    procedure SetSuspended2(const Value: Boolean);
  public
    property Suspended2: Boolean read FSuspended2 write SetSuspended2;
    procedure CheckSuspendRequest;
    procedure Suspend2;
    procedure Resume2;
    destructor Destroy; override;
    constructor Create(InitSuspended: Boolean);
  end;

implementation

{ TSuspendableThread }

constructor TSuspendableThread.Create(InitSuspended: Boolean);
begin
  inherited Create(InitSuspended);
  ResumeEvent := TSimpleEvent.Create;
end;

procedure TSuspendableThread.CheckSuspendRequest;
begin
  if FSuspendRequest = True then begin
    FSuspendRequest := False;
    FSuspended2 := True;
    ResumeEvent.WaitFor($FFFFFFFF);
    FSuspended2 := False;
  end;
end;

procedure TSuspendableThread.Suspend2;
begin
  FSuspendRequest := True;
end;
```

```
destructor TSuspendableThread.Destroy;
begin
  ResumeEvent.Free;
  inherited;
end;

procedure TSuspendableThread.Resume2;
begin
  ResumeEvent.SetEvent;
end;

procedure TSuspendableThread.SetSuspended2(const Value: Boolean);
begin
  if Value <> FSuspended2 then begin
    if Value then Suspend2 else Resume2;
    FSuspended2 := Value;
  end;
end;

end.
```

Hinweis: Die Klasse *TSuspendableThread* überschreibt absichtlich nicht die geerbten Methoden von *TThread*, denn das geerbte *Resume* wird noch benötigt, um einen Thread überhaupt zu starten, wenn er nicht automatisch im Konstruktor gestartet wird.

Verwendung von TSuspendableThread *R144*

Die Verwendung von *TSuspendableThread* ist nun ganz einfach. Sie wird im Folgenden anhand des Beispielprogramms *SyncDemo* gezeigt, die getroffenen Maßnahmen lassen sich aber auch auf beliebige andere Threads übertragen. Zunächst wird die Thread-Klasse nicht mehr von *TThread*, sondern von *TSuspendableThread* abgeleitet:

```
type
  TSyncThread = class(TSuspendableThread)
    ...
```

Die weitere Klassendeklaration bleibt unverändert, da alles Notwendige bereits von *TSuspendableThread* geerbt wird.

Dann wird mindestens eine wiederholt durchlaufene Programmposition gesucht, in der der Thread keine wichtigen Systemressourcen belegt. An dieser Stelle wird die geerbte Methode *CheckSuspendRequest* aufgerufen. In *SyncDemo* ist dies z.B. am Anfang der ständig durchlaufenen Schleife, unmittelbar vor der Reservierung der entscheidenden Systemressource:

```
procedure TSyncThread.Richtig;
var
  CollisionDetected: Boolean;
  i: integer;
begin
  repeat
    CheckSuspendRequest;
    Data^.CritSec.Enter;
```

Was bleibt, ist natürlich noch der Aufruf der neuen Methoden *Suspend2* und *Resume2*. In *SyncView* werden diese aufgerufen, wenn der Benutzer die entsprechenden Schalter des Formulars drückt:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  T[ThreadSelection.ItemIndex].Suspend2;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  T[ThreadSelection.ItemIndex].Resume2;
end;
```

TThreadList

Der Schlusshinweis zum Thema der Synchronisierung ist der Klasse *TThreadList* gewidmet. Mit ihr können Sie aus mehreren Threads auf eine *TList*-ähnliche Liste zugreifen. *TThreadList* ist allerdings nicht von *TList* abgeleitet, sondern *enthält* ein *TList*-Objekt, auf das Sie auch direkt zugreifen können – wobei wieder eine *Lock*-Methode ins Spiel kommt: *TThreadList.LockList*. *TThreadList* ist so einfach zu bedienen, dass an dieser Stelle ein Verweis auf die Online-Hilfe sowie auf das Kapitel über *TList* genügen soll.

TThreadList ist eine Liste *für* Threads, nicht zu verwechseln mit der Liste *bestehend aus* Threads, die im Beispielprogramm des nächsten Kapitels vorkommen wird.

4.7.4 Ein Utility mit dynamischer Thread-Anzahl

Dieser Abschnitt zeigt am Beispiel der Anwendung *WaveProcessingThreads*, wie Sie eine sich verändernde Zahl von Threads mit Hilfe eines einfachen *TList*-Objekts verwalten können. Aus Platzgründen können die anderen Aspekte dieser Anwendung hier jedoch nur kurz besprochen werden, zur genaueren Information sei daher auf den Quelltext auf der CD verwiesen.

Als Beispiel für eine rechenintensive Aufgabe dient hier die Bearbeitung von Klangdateien im WAV-Format, wobei die genaue Art der Bearbeitung nicht von Bedeutung ist (das Beispielprogramm kann nur eine Art Chorus-Effekt zum Klang hinzufügen).

Damit die Threads auch längere Zeit zum Rechnen haben, sollten Sie große WAV-Dateien verwenden. Für alle Fälle enthält das Programm auch eine Funktion, um sehr große Klangdateien selbst zu »synthetisieren« (falls es erlaubt ist, bei der Berechnung einer einfachen Sinusschwingung steigender Frequenz schon von »synthetisieren« zu sprechen).

Die Beispielanwendung (Abbildung 4.16) demonstriert nebenbei auch einige andere interessante Möglichkeiten von 32-Bit-Windows:

- die Verwendung mehrerer riesiger Speicherblöcke
- die direkte Ausgabe von voll programm-synthetisierten Sounddaten über das Low-Level-Multimedia-Interface ohne den Umweg über WAV-Dateien
- die grafische Darstellung der Wellenform einer WAV-Datei

Aus Platzgründen können diese Funktionen hier jedoch nicht detailliert besprochen werden.

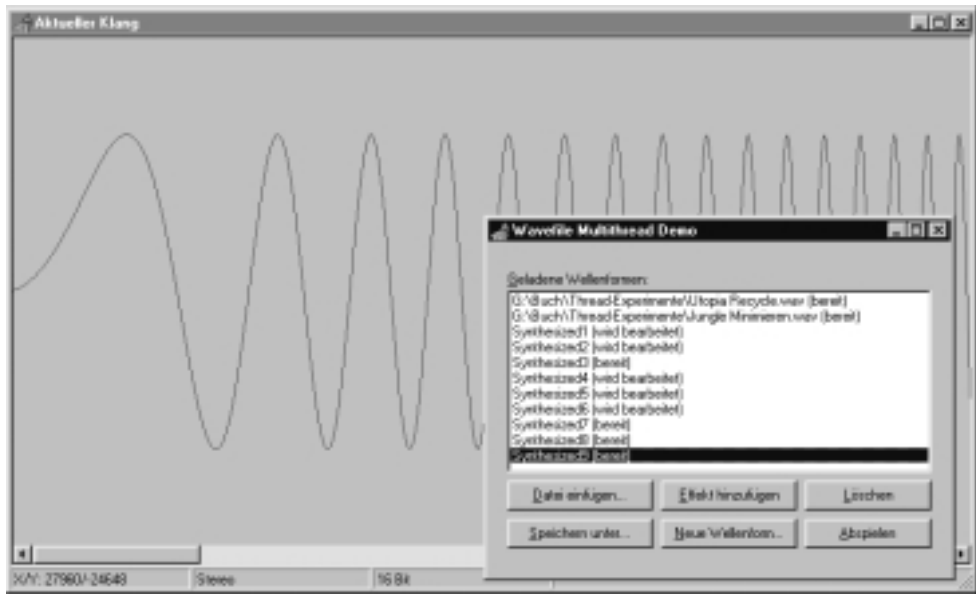


Abbildung 4.16: Die Beispielanwendung WaveProcessingThreads mit einer Liste von Klängen, von denen einige gerade bearbeitet werden

Bedienung

WaveProcessingThreads erlaubt es Ihnen, mehrere WAV-Dateien zu laden, deren Namen dann in einer Liste angezeigt werden (Abbildung 4.16). Die Anwendung führt jede län-

gere Bearbeitung einer dieser Dateien in einem eigenen Thread durch. Wenn Sie z.B. den Schalter EFFEKT HINZUFÜGEN drücken, startet die Anwendung einen neuen Thread, der die gewählten WAV-Daten mit einem Effekt versieht. Wie ebenfalls in der Abbildung zu erkennen ist, zeigt es die Anwendung in der Liste an, wenn eine Datei gerade von einem Thread bearbeitet wird.

Während dieser Bearbeitung können Sie einen anderen Klang aus der Liste wählen und auch diesen bearbeiten lassen, oder Sie können neue Klänge erzeugen (Schalter NEUE WELLENFORM...). Es ist natürlich nicht möglich, eine einzige Wellenform von mehreren Threads gleichzeitig bearbeiten zu lassen.

Implementierung der Liste von Klangdaten und Threads *R143*

Wie alle anderen Objekte können Sie natürlich auch *TThread*-Objekte in einer Liste der Klasse *TList* speichern. Während der Laufzeit der Anwendung könnten Sie dieser Liste je nach Bedarf neue Threads hinzufügen oder nicht benötigte Threads löschen. In einer Anwendung wie *WaveProcessingThreads* empfehlen sich jedoch nicht die Threads als Grundelement der Liste, sondern die Daten, mit denen diese Threads arbeiten, in diesem Fall also die Klangdaten.

Hinweis: Das Beispielprogramm benötigt keine der am Ende von Kapitel 4.7.3 erwähnten *ThreadLists*, da sein *TList*-Objekt nur von einem einzigen Thread, dem Haupt-Thread, angesprochen wird. Eine Synchronisation zwischen den verschiedenen Threads (der Daseinszweck von *TThreadList*) ist also nicht notwendig.

WaveProcessingThreads definiert daher für diese Klangdaten eine Struktur namens *TSound*. Jeder einzelne Sound enthält eine eigene *TThread*-Variable. Solange der Klang nicht bearbeitet wird, ist diese Variable auf *nil* gesetzt.

```
type
  TSound = class
    Thread: TSoundProcessThread;
    // Die eigentlichen Klangdaten:
    Data: TBigArray; // Sample-Daten, speichert auch
                     // Auflösung und Zahl der Kanäle
    SampleRate: Integer;
    FromFile: Boolean; // True -> aus Datei geladen
                       // False -> synthetisierter Klang
    // falls aus einer Datei geladen:
    FileName: string;
    Header: TWavHeader; // Header einer evtl. geladenen Datei
    // Benachrichtigung des Formulars, falls sich der
    // Thread-Status ändert (der Thread beendet wird):
    FormNotifyProc: TNotifyEvent;

    // Es gibt zwei Konstruktoren. Die Klangdaten können...
```

```
    // ... aus einer Datei geladen werden
    constructor Create(FileName: string);
    // ...oder synthetisiert werden (F kann z.B. die Sinus-Funktion sein):
    constructor CreateFromFunction(F: TSynthFunction; Rate, Size: Integer;
                                   Bytes, Channels: Byte);
    procedure ProcessSound(NotifyProc: TNotifyEvent);
    procedure ThreadTerminateNotify(Sender: TObject);

    // Hilfsfunktionen:
    procedure Play; // Ausgeben über das Multimedia-API
    procedure SaveAs(FN: string); // in Datei speichern
    procedure FillHeaderStruct;  // Datei-Header erstellen, falls
                                 // Klangdaten neu synthetisiert wurden
  end;
```

Mehrere solcher Klangobjekte werden nun in der Liste *Sounds* gespeichert, die in Form einer Formularvariablen vorliegt:

```
type
  TForm1 = class(TForm)
    ...
  public
    Sounds: TList;
```

Für die Erweiterung der Liste ist die bekannte *TList*-Methode *Add* zuständig. Bei jedem Hinzufügen eines neuen Klangobjekts wird außerdem die ListBox neu ausgegeben (zur Methode *UpdateList* siehe CD) und das neue Klangobjekt in dieser ListBox selektiert. Diese drei Aktionen sind in der Methode *AddSound* zusammengefasst:

```
procedure TForm1.AddSound(NewAuftrag: TSound);
begin
  Sounds.Add(NewAuftrag);
  UpdateList;
  ListBox.ItemIndex := ListBox.Items.Count-1;
end;
```

Der Schalter DATEI EINFÜGEN... ist mit der folgenden Methode verknüpft, die *AddSound* aufruft:

```
procedure TForm1.OpenBtnClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    AddSound(TSound.Create(OpenDialog.FileName));
end;
```

Auch der Schalter NEUE WELLENFORM... ruft *AddSound* auf, blendet vorher aber eine Dialogbox ein, die in einem weiteren Formular definiert ist, und verwendet einen zweiten Konstruktor der Klasse *TSound*. All diese Details sind jedoch unabhängig von der Verwaltung der Threads, weswegen hier nicht mehr weiter darauf eingegangen werden kann.

Starten der Threads

WaveProcessingThreads startet einen Thread, wenn Sie mit NEUE WELLENFORM... ein neues Klangobjekt erzeugen oder ein bestehendes mit EFFEKT HINZUFÜGEN verändern. Als Beispiel für die Thread-Verwaltung soll hier der letztgenannte Schalter dienen. Sein *OnClick*-Ereignis ist mit der folgenden Methode verknüpft, die das in der ListBox gewählte *TSound*-Objekt feststellt und dessen *ProcessSound*-Methode aufruft:

```
procedure TForm1.BearbeitenClick(Sender: TObject);
var
  SelectedSound: TSound;
begin
  if ListBox.ItemIndex <> -1 then begin
    SelectedSound := TSound(Sounds[ListBox.ItemIndex]);
    SelectedSound.ProcessSound(ThreadTerminateNotify);
    UpdateList;
  end;
end;
```

Da die Thread-Variable Teil der Klasse *TSound* ist, übernimmt mit *ProcessSound* auch eine Methode dieser Klasse die gesamte Verwaltung des Threads. Falls sie aufgerufen wird, obwohl bereits ein Thread aktiv ist, tut sie nichts, außer eine entsprechende Fehlermeldung anzuzeigen. Andernfalls erzeugt sie einen neuen *TSoundProcessThread* (die Parameter des *Create*-Konstruktors spielen hier keine Rolle) und startet ihn mit der Methode *Resume*. Außerdem verknüpft sie eine andere Methode des *TSound*-Objekts mit dem *OnTerminate*-Ereignis des Threads. Der Parameter *NotifyProc* wird später erläutert.

```
procedure TSound.ProcessSound(NotifyProc: TNotifyEvent);
begin
  if not Assigned(Thread) then begin
    FormNotifyProc := NotifyProc;
    Thread := TSoundProcessThread.Create(Data, Header, Effect);
    Thread.OnTerminate := ThreadTerminateNotify;
    Thread.Resume;
  end else
    MessageDlg('Alter Thread arbeitet noch. Kein neuer Thread wurde'
               +'erstellt.', mtInformation, [mbCancel], 0);
end;
```

Die Bedeutung des OnTerminate-Ereignisses R145

Warum es erforderlich ist, das *OnTerminate*-Ereignis zu verwenden, können Sie aus der zuletzt gezeigten Methode *ProcessSound* erkennen. Sie stellt sicher, dass immer maximal ein Thread für ein bestimmtes Soundobjekt aktiv ist, indem sie überprüft, ob der Variable *Thread* bereits ein Objekt zugewiesen (*assigned*) ist. Dazu ist es erforderlich, dass *Thread* immer dann auf *nil* gesetzt wird, wenn kein Thread mehr arbeitet.

Wenn der Thread endet, wird *Thread* jedoch nicht automatisch auf *nil* gesetzt. Das Einzige, was beim Ende des Threads automatisch geschehen *kann*, ist, dass das Thread-Objekt gelöscht wird (falls Sie das *TThread*-Property *FreeOnTerminate* auf *True* gesetzt haben).

Das Beispielprogramm muss die Variable *Thread* also selbst auf *nil* setzen, sobald der Thread beendet ist. Es könnte theoretisch mit *Thread.WaitFor* auf diese Beendigung warten, jedoch würde *WaitFor* den Haupt-Thread blockieren und den gesamten Sinn des Threads zunichte machen. Aus diesem Grund stellt *TThread* das Ereignis *OnTerminate* zur Verfügung, das aufgerufen wird, sobald der Thread endet.

TSound verknüpft das Ereignis mit der folgenden Methode, die das aktuelle *Thread*-Objekt freigibt, die Variable auf *nil* setzt und außerdem das Formular vom Ende des Threads benachrichtigt:

```
procedure TSound.ThreadTerminateNotify(Sender: TObject);
begin
  Thread.Free;
  Thread := nil;
  FormNotifyProc(Sender);
end;
```

Dabei ist *FormNotifyProc* eine Methode des Formulars, die dieses bei der Erzeugung des Threads im Parameter *NotifyProc* der Methode *TSound.ProcessSound* angegeben hat. Das Formular möchte nämlich auch vom Ende des Threads informiert werden, da es bei diesem Anlass die ListBox neu ausgeben muss, die ja auch Auskunft über aktive Threads gibt. In der oben gezeigten Methode *BearbeitenClick* wird die folgende Methode *ThreadTerminateNotify* zur Benachrichtigung des Formulars angegeben:

```
procedure TForm1.ThreadTerminateNotify(Sender: TObject);
begin
  UpdateList;
  WavForm.Invalidate; // Fenster mit der Grafikdarstellung aktualisieren
end;
```

Wenn also einer der Threads endet, erzeugt die VCL ein *OnTerminate*-Ereignis und ruft damit *TSound.ThreadTerminateNotify* auf. Diese Methode aktualisiert das *TSound*-Objekt und ruft *TForm1.ThreadTerminateNotify* auf, die schließlich das Formular aktualisiert.

Und auf der CD finden Sie ...

... die größeren Teile des Quelltextes, die hier nicht mehr besprochen werden können. Dazu gehört das Innere der Threads, also z.B. das Berechnen des Klangeffektes (einer Art Chorus-Effekt, der mit Hilfe eines 1000 Samples fassenden Puffers realisiert ist), die Ausgabe des Klanges über die API-Funktion *waveOutWrite*, die Teil des Low-Level-Multimedia-Interfaces von Windows ist und einen Datenblock direkt an die Soundkarte sendet, sowie die Ausgabe der Wellenform in einem Grafikfenster.

5 Die selbstständige Delphi-Anwendung

Aufbauend auf den Grundlagen der Kapitel 3 und 4 behandelt dieses Kapitel weitere wichtige Themen wie z.B. MDI-Anwendungen, Docking von Symbolleisten und Hilfsfenstern, virtuelle Koordinatensysteme, Drucken, Druckvorschau, Drag&Drop und XML. Der Name des Kapitels deutet bereits an, dass es hier eher um das geht, was eine Delphi-Anwendung alleine tun kann – ohne mit anderen Anwendungen, Bibliotheken oder Rechnern zusammenzuarbeiten. Diese »kooperativen« Themen werden in Kapitel 8 ausführlich zur Sprache kommen.

Die meisten Abschnitte dieses Kapitels zeigen Programmausschnitte aus einer größeren Beispielanwendung, dem *TreeDesigner 3.5*. Allerdings sollten Sie nicht erwarten, dass dieses Kapitel Ihnen den TreeDesigner vollständig erklären kann. Schon alleine das komplette Listing seiner Object-Pascal-Dateien würde mehr als 150 Seiten in Anspruch nehmen. Sie sollten den TreeDesigner daher eher als eine besondere Attraktion der CD-ROM ansehen denn als ein (Lern-)Ergebnis dieses Kapitels. Dieses Kapitel orientiert sich an wichtigen Themen und verwendet den TreeDesigner, um eine von vielen Möglichkeiten der praktischen Umsetzung zu zeigen. Damit die wesentlichen Aspekte nicht untergehen, sind die Listings gegenüber dem Original-Quelltext auf der CD-ROM teilweise gekürzt worden. Die Fokussierung auf die wesentlichen Aspekte soll es Ihnen auch erleichtern, diese in eigene Anwendungen zu übernehmen.

Wenn Ihnen dies jedoch nicht ausreichen sollte, sondern Sie die Arbeitsweise speziell des TreeDesigners im Detail durchschauen wollen, so sind die Listings dieses Kapitels nur die erste Stufe zu diesem Verständnis. Als zweite Stufe bietet es sich an, den TreeDesigner in die Delphi-IDE zu laden, in den Sie interessierenden Methoden die Erweiterungen gegenüber den Buch-Listings zu sichten und mit dem Debugger den Ablauf des Programms zu verfolgen.

Die Abschnitte dieses Kapitels müssen nicht in der vorgegebenen Reihenfolge durchgelesen werden. Allerdings kann es sehr hilfreich sein, wenn Sie sich zuerst mit dem groben Aufbau des TreeDesigners und dessen Bedienung vertraut machen. Hierfür bietet sich das Kapitel 5.1.3 zur Lektüre an.

5.1 Der TreeDesigner

Der charakteristischste Unterschied zwischen der Beispielanwendung dieses Kapitels, dem TreeDesigner, und den anderen Beispielprogrammen des Buches besteht darin, dass der TreeDesigner ein Dokument bearbeitet, das Sie in verschiedenen Ansichten am Bildschirm darstellen, ausdrucken und in Dateien speichern können, während die dialogorientierten Applikationen der anderen Kapitel eher Utility-Funktionen haben oder »fremde« Daten bearbeiten, die Sie nicht selbst speichern müssen (dies gilt vor allem für die Datenbankanwendungen). Diese beiden Anwendungstypen zeigen auch in der Entwicklung Unterschiede.

Entwicklung dialogorientierter Anwendungen

Mit Delphis visuellen Werkzeugen lassen sich komplexe Benutzerschnittstellen mit vielen Dialogboxen und Eingabeformularen sehr leicht entwerfen. Für bestimmte Anwendungstypen genügen die in den bisherigen Kapiteln besprochenen Grundlagen bereits – so z.B. für Programme, die eine einfache Aufgabe zu erfüllen haben, die der Benutzer in einem komfortablen Dialog genauer beschreiben kann. Auch Datenbankanwendungen sind meistens stark dialogorientiert (jedoch benötigen Sie für diese auch die in Kapitel 7 beschriebenen Grundlagen).

Entwicklung dokumentbasierter Anwendungen

Der andere Anwendungstyp soll an dieser Stelle als *dokumentbasiert* bezeichnet werden. Allein schon das Dokument bringt einige Aufgaben mit sich, die in Delphi nicht visuell gelöst werden können:

- Um die im Dokument enthaltenen Daten effizient verwalten zu können, benötigt das Programm entsprechende Datenstrukturen. Bei einfachen Texteditoren können diese aus einer Stringliste bestehen, wie sie von *TMemo* zurückgeliefert wird, die meisten Anwendungen müssen jedoch ihre eigene Datenstruktur definieren.
- Das Speichern und Laden der Daten von und zur Datei wird zwar meistens durch ein im Objektinspektor stehendes Event veranlasst, muss aber ansonsten ohne Komponentenhilfe auskommen (zumindest bieten die Komponenten, die sich in Delphis Lieferumfang befinden, noch keine Unterstützung dafür).
- Die Darstellung des Dokuments ist zwar für den Benutzer der Anwendung visuell, kann aber nicht zur Entwurfszeit festgelegt werden, sofern Sie die Grafikausgabemethoden von *TCanvas* aufrufen und keine der vorgefertigten (Grafik-)Komponenten benutzen.

Darüber hinaus enthält die Benutzerschnittstelle einer dokumentbasierten Anwendung oft Merkmale, die für ein dialoggesteuertes Tool teilweise gar nicht benötigt werden:

- Viele Anwendungen erlauben dem Benutzer, mehrere Dokumente gleichzeitig zu bearbeiten, und brauchen entsprechend eine geeignete Schnittstelle zur Handhabung mehrerer Dokumente, wie etwa das standardisierte Multi Document Interface (MDI). Mit Delphi ist es besonders einfach, aus einer gut entworfenen SDI-Anwendung (Single Document Interface) eine MDI-Anwendung zu machen.
- Bei der Bearbeitung des Dokuments verarbeitet die Anwendung die Eingaben des Benutzers direkter als eine dialogorientierte Anwendung. Während letztere relativ komplexe Ereignisse wie *OnClick* bearbeitet, muss eine dokumentbasierte Anwendung die zwei oder drei Bestandteile des *OnClick*-Ereignisses oft einzeln betrachten: »Taste drücken«, optional »Maus bewegen« und abschließend »Taste loslassen«. Ein weiteres Beispiel ist die Tastatureingabe: In dialogorientierten Anwendungen findet sie oft automatisch in Editierfeldern statt, so dass Sie nur noch das Ergebnis abfragen müssen, z.B. über *Edit1.Text*. Bei der Dokumentbearbeitung müssen Sie die Tastaturereignisse vielleicht selbst bearbeiten.

5.1.1 Wahl eines Beispielprogramms

Für unser Beispielprogramm benötigen wir als Erstes eine Aufgabe, in diesem Fall also den Typ des zu bearbeitenden Dokuments. Bei genauerer Betrachtung der elementaren Anwendungstypen Textverarbeitung, Tabellenkalkulation und Grafikprogramm stellt sich schnell heraus, welcher für unsere Beispielanwendung der geeignete ist:

- Textverarbeitungsprogramme, die mehr können als die *Memo*-Komponente, erfordern so viel Aufwand für die Datenverwaltung, dass sie eher in Bücher über allgemeine Programmiertechnik gehören.
- Für Tabellenkalkulationsprogramme gilt das Gleiche, sobald neben nackten Zahlen auch Ausdrücke in den Zellen stehen sollen, die geparst werden wollen.
- Grafikprogramme erlauben auch ohne komplizierte Algorithmen bereits den Aufbau von sinnvollen Dokumenten, wie in Abbildung 5.1 gezeigt.

Unser Beispielprogramm wird also ein Grafikprogramm, das die Grafik aus einzelnen, frei skalierbaren Objekten wie Rechtecken, Kreisen, Ellipsen, Linien und anderen Formen wie Rauten und Sechsecken, die jeweils durch einen zentrierten Text beschriftet werden können, zusammensetzt. Die zweite grundlegende Möglichkeit eines Grafikprogramms wäre eine Anwendung, die mit Bitmaps arbeitet und die Manipulation jedes einzelnen Pixels erlaubt. Ein solches Malprogramm finden Sie bereits in der Borland-Dokumentation (`Demos\Doc\GraphEx.dpr`), allerdings hat dieses wenig mit der Dokumentverwaltung zu tun, da es fast die gesamte Arbeit von den VCL-Komponen-

ten *TImage* und *TBitmap* abgenommen bekommt. Außerdem kann das Malprogramm verschiedene Windows-Bereiche wie Skalierung am Bildschirm und WYSIWYG-Darstellung nicht demonstrieren.

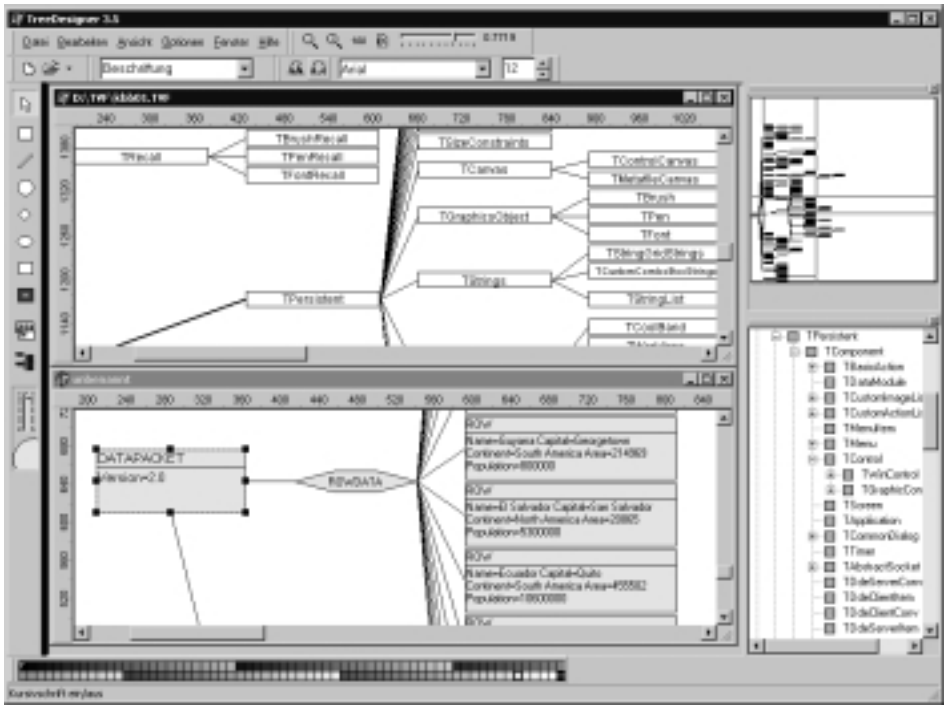


Abbildung 5.1: Der TreeDesigner mit zwei geöffneten Dokumenten und am rechten Rand angedockten Hilfsfenstern (im unteren Fenster die grafische Darstellung eines XML-Dokuments – der mit Delphi Professional/Enterprise mitgelieferten Datenbanktabelle Country.xml)

5.1.2 Spezielle Fähigkeiten

Da unsere Anwendung als allgemeine Grafikanwendung kaum mit professionellen Paketen konkurrieren kann, werden einige spezielle Funktionen eingebaut, die ihre Benutzung dennoch lohnend machen. So können Sie zwischen den grafischen Objekten Verbindungen definieren, die vom Programm automatisch durch Verbindungslinien dargestellt werden. Gegenüber handgezeichneten Verbindungslinien gibt es mehrere Vorteile:

- Wenn Sie ein Objekt verschieben, wird die Verbindung automatisch angepasst,
- oder alle verbundenen Objekte werden ebenfalls verschoben.
- Eine so verbundene Struktur kann vom Programm als Baum interpretiert werden,

- der mit etwas zusätzlichem Code sogar automatisch erzeugt werden kann.

Abbildung 5.1 zeigt auch den Grafikelemente-*TreeView*, der alle Grafikelemente des aktuellen Dokuments hierarchisch darstellt und Ihnen die Navigation innerhalb einer größeren Grafik erleichtert, ähnlich wie der CodeExplorer der Delphi-IDE es für die Pascal-Quelltexte vormacht.

Anzeige von Klassenhierarchien

Insbesondere die Fähigkeit der automatischen Baumerzeugung ist für Delphi-Programmierer interessant, da durch sie eine Alternative zu der Hierarchiedarstellung des in Delphi integrierten Browsers entsteht: Während die Wurzel des Hierarchiebaumes in diesem ganz oben steht und alle weiteren Klassen nur untereinander (mit etwas Einrückung versehen) angeordnet sind, lässt sich die grafische Baumdarstellung innerhalb des Grafikprogramms vielfältig ändern, zurechtrücken, farbig markieren und auf diese Weise übersichtlicher gestalten und sogar ausdrucken.

Lieferant für die Daten der Klassenhierarchie ist das auf der CD mitgelieferte Programm *Itéa*. Um die Klassenhierarchie-Grafik für ein bestimmtes Delphi-Projekt aufzubauen, führen Sie grob gesehen die folgenden Schritte aus:

- Starten Sie Itéa und lesen Sie dort das Projekt ein, dessen Klassenhierarchie Sie sehen wollen. Genaueres hierzu lesen Sie bitte in Kapitel 5.1.5 nach.
- Wählen Sie in Itéa den Menüpunkt BROWSER | KLASSENHIERARCHIE -> ZWISCHENABLAGE und anschließend im TreeDesigner den Menüpunkt BEARBEITEN | KLASSENHIERARCHIE EINFÜGEN oder die entsprechenden Aktionsschalter.

Auf diese Weise werden die Klasseninformationen über die Zwischenablage an den TreeDesigner übertragen, der für jede Klasse ein Grafikobjekt erzeugt. Nach der Übertragung ordnet der TreeDesigner die Grafik automatisch zu einem Baum an.

Den letzten Schritt können Sie jederzeit per Hand wiederholen, indem Sie den gewünschten Wurzelknoten markieren (z.B. über einen Doppelklick auf den Knotennamen im TreeExplorer) und dann den Anordnen-Befehl ausführen. Wenn Sie sich für einen anderen Knoten als *TObject* entscheiden, erhalten Sie einen möglicherweise sehr übersichtlichen Teilbaum aus der Klassenhierarchie. (Die VCL besteht aus erheblich mehr Klassen, als gleichzeitig in ein TreeDesigner-Dokument passen würden. Wenn Sie viele VCL-Units in Itéa eingelesen haben, erhalten Sie im TreeDesigner zunächst eine nicht lesbare Hierarchiedarstellung. Um dies zu ändern, müssen Sie sich im TreeDesigner auf einen Teilbaum unterhalb von *TObject* beschränken oder die im TreeDesigner integrierten Funktionen zum Ausblenden von Teilbäumen verwenden.) Natürlich können Sie auf dieselbe Weise auch von der VCL unabhängige, selbst erstellte Baumstrukturen anordnen.

Hinweise: Die CD-ROM enthält bereits fertige TreeDesigner-TVF-Dateien mit der Grafiken der VCL-Hierarchie. Diese können Sie im TreeDesigner mit DATEI | ÖFFNEN... ohne Zuhilfenahme von Itéa laden.

5.1.3 Kurzbeschreibung und Bedienung

Wie bereits angedeutet, handelt es sich beim TreeDesigner um ein Grafikprogramm, in dem Sie mit einfachen grafischen Formen wie Linien, Rechtecke, Ellipsen und mit spezielleren Formen arbeiten, die jeweils mit einem zentrierten Text beschriftet werden können. Zwischen einzelnen Objekten können Sie Verbindungen definieren, die automatisch gezeichnet werden. Die Arbeitsfläche des TreeDesigners ist ein Zeichenblatt, das per Voreinstellung eine quadratische Form hat, die mit 3000*3000 Pixeln weit über den Bildschirm hinausgeht und daher innerhalb des Fensters gescrollt werden kann.

Falls Sie eines der Vorgängerbücher mit einer älteren TreeDesigner-Version besitzen, werden Sie sich möglicherweise für die Unterschiede zwischen den Versionen interessieren:

- Die auffälligste Erweiterung der Version 3.5 ist die Aufbereitung einer XML-Datei in einer grafischen Baumstruktur (siehe Abbildung 5.2). Dafür war es notwendig, einen neuen Typ von Grafikelement einzuführen, der auch mehrzeiligen Text mit automatischem Zeilenumbruch darstellen kann. Die Grafikelemente nutzen nun die Polymorphie, damit verschiedene Elemente verschiedener Klassen in Dateien gespeichert werden können. Damit kann nun auch das Verlaufselement (ein Rechteck, das mit einem Farbverlauf gefüllt wird) in Dateien gespeichert werden. Eine zweite XML-Erweiterung des TreeDesigners besteht darin, dass dieser auch seine eigenen Daten (bisher in binären `.tvf`-Dateien abgelegt) in XML-Dateien schreiben kann (in den Grafikelementen angezeigte Bilder werden *nicht* in der XML-Datei gespeichert).
- Die Version 3.0 brachte gegenüber der Version 2.5 folgende Erweiterungen: Ein Hilfsfenster gibt eine Übersicht über das aktuelle Dokument und zeigt, welcher Ausschnitt gerade im Dokumentfenster sichtbar ist. Durch das Dokument-View-Konzept erlaubt der TreeDesigner seit dieser Version die Bearbeitung eines Dokuments in mehreren gleichzeitig geöffneten Dokumentfenstern. Außerdem wurden die Lineale optimiert, die Handhabung des TreeExplorers verbessert, einfache Einstellungsdialoge für die einzelnen Grafikobjekte hinzugefügt und weitere Verbesserungen eingebaut.
- Neu in der Version 2.5 waren unter anderem der TreeView mit allen Grafikelementen des aktuellen Dokuments, die Möglichkeit, neben Text auch Grafikdateien (Bitmaps oder Metadateien) in einzelne Grafikelemente einzublenden, und die Griffe, mit denen Sie Grafikobjekte wie vom Formularentwurf in der Delphi-IDE bekannt in der Größe ändern können.

Mausaktionen

Zum Zeichnen von neuen Objekten wählen Sie aus der Werkzeugleiste den Schalter mit der gewünschten grafischen Form aus und zeichnen das Objekt mit der Maus auf die Arbeitsfläche ein. Solange der entsprechende Schalter in der Werkzeugleiste gedrückt ist, können Sie nur neue Objekte einzeichnen, aber keine bestehenden Objekte verändern. Diesen Zeichenmodus können Sie auch daran erkennen, dass der Mauszeiger die Form eines Fadenkreuzes angenommen hat.

Wenn Sie in der Werkzeugleiste den Schalter mit dem Mauszeigersymbol wählen, werden keine neuen Objekte gezeichnet, sondern Sie können die bestehenden Objekte manipulieren:

- Zum Verschieben ziehen Sie das Objekt einfach mit der Maus an seinen neuen Ort.
- Zum Markieren klicken Sie es mit der Maus an, zusammen mit Shift können Sie mehrere Objekte auswählen.
- Um einen Bereich von Objekten zu markieren, zeichnen Sie mit der Maus ein Rechteck um die zu markierenden Objekte. Sie dürfen die Maus dazu jedoch nicht über einem Objekt drücken, da Sie dieses sonst verschieben.
- Um die Größe eines markierten Objekts zu verändern, gehen Sie so vor wie beim Verändern der Größe von Komponenten im Formulareditor der Delphi-IDE: Ziehen Sie an einem der »Griffe«, die im Rahmen des Objekts gezeichnet werden.
- Um Objekte miteinander zu verbinden, markieren Sie einige »Zielobjekte« und klicken, während Sie Strg halten, auf das Objekt, von dem die Linien zu den Zielobjekten ausgehen sollen (mehr über diese Funktion können Sie in Kapitel 5.3.4 lesen).
- Um ein Objekt zu verschieben, ohne dass der gesamte von diesem Objekt ausgehende Teilbaum ebenfalls verschoben wird, drücken Sie Shift, bevor Sie mit dem Verschieben beginnen.

Tastaturaktionen

Unabhängig von der Einstellung des Zeichenwerkzeugs können Sie zwei Tastaturbefehle einsetzen (sofern der Eingabefokus sich nicht in einem Eingabeelement der Symbolleisten befindet):

- Um Objekte zu löschen, markieren Sie sie und drücken Entf.
- Um das oberste Grafikelement hinter alle anderen Elemente zu verschieben, drücken Sie PgDn. Diese Funktion können Sie auch dazu verwenden, durch alle vorhandenen Grafikelemente zu blättern, denn der TreeDesigner rückt das neue oberste Element automatisch in den sichtbaren Fensterbereich.

Editieren der Attribute

Sie können bei jedem Objekt die folgenden Attribute verändern: Linien-, Füll- und Schriftfarbe, Schriftart und -größe, die Form des Objekts sowie den Beschriftungstext.

Alle Attribute können Sie über die Symbolleisten wie folgt einstellen: Markieren Sie zuerst die Objekte, die Sie verändern wollen, und stellen Sie dann die Attribute ein:

- Geben Sie einen Beschriftungstext in das Editierfeld ein.
- Wählen Sie eine Schriftart aus der aufklappbaren Liste aus.
- Geben Sie neben dieser Listbox die Größe der Schrift ein. Über die beiden zusätzlichen Schalter können Sie auch Fett- und Kursivschrift einstellen.
- Um die Dicke der Linien zu verändern, verschieben Sie den Regler am unteren Ende der vertikalen Werkzeugleiste.
- Um die Form der gewählten Objekte zu ändern, doppelklicken Sie auf den Schalter für die neue Form.

Für die Farben stellt der TreeDesigner eine Farbpalette zur Verfügung, in der drei Farben markiert sind (siehe Abbildung 5.2):

- Die mit einer Linie markierte Farbe wird zum Zeichnen der Linie verwendet (markieren Sie diese Farbe mit der linken Maustaste).
- Die Farbe, die mit einem Kreis versehen ist, findet sich zum Füllen des Objekts wieder (Markieren mit rechter Maustaste).
- Und die Farbe mit der »T« -Markierung liefert die Farbe für die Beschriftung (Markieren mit Shift +linker Maustaste).

Die Füllungsfarbe der Objekte können Sie auch per Drag&Drop ändern, indem Sie die gewünschte Farbe mit der linken Maustaste auf das Objekt ziehen (dabei sollten keine Objekte markiert sein, da sonst deren Linienfarbe geändert würde).

Die Koordinaten eines Elements können Sie auch dialoggesteuert einstellen, wenn Sie das Element mit der rechten Maustaste anklicken und den Popup-Menüpunkt ATTRIBUTE... ausführen (es werden dann nur die Attribute des angeklickten Elements geändert, auch wenn ein anderes Element markiert sein sollte).

Alle Einstellungen in den Symbolleisten werden auch zum Zeichnen eines neuen Objekts verwendet, mit Ausnahme des Beschriftungstextes, der nach einmaliger Verwendung wieder gelöscht wird (allerdings können Sie mehrere Objekte markieren und deren Beschriftung gleichzeitig verändern).

Unter dem Regler für die Linienstärke befindet sich außerdem ein kleines Vorschaufeld, in dem Sie aktuelle Hintergrundfarbe, Linienfarbe und Linienbreite erkennen können.

Standardaktionen

Der TreeDesigner verfügt natürlich über die üblichen von MDI-Anwendungen gewohnten Merkmale wie die Funktionen des Fenstermenüs, das Speichern und Laden von Dateien (das Format der vom TreeDesigner erzeugten Dateien wird durch die Endung `.tvf` gekennzeichnet) und über die typischen Zwischenablageoperationen BEARBEITEN | AUSSCHNEIDEN, BEARBEITEN | KOPIEREN und BEARBEITEN | EINFÜGEN. AUSSCHNEIDEN und KOPIEREN beziehen sich dabei jeweils auf alle im aktuellen Dokument markierten Objekte.

Auch für diese Zwischenablage-Operationen verwendet der TreeDesigner sein eigenes Format, das heißt, Sie können aus dem TreeDesigner kopierte Objekte nicht in anderen Anwendungen einfügen und umgekehrt keine Objekte aus anderen Anwendungen in den TreeDesigner übernehmen – zumindest nicht über dieses Menü. Innerhalb eines einzelnen TreeDesigner-Grafikelements können Sie jedoch wie folgt Grafiken aus anderen Quellen anzeigen.

Anzeige von Grafiken innerhalb der Objekte

Im Popup-Menü der einzelnen Grafikelemente finden Sie unter anderem drei Punkte, die sich mit dem Inhalt des Elements befassen:

- GRAFIK AUS DATEI LADEN... öffnet einen *TOpenPictureDialog* zur Auswahl einer Grafikdatei, deren Inhalt daraufhin im Grafikelement angezeigt wird.
- GRAFIK AUS ZWISCHENABLAGE EINFÜGEN fügt den Grafikinhalt der Zwischenablage in das aktuelle Grafikelement ein..
- INHALT LÖSCHEN stellt den Ursprungszustand des Grafikelements wieder her, eine eventuell mit den beiden vorher genannten Funktionen eingefügte Grafik wird also gelöscht.

Da zur Verwaltung der Grafik ein *TPicture*-Objekt verwendet wird, können Sie alle Grafikdateien laden, die mit *TPicture* kompatibel sind. Durch jedes Einfügen eines neuen Elementinhalts wird der bisherige Inhalt gelöscht. Die Grafik wird per *StretchDraw* gezeichnet und füllt die rechteckigen Umrisse eines Grafikelements immer voll aus, dabei findet auch bei nicht-rechteckigen Elementen kein Clipping statt. Beim Speichern des Dokuments wird die Grafik über *TPicture.SaveToStream* in die TVF-Datei geschrieben, wird also zu einem fest integrierten Bestandteil des TreeDesigner-Dokuments.

Diese Funktionen sind hauptsächlich zur Demonstration gedacht, wie durch relativ kleine Erweiterungen am Programmcode andere Inhalte innerhalb der einzelnen Grafikelemente angezeigt werden können. Die Implementierung dieser Funktionen kann hier aus Platzgründen nicht erläutert werden, aber sie wurde über eine eigene Unit namens *GrCont* mit der Absicht entworfen, leicht nachvollziehbar und auf andere Typen von Grafiken erweiterbar zu sein.

Hinweis: Da die neue Version des TreeDesigners auch die Speicherung abgeleiterer Grafikelement-Klassen unterstützt, bietet es sich unter Umständen an, statt des in der Unit *GrCont* vorgezeichneten Weges gleich eine eigene neue Klasse von *TGraphicElement* abzuleiten (mit Hilfe von *GrCont* ist es nur möglich, den Inhalt eines Elements einer bereits bestehenden Klasse zu definieren).

Ansichten und Drucken

Über den Regler in der Mauspalette können Sie die Vergrößerung, mit der die Grafik angezeigt wird, zwischen 0,1 und 10 einstellen. Dabei versucht der TreeDesigner, je nach Zoom-Modus den derzeitigen Mittelpunkt oder die markierten Objekte auch nach der Vergrößerung / Verkleinerung zentriert darzustellen.

Der Druckdialog lässt Ihnen die Wahl zwischen zwei verschiedenen Druck-Modi:

- Im Modus *1 Einheit = 0,1 mm* wird der gesamte Bereich des Arbeitsblatts gedruckt. Mit dem Menübefehl ANSICHT | SEITENANZEIGE können Sie das Arbeitsblatt auf die zurzeit im Druckertreiber gewählte Seitengröße »zurechtschneiden«. Um es ungefähr in der Größe des Drucks auf dem Bildschirm zu sehen, müssen Sie beispielsweise bei 17-Zoll-Monitoren den Vergrößerungsfaktor 0,2 einstellen.
- Im Modus *Druckgröße anpassen* können Sie den Ausdruck frei auf der Seite positionieren und skalieren. Siehe hierzu Kapitel 5.6.4.

Hinweis: Beachten Sie auch die Funktion BEARBEITEN | KOPIEREN, mit der Sie die aktuelle Grafik als Metadatei in andere Anwendungen exportieren können (siehe Kapitel 4.5.1).

Anpassen der Benutzeroberfläche

Zur Laufzeit haben Sie beim TreeDesigner einige Möglichkeiten, die Oberfläche des Programms anzupassen:

- Alle Symbolleisten unter der Menüzeile sowie die Farbpalette benutzen die Docking-Technik der VCL, sind also abtrennbar und beliebig am oberen oder unteren Fensterrand andockbar. Über das Popup-Menü der Symbolleisten können Sie

auswählen, welche Leisten sichtbar sind. Der TreeDesigner speichert die Konfiguration der Leisten zwischen den Programmläufen. Die Werkzeugleiste am linken Fensterrand ist fest verankert.

- Andockbar sind auch das Grafikelemente-TreeView und das Übersichtsfenster; einzig zugelassene Andockstelle ist allerdings der rechte Rand des Hauptfensters.
- Das Popup-Menü zu den Linealen hat nur einen Menüpunkt, mit dem Sie die Schriftart anpassen können, die in den Linealen verwendet wird.
- Im Popup-Menü des TrackBars wählen Sie zwischen zwei verschiedenen Modi für die Skalierung des Zoom-Reglers (siehe Kapitel 5.6.3) und zwischen drei zusätzlichen Zoom-Modi, die festlegen, ob beim Zoomen auf die Bildmitte, auf die Markierung oder überhaupt nicht fokussiert werden soll.
- Mit OPTIONEN | TASTENKÜRZEL... gelangen Sie in den Tastenkürzeleditor aus Kapitel 4.6.3, in dem Sie jedem der Hauptmenüpunkte ein anderes Tastenkürzel zuweisen können.

Eine Übersicht über die Dialogfenster des TreeDesigners finden Sie in Kapitel 5.1.4.

Der TreeExplorer

Der Grafikelemente-TreeView (im Code auch als TreeExplorer bezeichnet) listet alle Grafikelemente des aktuellen Dokuments anhand ihrer Beschriftung auf, wobei verbundene Elemente hierarchisch dargestellt werden. Er ist am rechten Rand des Hauptfensters andockbar. Sein Inhalt wird nur dann automatisch aktualisiert, wenn er aktiviert wird und seit seiner letzten Aktualisierung ein anderes Dokumentfenster aktiviert wurde. Durch einen Doppelklick in den TreeView-Bereich ist jedoch jederzeit eine manuelle Aktualisierung möglich.

Gegenüber der Version 2.5 des TreeDesigners wurde die Handhabung des TreeExplorers weiter auf die Bearbeitung von VCL / CLX-Hierarchiegrafiken optimiert:

- Gleich nach der Aktivierung des TreeExplorer-Fensters wird in diesem automatisch ein Knoten markiert, der im aktuellen Grafikdokument sichtbar ist. Dies ist wichtig, wenn viele Knoten in einer Grafik unsichtbar sind, weil sie einem ausgeblendeten Teilbaum angehören.
- Durch einen Doppelklick auf einen Baumknoten markieren Sie das entsprechende Grafikelement im Dokumentfenster – vorausgesetzt, dass dieses durch eine eindeutige Beschriftung identifizierbar ist.
- Bei einer normalen Auswahl (mit Einfachklick) eines TreeView-Elements wird das entsprechende Element (falls eindeutig beschriftet) im Zeichenfenster nur sichtbar gemacht, aber nicht markiert. Wenn Sie einen unsichtbaren Knoten wählen, erhalten Sie einen entsprechenden Hinweis in der Statuszeile.

- Umgekehrt können Sie ein Grafikelement im Zeichenfenster markieren und dieses automatisch im TreeView hervorheben lassen. Hierzu dient ein entsprechender Punkt im Popup-Menü des TreeExplorers.

Das Übersichtsfenster

Ein weiteres andockbares Hilfsfenster ist das Übersichtsfenster. Es bezieht sich ebenfalls immer auf das aktuelle Dokument und zeigt dieses in einer Gesamtansicht an (der Dokumentinhalt wird also auf die Größe dieses Fensters verkleinert). Hilfslinien in der Übersicht markieren außerdem den im Dokumentfenster sichtbaren Bereich. Das Übersichtsfenster wird immer automatisch aktualisiert (bei jeder Änderung im aktuellen Dokument, beim Wechsel zwischen Dokumentfenstern und beim Scrollen innerhalb dieser Fenster).

Dialoge

Weitere, eventuell noch nicht erwähnte Dialogfenster des TreeDesigners gehen aus der Formular- und Dateiübersicht des nachfolgenden Abschnitts hervor.

5.1.4 Dateien auf der CD

Auf der CD finden Sie zwei verschiedene Programmversionen des TreeDesigners. In der vereinfachten Version (Abbildung 5.2) können Sie immer nur ein Grafikdokument gleichzeitig bearbeiten, es sei denn, Sie starten den TreeDesigner mehrmals (was aber nur außerhalb der Delphi-IDE möglich ist). Wenn Sie also eine neue Grafik laden, ersetzt diese die bisherige Grafik im TreeDesigner-Fenster. Die Ein-Dokument-Version wird im Folgenden auch als SDI-Version (Single Document Interface) bezeichnet. Ihre Projektdatei finden Sie auf der CD-ROM unter dem Namen TreeDesignerSDI.

Die zweite Version (*TreeDesigner*, siehe Abbildung 5.1) ist eine MDI-Anwendung, in der Sie mehrere Grafiken in eigenen Fenstern bearbeiten können, welche sich alle innerhalb eines übergeordneten Hauptfensters befinden. Die Unterschiede zwischen SDI- und MDI-Version werden in Kapitel 5.7 behandelt, in allen anderen Kapiteln spielt es keine Rolle, ob Sie die MDI- oder die SDI-Version vor sich haben.

Formulare und Programmdateien

Der TreeDesigner beinhaltet in der MDI-Version zwölf Formulare, die zur Laufzeit als sichtbare Fenster auftreten, drei Formulare bzw. Frames, die durch Vererbung in einigen der zwölf sichtbaren Formulare erscheinen, sowie ein weiterer Frame, der die Werkzeugleiste am linken Rand des Hauptfensters definiert. Er verwendet sechs der auf der CD mitgelieferten Komponenten, von denen vier in Kapitel 6 entwickelt werden. Zur Laufzeit ständig sichtbar sind die auf den folgenden Formularen basierenden Fenster:

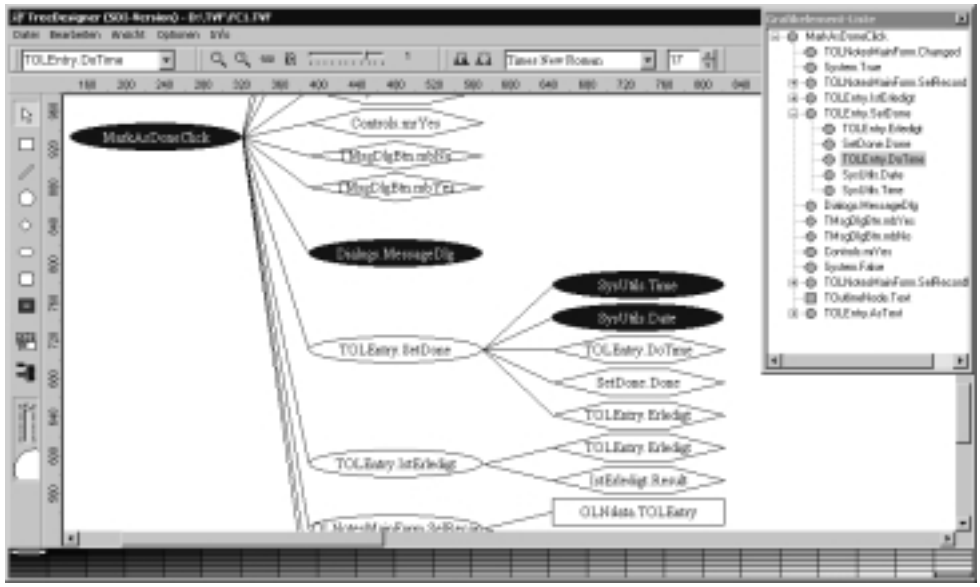


Abbildung 5.2: Die SDI-Version des TreeDesigners mit einer Grafik aus Itéa

- In der MDI-Version dient das Formular *MainForm* (Unit *MainForm*) als Hauptfenster. Es beinhaltet ein kleines Hauptmenü, das die von den einzelnen Dokumenten unabhängigen Menüpunkte enthält. Außerdem finden Sie in diesem Fenster die Werkzeugleiste, die Farbpalette, Platzhalter für die Symbolleisten der Kindfenster sowie die Andockstelle für den TreeExplorer.
- *TDocumentForm* (Unit *DocForm*) ist die Klasse für das Dokumentfenster. *MainForm* verwendet diese Klasse als Schablone für die MDI-Fenster; in der SDI-Version gibt es nur ein Exemplar dieser Klasse, das dann das Hauptfenster ist (Name: *DocumentForm*). Zur SDI-Version siehe auch Kapitel 5.7.7, *Verwenden des MDI-Kindfensters als SDI-Hauptformular.*
- *TreeExplorerForm* (Unit *TreeExplorer*) definiert den in Kapitel 5.1.3 beschriebenen, am rechten Rand des Hauptfensters andockbaren TreeExplorer.
- *OverviewForm* (Unit *Overview*) ist das zweite andockbare Hilfsfenster. Es zeigt immer eine Gesamtansicht des aktuellen Dokuments und wurde ebenfalls schon in Kapitel 5.1.3 erwähnt.

Hinweis: Aufgrund der in Kapitel 8.6.2 unter *Formular und Automationsklasse in einer Unit* beschriebenen Vorgänge finden Sie das Formular der Unit *DocForm* nicht in der Formularliste der IDE. Sie können es jedoch immer anzeigen, wenn Sie von der Unit *DocForm* aus zwischen Formular und Unit umschalten (F12 bzw. Symbolleiste und Menü ANSICHT)

Die sieben restlichen Formulare werden als modale Dialoge verwendet:

- *PrintDialog* ist der Druckdialog, den der TreeDesigner anstelle des Standard-Druckdialogs anzeigt. Der Dialog enthält eine Druckvorschau und verschiedene Optionen für die Druckgröße und -position.
- *DocListForm* wird über den Menüpunkt ANSICHT | LISTE DER GEÖFFNETEN DOKUMENTE aufgerufen und listet getreu seines Namens alle Dokumente auf, die in irgendeinem der Dokumentfenster angezeigt werden. Außer den Dateinamen gibt es in der Liste noch an, in wie vielen Fenstern das jeweilige Dokument angezeigt wird (eine eventuelle Anzeige des Dokuments im Übersichtsfenster wird mitgezählt) und ob der Inhalt seit dem letzten Speichern verändert wurde.
- *ObjectAttrDlg* (Unit *AttrDlg1*) stellt einen Dialog dar, in dem Sie die Koordinaten, die Beschriftung und die Form des aktuellen Grafikelements anpassen können. Für die genaue Einstellung der Koordinaten ist dieser Dialog nützlich, ansonsten dient er hauptsächlich als Beispiel für den Einsatz von Frames (siehe Kapitel 3.7).
- *InfoForm* dient zur Anzeige von Informationen zum TreeDesigner und wird wie üblich über das Hilfemenü aufgerufen.
- *ScaleDialog* ist ein modaler Dialog und wird durch den Menüpunkt BEARBEITEN | SKALIEREN aufgerufen. Er enthält nur zwei Editierfelder, in denen Sie angeben, um welchen Faktor die Grafik in Breite und Höhe skaliert werden soll.
- *TreeArrangeOptionsDialog* enthält einige Optionen für die automatische Baumanordnungsfunktion: Abstand zwischen den Hierarchieebenen, Breite und Höhe einer Ebene sowie Ausrichtung des Baumes (horizontal oder vertikal).
- *OutlineDemoForm* (Unit *OlDemo*) wurde aus Gründen der »Abwärtskompatibilität« aus dem TreeDesigner 2.0 übernommen und gibt unter Verwendung der *TOutline*-Komponente eine alternative Baumdarstellung der im TreeDesigner bearbeiteten Grafik.

Schließlich finden Sie in der Formularliste noch fünf weitere Mitglieder, die mit den in Kapitel 3.7 beschriebenen Vererbungs-Techniken zu tun haben:

- *FrameBasicObjectAttrs* definiert den Frame mit den Dialogelementen zur Einstellung von grundlegenden Objekteigenschaften.

- *ObjectAttrBaseForm* (Unit *AttrDlg0*) definiert die Basisklasse für die Dialoge zum Einstellen der Objekteigenschaften. Die einzigen darin bereits definierten Dialogelemente stammen aus dem Frame *FrameBasicObjectAttrs*. Die zur Laufzeit in Erscheinung tretenden Dialoge, die auf diesem Formular basieren, sind *ObjectAttrDlg* (siehe oben) und *VerlaufObjectAttrDlg*.
- *GradientObjectAttrDlg* (Unit *AttrDlg2*) definiert den Dialog zum Einstellen der Attribute der speziellen Farbverlaufs-Grafikobjekte. Zu weiteren Details bezüglich der Attributdialoge siehe Kapitel 3.7.
- *GlobalToolBar* ist das Frame, das als Werkzeugleiste an der linken Seite des Hauptfensters eingeblendet wird. Das Werkzeugleisten-Panel hätte natürlich ohne Frame auch direkt im Hauptfenster eingezeichnet werden können. Die Entscheidung für die Auslagerung dieses Panels in einen Frame hängt damit zusammen, dass der TreeDesigner sowohl in einer MDI- als auch in einer SDI-Version erscheinen sollte. Dies spielt aber im weiteren Verlauf dieses Kapitels keine Rolle.
- *DockableForm* stellt eine gemeinsame Basisklasse für die Formulare *OverviewForm* und *TreeExplorerForm* dar. Es definiert den in beiden Formularen benötigten Code sowie die allgemeinen Property-Einstellungen für das Docking dieser Formulare (siehe auch Kapitel 5.8.2).

Da die Erläuterung des TreeDesigners sich nicht über das ganze Buch ausdehnen kann und Themen wie Dialoge und TreeViews bereits in anderen Kapiteln erläutert wurden, werden sich die folgenden Abschnitte auf das Haupt- und Dokumentformular konzentrieren. In Kapitel 5.6.4 kommt außerdem der Drucken-Dialog zur Sprache, und das TreeExplorer-Fenster wird in Kapitel 5.8.2 noch einmal erscheinen, aber nur in seiner Eigenschaft als Docking-Fenster.

5.1.5 Kurzvorstellung von Itéa

Zur Bereitstellung der Klassenhierarchie in der Zwischenablage dient die auf der CD mitgelieferte Itéa-Entwicklungsumgebung. Itéa wurde zu den Zeiten von Delphi 1 und 2 unter anderem dazu entworfen, die jetzt von Delphis AppBrowser-IDE bekannten Vorteile in einer eigenständigen browserartigen Umgebung zur Verfügung zu stellen. Mit dem Erscheinen von Delphi 4 endete die Entwicklung von Itéa als eigenständige IDE vorerst. Itéa wurde nur so weit an die neuen Sprachmerkmale von Delphi 6 angepasst, dass damit die VCL-Quelltexte eingelesen und die Klassenhierarchie an den TreeDesigner gesendet werden können. Die Funktionen von Itéa stehen übrigens auch als DLL zur Verfügung. Die im Anhang beschriebenen IDE-Erweiterungen nutzen diese DLL dazu, Delphis AppBrowser-Fähigkeiten zu erweitern und teilweise sogar für die Personal/Standard-Ausgabe nachzubilden.

Eine Installationsanleitung und eine Dokumentation für Itéa finden Sie auf der CD. Die folgenden Abschnitte werden nur kurz die für die Zusammenarbeit zwischen TreeDesigner und Itéa erforderlichen Eigenschaften erläutern.

Laden von Projekten

Um ein Projekt in Itéa bearbeiten zu können, starten Sie Itéa und verwenden den Menüpunkt DATEI | PROJEKT ÖFFNEN, um ein Projekt zu laden, wobei Sie im Dateiauswahldialog die DPR-Datei auswählen.

Beim Laden eines Projekts benötigt Itéa für jede im Projekt verwendete Unit eine Datei, die zumindest den Quelltext des Interface-Teils dieser Unit enthält. Bei den Delphi-Versionen, bei denen nicht der gesamte VCL-Quelltext mitgeliefert wird, finden Sie zumindest ein DOC-Verzeichnis mit INT-Dateien, die die Interface-Teile aller Units enthalten. Damit Itéa ein Projekt vollständig einlesen kann, müssen die entsprechenden Verzeichnisse von Delphi in den Itéa-Optionen eingestellt sein. Normalerweise findet das Installationsprogramm diese Verzeichnisse selbst, fehlende Verzeichnisse können Sie sogar noch während des Ladens eines Projekts in Itéa ergänzen.

Nach dem Laden eines Projekts öffnet Itéa ein Browserfenster für das gesamte Projekt (Projektfenster), das unter anderem Listen der globalen Klassen, Typen, Variablen und Funktionen enthält. In diesem Fenster könnte man jetzt ein Modul aus der Liste *Alle Module* oder eine Klasse aus der Liste *Implementierte Klassen* auswählen und in einem neuen Browserfenster öffnen, um beispielsweise die Methoden der Klasse zu editieren.

Wenn Sie jedoch lediglich eine Klassenhierarchie in den TreeDesigner kopieren wollen, können Sie Itéa nach Ausführung des Menüpunktes BROWSER | KLASSENHIERARCHIE -> ZWISCHENABLAGE sofort wieder schließen. Sie können den Inhalt der Zwischenablage zur Sicherheit auch zuerst mit einem Texteditor in einer Textdatei speichern. So könnten Sie die Klassenhierarchie jederzeit neu in den TreeDesigner einfügen, ohne Itéa das ganze Projekt neu einlesen lassen zu müssen.

Die weitere Verarbeitung der Klassenhierarchie-Daten im TreeDesigner wurde bereits in Kapitel 5.1.2 beschrieben.

Hinweise: Itéa speichert die Klassenhierarchie im Format *CF_TEXT* in der Zwischenablage. Wenn vorher andere Texte in anderen Formaten in der Zwischenablage enthalten waren, kann es sein, dass Sie – zumindest unter Windows NT – den von Itéa kopierten Text nicht in alle Textprogramme einfügen können.

Weitere Grafiktypen

Neben den Klassenhierarchiegrafiken kenn Itéa noch drei weitere Arten von Grafiken, die ebenfalls im TreeDesigner bearbeitet werden können:

- *Bäume verwendeter Ids:* Sie zeigen alle Bezeichner, die in einer bestimmten Methode benutzt werden, also sowohl die Typen und Variablen, die darin verwendet, als auch die Methoden, die darin aufgerufen werden. (Dabei sind die Variablen, Typen und Methoden durch unterschiedliche Formen der Umrandung unterscheidbar).

  Für die aufgerufenen Methoden, die wieder andere Bezeichner verwenden, wird dieser Vorgang wiederholt, so dass eine tief geschachtelte Struktur wie in Abbildung 5.2 entsteht, wobei Sie die maximale Schachtelungstiefe festlegen können. Rekursiv aufgerufene Funktionen werden nicht mehrmals dargestellt, sondern mit einer doppelten Umrandung und dem Zusatz »(rec.)« versehen.

- Umgekehrt können Sie in einem *Baum abhängiger Ids* auch anzeigen, von wo aus eine bestimmte Methode verwendet wird. Diese Grafik ist gewissermaßen das Gegenteil der letztgenannten Grafik. Sie zeigt den Inhalt verschiedener Listen *Abhängige Ids* der Browserlisten.

- Schließlich haben Sie noch die Möglichkeit, *Datenstrukturen* als Bäume darstellen zu lassen. In diesen sind nicht nur die Variablen eines Records oder eines Objekts enthalten, sondern auch alle Variablen von Unterobjekten, von Unterunterobjekten usw.

Für die letztgenannten drei Baumtypen müssen Sie in Itéa zuerst das Programmobjekt wählen, für das der Baum angezeigt werden soll; im BROWSER-Menü zeigt Ihnen Itéa dann die verschiedenen Grafiktypen. Es ist allerdings damit zu rechnen, dass die Bäume der verwendeten und der abhängigen IDs nicht vollständig sind, da Itéa dazu den gesamten Object-Pascal-Code vollständig parsen müsste, was jedoch nicht bei allen Methoden funktioniert (zumal die Itéa-Fähigkeiten zum Parsen von Methoden-Code sich noch auf dem Niveau der letzten Itéa-Beta-Version für Delphi 4 befinden).

Editieren und Ausdrucken der Grafiken

Wenn Sie eine der drei beschriebenen Grafiken in Itéa erzeugen, startet Itéa eine eigene Version des TreeDesigners, den *TreeViewer*. Da dieser jedoch gegenüber dem TreeDesigner 3.5 veraltet ist und weder drucken noch Metadateien exportieren kann, ist es empfehlenswert, nicht nur die Klassenhierarchie, sondern *alle* Grafiken statt in Itéas TreeViewer im TreeDesigner 3.5 weiterzuverarbeiten. Speichern Sie dazu in Itéas *TreeViewer* die Grafik einfach in einer TVF-Datei und laden Sie diese dann in den aktuellen TreeDesigner. Im Falle der Klassenhierarchiegrafiken können Sie den Aufruf des TreeViewers ganz vermeiden, indem Sie die beschriebene Vorgehensweise verwenden (BROWSER | KLASSENHIERARCHIE -> ZWISCHENABLAGE in Itéa aufrufen, dann Einfügen im TreeDesigner; siehe Kapitel 5.1.2).

5.2 Das Hauptfenster

Dieses Kapitel beschäftigt sich mit dem groben Aufbau der Fenster des TreeDesigners und behandelt dabei beispielhaft sowohl die zahlreichen Verwendungsmöglichkeiten von Symbolleisten als auch grundlegende Aufgaben von Dokumentformularen aller Art:

- Bezüglich der Symbolleisten beginnt Abschnitt 5.2.1 im Kleinen, und zwar bei den einzelnen Leisten, basierend auf den Klassen *TToolbar* oder *TPanel*.
- Abschnitt 5.2.2 beschreibt die Einbindung dieser Leisten in das Gesamtkonzept des Fensters, wozu die Verwendung von *TControlBar*- und *TCoolBar*-Komponenten und die freie Positionierung der Leisten durch den Benutzer (inklusive Andocken) gehören.
- Abschnitt 5.2.3 widmet sich der Menüleiste des TreeDesigners, welche auf eine der Delphi-IDE entsprechenden Art mit Hilfe einer Toolbar realisiert wurde.
- Abschnitt 5.2.4 stellt kurz die scrollbare Zeichenfläche des Dokumentformulars vor.
- In den Abschnitten 5.2.5 und 5.2.6 geht es um weitere Aufgaben, die eigentlich jedes Dokumentfenster verrichten muss, wie etwa die Behandlung der grundlegenden Punkte des Dateimenüs, eine Sicherheitsabfrage zum Speichern der Datei, Markierung und Aktivierung von Menüpunkten und Bereitstellung von Popup-Menüs.

Hinweis: In der hier vorausgesetzten MDI-Version des TreeDesigners werden drei der Symbolleisten, die zur Laufzeit unter der Menüzeile des Hauptfensters erscheinen, im Dokumentformular definiert. Die besonderen Vorteile dieser Vorgehensweise werden in Kapitel 5.7.5 beim dynamischen Werkzeugleistenmanagement erläutert, in diesem Kapitel spielt es noch keine Rolle, wie die Leisten zur Laufzeit in das Hauptfenster gelangen.

5.2.1 Der Entwurf von Symbol- und Werkzeugleisten

Obwohl es nicht verboten ist, ein Dokumentformular ähnlich wie ein Dialogfenster an beliebigen Positionen mit Steuerelementen anzureichern, werden die benötigten Steuerelemente üblicherweise in Symbol- oder Werkzeugleisten zusammengefasst. Als Basiskomponenten zur Aufnahme der einzelnen Steuerelemente kommen in Delphi vornehmlich zwei Komponentenklassen in Frage: *TToolbar* von der Seite WIN32 und *TPanel* von der Standardseite.

Beide dienen den in ihnen enthaltenen Komponenten als Elternfenster, während das Formular der Besitzer der Komponenten ist (Genaueres zu Steuerelement-Gruppen

siehe Kapitel 3.5.1). Ein besonderer Vorteil der Elternschaft ist, dass die Komponenten bei jeder Verschiebung des Panels mitbewegt werden – das beinhaltet auch eine vom Anwender zur Laufzeit gewünschte Verschiebung, zum Beispiel vom oberen an den unteren Fensterrand (per Umschaltung des *Align*-Properties von *alTop* auf *alBottom*) oder auch alle Verschiebungen im Rahmen des Dockings.

Panels

Um aus einem Panel eine Zeile oder Spalte zu machen, die an einem der vier Fensterränder angeheftet ist und die sich bei jeder Größenänderung des Fensters mit anpasst, müssen Sie lediglich das Panel-Property *Align* auf einen der Werte *alTop*, *alBottom*, *alLeft* oder *alRight* setzen (siehe Kapitel 3.3.1). Drei der Koordinaten werden daraufhin automatisch bestimmt. Im Falle einer Mauspalette (*alTop*) sind das z.B. die Properties *Left*, *Width* und *Top*. Sie können also die Breite und Position eines *alTop*-ausgerichteten Panels nicht mehr verändern, da es sich hier nach seinem Elternfenster, also normalerweise dem Formular, richtet. Lediglich Änderungen der Höhe sind bei *alTop* und *alBottom* noch möglich.

ToolBars *R2*

Die Klasse *TToolBar* beinhaltet automatische Mechanismen zur Anordnung von Unterelementen: Grundsätzlich werden die Elemente in einer Zeile von links nach rechts angeordnet, und zwar dicht aneinander. Um Zwischenräume zu erhalten, müssen Sie spezielle Trennelemente einfügen. Zur Entwurfszeit können Sie die einzelnen Elemente außerdem per Drag&Drop an eine andere Position bewegen.

Als Unterelemente kommen grundsätzlich alle Komponenten in Frage, normalerweise besteht eine ToolBar jedoch hauptsächlich aus Elementen der Klasse *TToolButton*, die Sie zur Entwurfszeit über das Kontextmenü der ToolBar einfügen. In diesem Menü ist von einem »neuen Schalter« und einem »neuen Trenner« die Rede, tatsächlich handelt es sich aber bei beiden um die gleichen *TToolButton*-Elemente, die sich nur durch das Property *Style* unterscheiden. Und es gibt sogar noch mehr als diese beiden Stile:

- *tbsButton* ist der schon erwähnte normale Schalter. Seine Oberfläche wird normalerweise durch ein grafisches Symbol geziert, kann aber auch eine einfache Textbeschriftung zugewiesen bekommen (*TToolBar.ShowCaptions* muss dann *True* sein).
- *tbsSeparator* steht für das schon erwähnte Trennelement, dessen Breite Sie frei wählen können.
- *tbsDivider* ist eine Alternative zu *tbsSeparator*, es zeigt zusätzlich eine vertikale Linie an.
- *tbsCheck* ist eine Variation von *tbsButton*. Äußerlich stimmt ein *tbsCheck*-Element mit einem normalen Schalter überein, das Verhalten gleicht jedoch eher einer Check-

Box, denn ein einmaliges Anklicken versetzt den Schalter in einen dauerhaften gedrückten Zustand, der bis zum nächsten Klicken anhält.

- *tbsDropDown* ergänzt den *tbsButton*-Schalter durch einen Pfeil, über den sich ein DropDown-Menü aufklappen lässt. Über die normale Schaltfläche lässt sich weiterhin eine Aktion direkt, also ohne Menü, auslösen. Typisches Beispiel für diese Schalter sind Schalter zum Öffnen von Dateien, bei denen über den Pfeil eine Liste der zurückliegenden Dateien aufgerufen werden kann, z.B. in der Delphi-IDE oder im TreeDesigner.

Um einen *TToolButton* mit einem Bild zu versehen, geben Sie im Property *ImageIndex* einen Bildindex an. Dieser bezieht sich auf die Bilderliste, mit der die Toolbar verknüpft ist, und zwar über das Property *TToolBar.Images*. Für die Gestaltung einer ToolBar benötigen Sie also auch noch eine *TImageList*-Komponente, in die Sie die Bilder für alle ToolButtons laden, wie in Kapitel 3.6.2 in Zusammenhang mit ListViews beschrieben.

Die Schalterleisten des TreeDesigners

Da es nahe liegt, die Schalter mit den verschiedenen grafischen Formen als »Zeichenwerkzeuge« zu bezeichnen, soll die Leiste am linken Rand des TreeDesigner-Hauptfensters als »Werkzeugleiste« bezeichnet werden, alle anderen Leisten am oberen und unteren Rand als »Symbolleisten« . Während für alle anderen Symbolleisten die Klasse *TToolbar* verwendet wurde, basiert die Werkzeugleiste auf der Komponente *TPanel*, da es in dieser einfacher ist, Schalter vertikal übereinander anzuordnen. Die Werkzeugleiste soll außerdem nach den Schaltern noch einen kleinen Vorschaubereich für die eingestellten Farben und die Liniendicke enthalten. Auch dieser lässt sich in einem *TPanel* erheblich stressfreier entwerfen als in einer Toolbar, die einem mit diversen Funktionen zur automatischen Anordnung und Größeneinstellung des Öfteren ins Handwerk pfuscht.

Die Symbolleisten enthalten außer weiteren Schaltern auch interaktive Elemente. Diese und alle weiteren Komponenten der Werkzeug- und Symbolleisten, die in den folgenden Kapiteln innerhalb des Beispielcodes erwähnt werden, sind in der folgenden Tabelle zusammengefasst:

Komponente	Klasse	Aufgabe
SpeedBtnArrow	TSpeedButton	schaltet den Manipulationsmodus an, in dem die Maus zum Verschieben, Markieren und Ändern der Größe verwendet wird.
SpeedBtnRectangle, SpeedBtnEllipse, SpeedBtnLine, speedBtnRhomb, ...	TSpeedButton	wählen jeweils eine Form aus, die gezeichnet werden soll. In Frage kommen die Formen, die im Aufzählungstyp *TShapeType* der Unit *GrDoc* genannt sind (stNone, stRect, stEllipse, stLine, stHexagon, stRhomb, stRoundRect).

Komponente	Klasse	Aufgabe
SpeedBtnArrangeTree	TSpeedButton	Schalter zum Anordnen des Baums.
ShapeText	TEdit	Hier geben Sie die Beschriftung für das gewählte Element oder die gewählten Elemente ein.
FontList	TComboBox	Hier können Sie eine andere Schrift für die markierten Elemente auswählen.
FontSize	TUpDown	Pfeilschalter-Kontrollelement (Klasse *TUpDown*) zur Einstellung des Zahlenwertes im nebenstehenden Editierfeld *FontSizeEdit*.
ZoomFactorBar	TTrackBar	Regler, mit dem Sie fließend auf den Mittelpunkt des dargestellten Bildbereichs zoomen können bzw. mit dem Sie das Bild verkleinern können.
ZoomLabel	TLabel	zeigt den aktuellen Vergrößerungsfaktor an (0.1 bis 10.0).

Einstellungen der Properties im Objektinspektor sind für verschiedene Funktionsbereiche notwendig: zur Anzeige von Symbolen auf den Schaltflächen, zur Bildung von Schaltergruppen (mit diesen beiden Themen befassen sich zwei der folgenden Abschnitte) und zum allgemeinen optischen Erscheinungsbild von Schaltern und Leisten.

Zu Letzterem gehört vor allem, dass der TreeDesigner flache Schalter verwendet. Hierzu wurden alle *Flat*-Properties von *TToolBar* und *TSpeedButton*-Komponenten auf *True* gesetzt. *TToolButton*-Komponenten brauchen nicht einzeln auf flach geschaltet zu werden, da sie immer zusammen mit der gesamten Toolbar zwischen flach und dreidimensional hin- und hergeschaltet werden.

Property-Einstellungen nachvollziehen *R5*

Die weiteren Properties sollen hier nicht im Einzelnen untersucht werden, da sich die meisten schon durch die Bezeichnung im Objektinspektor selbst erklären. Wenn Sie sich einmal fragen sollten, wie in einem beliebigen Beispielprogramm, etwa auch einem aus Delphis Lieferumfang, welche Properties bei einer Komponente verändert wurden, brauchen Sie nur ANSICHT ALS TEXT aus dem lokalen Menü des Formulars auszuwählen. In der Textdarstellung werden *alle* von der Standardeinstellung abweichenden Werte genannt, die nicht davon abweichenden Werte werden normalerweise nicht genannt. Mit der Suchfunktion finden Sie auch in der Textdarstellung jede Komponente sofort. Für die Toolbar mit den Zoom-Einstellungen im TreeDesigner werden beispielsweise die folgenden Werte angezeigt:

```
object ZoomToolbar: TToolBar
    Left = 0
    Top = 45
    Width = 796
    Height = 28
```

```
ButtonWidth = 25
Caption = 'Zoomeinstellung'
EdgeBorders = []
Flat = True
Images = MenuImages
...
```

Bebilderung

Das Dokumentformular des TreeDesigners enthält eine *TImageList*-Komponente mit ca. 20 Bildern, die sowohl für die Schalter als auch für die Menüs verwendet werden. Das bedeutet, dass die *Images*-Properties aller Toolbars und auch das der *TMainMenu*-Komponente auf diese Bilderliste verweisen. Damit die Bilder in die Menüs passen, ohne die üblichen Maße für einen Menüeintrag durcheinander zu bringen, wurden sie auf eine Größe von 16*16 Pixeln beschränkt. Ein weiterer Grund für diese Vorgehensweise ist, dass die mit Delphi mitgelieferten Beispielbilder in dieser Größe vorliegen. Die im TreeDesigner verwendeten Bitmaps stammen zu einem Teil aus den Bilddateien des Verzeichnisses `Borland\Gemeinsame Dateien\images`, zum Teil aus den Projektschablonen für MDI- und Windows95-Logo-konforme Anwendungen. Das Bild für die Baum-Anordnung ist vom Autor erstellt und von einem Bildverarbeitungsprogramm auf 16*16-Pixel heruntergerechnet worden.

Die SpeedButton-Schalter in der Werkzeugleiste verwenden ein etwas größeres Schalter-Format und 20*20-Pixel große Bitmaps, die allerdings nicht in eine Bilderliste, sondern direkt in das *Glyph*-Property der einzelnen Schalter geladen werden.

Hinweis: Da *Glyph*-Properties Objekte des Typs *TBitmap* sind, können Sie mit den einfachen *TBitmap*-Methoden auch zur Laufzeit neue Bilder in die Schalter laden. Wenn Sie für verschiedene Schalterzustände verschiedene Bitmaps verwenden wollen, müssen Sie die in der Online-Hilfe beschriebenen *TSpeedButton*-Properties *Glyph*, *Layout*, *Margin*, *Spacing* und *NumGlyphs* verwenden und mehrere Bilder in einer Bitmap speichern.

Schalter-Gruppen

Die Zeichenwerkzeug-Schalter im TreeDesigner verhalten sich ähnlich wie eine Radioschaltergruppe, es kann also immer nur ein Schalter gewählt sein. Während der TreeDesigner in diesem Fall *TSpeedButtons* verwendet, können Sie Schaltergruppen auch mit *ToolButtons* realisieren. Damit die VCL die Einhaltung der Gruppen-Regel sicherstellt und den bisher gewählten Schalter automatisch abwählt, wenn ein anderer Schalter der Gruppe gedrückt wird, müssen Sie

- bei *TSpeedButtons* allen Schaltern der Gruppe eine übereinstimmende Kennzahl im Property *GroupIndex* geben, im TreeDesigner ist das der Index 1. *GroupIndex* ist normalerweise mit Null voreingestellt, was bedeutet, dass ein Schalter zu keiner Gruppe gehört.
- bei *TToolButtons* das *Grouped*-Property aller Schalter, die eine Gruppe bilden, auf *True* setzen. Wenn mehrere Gruppen in einer Toolbar vorhanden sind, erkennt die Toolbar die Gruppengrenzen an Separatorschaltern oder an anderen Schaltern, bei denen *Grouped* auf *False* gesetzt ist.
- In beiden Fällen bewirkt das Property *AllowAllUp* mit seiner standardmäßigen *False*-Einstellung, dass immer ein Schalter der Gruppe gedrückt sein muss. Da der TreeDesigner jedoch das standardmäßige Verhalten der Schalter etwas abändert, wurden die *AllowAllUp*-Properties seiner Gruppenschalter auf *True* gesetzt.

Programmierung

Die Programmierung von Symbolleisten kommt normalerweise mit Standard-Techniken aus und soll daher an dieser Stelle lediglich stichpunktartig abgehandelt werden:

- Zur Bearbeitung von Aktionsschaltern haben Sie die Wahl, die Bearbeitungsmethode einfach mit dem *OnChange*-Ereignis des Schalters zu verknüpfen oder eine Aktion in einer Aktionsliste dafür bereitzustellen, die Sie dann auch mit einem gleichbedeutenden Menüpunkt verknüpfen können. Zu Vorteilen und Verwendung von Aktionslisten siehe Kapitel 4.6.4.
- Die Einstellungen, die in den Kontrollelementen der Leisten vorgenommen werden, können direkt über Properties der Komponenten abgefragt werden, z.B. *ShapeText.Text* für das Editierfeld der Beschriftung und *PenWidth.Position* für die Einstellung des Trackbars für die Stiftbreite.
- Zur Abfrage von Schaltergruppen gibt es spezielle Techniken, die in Kapitel 3.5.2 behandelt sind. So können Sie im TreeDesigner etwa über die SpeedButtons an der linken Fensterseite eine Grafikform einstellen. Jeder SpeedButton steht dabei für einen anderen Wert in der Aufzählung *TShapeType*. Die *Tag*-Properties der SpeedButtons wurden auf den jeweils zu ihnen passenden *TShapeType*-Wert eingestellt, damit sich der Wert schneller ermitteln lässt (zu dieser Verwendung des *Tag*-Properties siehe *R12* auf Seite 379).
- Bei manchen Leisten-Steuerelementen soll sich eine Änderung sofort auf das bearbeitete Dokument auswirken. Im TreeDesigner wird z.B. der editierte Beschriftungstext (Komponente *ShapeText*) bei jeder Änderung im Grafikdokument aktualisiert und auch die Wahl einer anderen Farbe aus der Farbpalette wirkt sich sofort auf die selektierten Grafikobjekte aus. Hierzu müssen lediglich die Änderungs-Ereignisse bearbeitet werden, bei *ShapeText* das Event *TEdit.OnChange* und

bei der Farbpalette die Events *OnPenColorChange*, *OnBrushColorChange* und *OnFontColorChange*. Ein Beispiel für die Bearbeitung eines solchen Events finden Sie in Kapitel 5.3.2. Eine zusätzliche Herausforderung im Falle des TreeDesigners besteht darin, diese Events an das richtige Fenster weiterzuleiten, denn es gibt im TreeDesigner nur eine einzige Farbpaletten-Komponente für alle Dokumentfenster. Und die Farbänderungs-Events sollen sich nur auf das aktuelle Fenster auswirken. Wie dies im TreeDesigner gelöst ist, zeigt Kapitel 5.7.6.

5.2.2 Docking von Symbolleisten

Die vielfältigen Möglichkeiten des Dockings lassen sich grob in zwei verschiedene Einsatzgebiete unterteilen, die auch von der Delphi-IDE demonstriert werden:

- Beim Docking von Symbolleisten geht es darum, dass die Leisten, die früher (bis Delphi 3) immer fest verankert waren, nun beliebig verschoben werden können. Für den Benutzer kommt es hier wahrscheinlich hauptsächlich auf das Abtrennen der Symbolleisten vom übergeordneten Fenster an. Dieses Abtrennen wird in den Komponenten der VCL als *Undocking* bezeichnet, der Zustand einer in ein eigenes Fenster ausgelagerten Symbolleiste wird als *Floating* bezeichnet.
- Beim Docken von Fenstern geht es darum, dass die bisher einzeln vorliegenden Fenster nun verbunden werden, um dem Benutzer die Aufteilung der Bildschirmfläche zu vereinfachen und Probleme mit verdeckten Fenstern zu vermeiden.

Die zweite Art des Dockings wird in Kapitel 5.8.2 beschrieben, hier geht es nur um das Docking der Symbolleisten.

TControlBar und TCoolBar

Für die uns hierbei als Nächstes begegnende Komponente gibt wieder einmal die Delphi-IDE selbst das beste Anwendungsbeispiel. Menüleiste, Symbolleisten und die Komponentenpalette befinden sich in einem Exemplar der *TControlBar*-Komponente, die Borland als Alternative zu *TCoolBar* entwickelt hat. Während *TCoolBar* auf einem Microsoft-Steuerelement aus der Common Controls-Bibliothek basiert und für Delphi in der Unit *ComCtrls* definiert wird, ist *TControlBar* vollständig in über 1000 Zeilen der Unit *ExtCtrls* implementiert. *TControlBar* macht Borland (und alle Entwickler, die diese Komponente verwenden) von den ständigen Änderungen unabhängig, die Microsoft eine Zeit lang an seinen Common Controls vorgenommen hat.

Auch im TreeDesigner wurde der *ControlBar* der Vorzug gegenüber der *CoolBar* gegeben, unter anderem aus den schon in Kapitel 1.9.2 genannten Gründen. Normalerweise kann eine *TControlBar* jedoch sehr leicht gegen eine *TCoolBar* ausgetauscht werden, denn die Toolbars, aus denen beide meistens bestehen, brauchen dafür nicht verändert zu werden.

Bänder

Wie also die Delphi-IDE zeigt, dient eine *ControlBar* dazu, mehrere Symbolleisten auf einer Art Zeilenraster darzustellen und dem Benutzer zu erlauben, die Leisten flexibel auf diesem Raster anzuordnen. Dabei kann es auch erlaubt sein, eine Symbolleiste ganz aus der ControlBar herauszunehmen.

Sowohl *TControlBar*- als auch *TCoolBar*-Komponenten bestehen aus mehreren *Bändern*, wobei auch mehrere Bänder nebeneinander liegen können. Im Falle von *TCoolBar* werden die Bänder immer so ausgedehnt, dass die gesamte Breite der CoolBar in Anspruch genommen wird, *TControlBar* lässt dagegen auch Freiräume zu.

Ein Band hat an seiner linken Seite einen Griff, bei dessen grafischer Gestaltung sich *TControlBar*-Komponenten durch eine doppelte statt einer einfachen Linie zu erkennen geben. An diesem Griff können Sie das Band innerhalb einer Zeile verschieben (wobei die Position der anderen Bänder derselben Zeile automatisch angepasst wird) oder es in eine andere Zeile versetzen. Hat das *AutoSize*-Property der Control/CoolBar den Wert *True*, können Sie auf diese Weise sogar aus einer einzeiligen eine mehrzeilige Komponente machen (zu Entwurfszeit und Laufzeit).

Auch in der Verwaltung der Kindelemente stimmen ControlBar und CoolBar überein: Jedes Kindelement erhält ein eigenes Band mit eigenem Griff, daher werden normalerweise mehrere Einzelkomponenten in einer größeren Komponente zusammengefasst, so dass sie zusammen als Gruppe ein einziges Band zugeteilt bekommen. Die Bänder werden außerdem automatisch erzeugt, sobald sie eine Komponente einfügen. Bei der *TCoolBar*-Komponente haben Sie außerdem die Möglichkeit, Properties der einzelnen Bänder einzustellen, indem Sie sich über das Property *TCoolBar.Bands* eine Bänder-Liste auf den Bildschirm holen.

Das Hauptformular des TreeDesigners verwendet eine ControlBar mit fünf Bändern am oberen Fensterrand und eine weitere mit nur einem Band am unteren Rand, allerdings steht es dem Benutzer zur Laufzeit frei, einige der Bänder abzutrennen oder von der oberen in die untere Leiste und umgekehrt zu verschieben.

Docking-Vorbereitungen zur Entwurfszeit

Um eine Symbolleiste (z.B. *TToolBar*, *TPanel*) von ihrer Elternkomponente (z.B. *TControlBar*, *TCoolBar*) abtrennbar zu machen, sind nur wenige Schritte erforderlich:

- Bei der Komponente, die abgetrennt werden soll, stellen Sie das *DragKind*-Property auf *dkDock* und *DragMode* auf *dmAutomatic*. *DragMode* wird auch für Drag&Drop benutzt (siehe Kapitel 5.8.3) und besagt im Modus *dmAutomatic*, dass die VCL automatisch einen Ziehvorgang initiiert, wenn der Benutzer mit der linken Maustaste auf die Komponente klickt. Über *DragKind* teilen Sie der VCL mit, ob es sich bei

diesem Ziehvorgang um das in allen Delphi-Versionen unterstützte Drag&Drop oder um das erst ab Delphi 4 unterstützte Docking handeln soll.

- Bei der Elternkomponente der abtrennbaren Leiste gilt es, das *DockSite*-Property auf *True* einzustellen, sonst können Sie eine Leiste zwar abtrennen, aber nicht wieder andocken. Für die automatische Anpassung der Größe der Elternkomponente setzen Sie bei dieser auch noch das *AutoSize*-Property auf *True*.

Diese einfachen Maßnahmen können einer Anwendung bereits zu einer gut funktionierenden Docking-Funktionalität verhelfen, um die sich ausschließlich die VCL kümmert.

Ablauf des Dockings auf der Ebene des Programms

Aus der Sicht des Programms besteht ein Docking-Vorgang aus einer Abfolge von Ereignissen, die an die verschiedenen am Vorgang beteiligten Komponenten gesendet werden. Diese Komponenten sind die Elternkomponente, in der sich die gezogene Komponente vor dem Vorgang befand, die gezogene Komponente selbst und alle Komponenten, über die die Maus während des Vorgangs bewegt wird, inklusive der Komponente, auf der sie schließlich losgelassen wird.

Zunächst muss der Vorgang jedoch gestartet werden, was Sie auch im Programm tun können, indem Sie wie beim Drag&Drop die Methode *BeginDrag* aufrufen, beispielsweise anlässlich eines *OnMouseDown*-Ereignisses. Im automatischen Modus wird *BeginDrag* von der VCL selbst aufgerufen, von da an geht es in beiden Fällen auf die gleiche Art weiter:

- Beim Beginn des Vorgangs erzeugt die gezogene Komponente ein *OnStartDock*-Ereignis.
- Falls das gezogene Steuerelement nun aus einer Elternkomponente herausgezogen wird, meldet sich die Elternkomponente mit einem *OnUndock*-Ereignis.
- Jede mögliche Andock-Komponente, über der die gezogene Komponente nachfolgend bewegt wird, sendet dem Formular ein *OnDockOver*-Ereignis.
- Lässt der Benutzer das gezogene Element über einer zulässigen Andockstelle los, so gibt es von der Andockstelle ein *OnDockDrop*-, von der gezogenen Komponente ein *OnEndDock*-Ereignis.

Wenn Sie keines dieser Ereignisse bearbeiten, führt die VCL nach dem Loslassen der Maustaste den Andockvorgang durch, indem sie das *Parent*-Property der gezogenen Komponente auf die neue Andockstelle setzt. Das *Owner*-Property wird dabei nicht verändert, Besitzer der Komponente bleibt also das Formular, in das Sie die Komponente zur Entwurfszeit eingefügt haben. Die Elternkomponente, in der eine Komponente angedockt wird, wird in der VCL als *HostDockSite* bezeichnet.

Wird eine Komponente an eine Bildschirmposition gezogen, an der es keine Andockstelle gibt, so erzeugt die VCL automatisch ein neues Elternfenster im *wsSizeToolWin*-Stil und fügt die gezogene Komponente darin ein. Im VCL-Sprachgebrauch hat dieses Fenster die Bezeichnung *FloatingDockSite*. Wird die Komponente später wieder angedockt, so kann die VCL das Hilfsfenster wieder löschen. Standardmäßig verwendet die VCL für eine *FloatingDockSite* die Klasse *TCustomDockForm*. Sie können jedoch auch eine eigene (von *TCustomDockForm* abgeleitete) Klasse als Vorlage für diese Hilfsfenster einsetzen, indem Sie das Property *FloatingDockSiteClass* der andockbaren Komponente auf Ihre Klasse setzen.

Hinweis: Ein Hilfsfenster wird nur dann erzeugt, wenn die abgetrennte Komponente nicht selbst ein *Formular* ist, das auch ohne Elternkomponente frei über den Bildschirm bewegt werden kann. Dies ist im TreeDesigner beispielsweise beim TreeExplorer-Fenster der Fall, um den es in Kapitel 5.8.2 gehen wird.

Akzeptieren von andockbaren Elementen

Das *OnDockOver*-Ereignis ist ein besonders wichtiges, denn bei diesem erhält die Andockstelle eine Möglichkeit, anzuzeigen, dass sie für ein bestimmtes andockbares Element nicht geeignet ist. Und es können sehr schnell Situationen entstehen, in denen nicht jede Andockstelle jedes andockbare Fenster aufnehmen kann.

Im TreeDesigner gibt es beispielsweise drei Andockstellen: Die ControlBars am oberen und am unteren Fensterrand sowie ein Panel an der rechten Seite. Diesen stehen eine Reihe von Toolbars sowie das TreeExplorer-Fenster gegenüber. Es lässt sich leicht ausprobieren, dass die beiden ControlBars nicht zur Aufnahme des TreeExplorers geeignet sind, da die VCL ihre Größe der Höhe des TreeExplorers anpassen würde, was die den MDI-Kindfenstern zur Verfügung stehende Fläche arg dezimieren würde.

Im TreeDesigner ist also die Andockstelle für den TreeExplorer so programmiert, dass sie nur den TreeExplorer und keine Symbolleisten annimmt; die beiden ControlBars stellen genau die gegenteilige Bedingung. Das folgende Beispiel zeigt die Bearbeitung der von diesen ControlBars gesendeten *OnDockOver*-Ereignisse:

```
procedure TMainForm.ToolbarDockSitesDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
begin
  Accept := not (Source.Control is TTreeExplorerForm);
```

Wie oben beschrieben, tritt das *OnDockOver*-Ereignis bei jeder Bewegung der Maus über einer möglichen Andockkomponente auf. Standardmäßig setzt die VCL voraus, dass jede Andockstelle jede dockbare Komponente aufnehmen darf. Durch Zuweisung des Wertes *False* an den Variablenparameter *Accept* verhindern Sie das Andocken.

Wichtig ist hierbei auch die Unterscheidung zwischen der möglichen Andockstelle (*Sender*) und dem gezogenen Element (*Source.Control*). Der *Sender*-Parameter braucht im obigen Beispiel nicht beachtet zu werden, weil dafür nur die beiden ControlBars in Frage kommen, mit deren *OnDockOver*-Ereignis die Methode im TreeDesigner verknüpft ist.

Verstecken und Anzeigen von Symbolleisten

Anwendungen mit mehreren Symbolleisten erlauben es dem Benutzer normalerweise, die Leisten einzeln zu verstecken und wieder sichtbar zu machen. Damit der TreeDesigner auch hier den Spuren der Delphi-IDE folgt und Sie mit einem Klick der rechten Maustaste auf jeder Symbolleiste ein entsprechendes Menü aufrufen können, sind allerdings einige Zeilen Quelltext notwendig.

Im Hauptformular des TreeDesigners finden Sie unter dem Namen *ControlBarPopup* ein Popup-Menü, das je einen Eintrag für jede der sechs Leisten enthält. Jeder Menüpunkt ist einzeln an- und abwählbar und über das *OnClick*-Ereignis mit der folgenden Methode verknüpft:

```
procedure TMainForm.ShowColorPaletteClick(Sender: TObject);
var
  MenuItem: TMenuItem;
  Control: TControl;
begin
  { Feststellen, welcher der sechs Menüpunkte angewählt wurde: }
  if Sender = ShowColorPalette then Control := Palette
  else if Sender = ShowGlobalToolbar then Control := Toolbar1
  ...
  { Control enthält jetzt die anzuzeigende/zu versteckende Leiste }
  MenuItem:=Sender as TMenuItem;
  MenuItem.Checked := not MenuItem.Checked;
  if Control.Floating then
    Control.HostDockSite.Visible := MenuItem.Checked
  else Control.Visible := MenuItem.Checked;
end;
```

Entscheidend ist, dass es bei abtrennbaren Elementen nicht genügt, sie einfach sichtbar oder unsichtbar zu machen, sondern es muss zusätzlich geprüft werden, ob das Element sich nicht im Zustand *Floating* (= »frei schwebend«) befindet. Ist das der Fall, muss ja in erster Linie das Hilfs-Elternfenster (die *HostDockSite*) unsichtbar gemacht werden.

Speichern der Docking-Einstellungen *R118*

Es gibt noch eine zweite, vielleicht noch wichtigere Funktion, für die die VCL ebenfalls keinerlei Automatik zur Verfügung stellt: Normalerweise wird sich ein Anwender wünschen, dass er seinen persönlichen Bildschirmaufbau nicht nach jedem Programm-

start neu entwerfen muss, sondern dass das Programm die Position, Sichtbarkeit und sonstige Konfiguration der Symbolleisten speichert.

Der TreeDesigner speichert daher im *OnClose*-Ereignis des Hauptformulars alle Symbolleistenpositionen, um sie im nächsten *OnCreate*-Ereignis wiederherstellen zu können. Ort der Speicherung ist die Windows-Registry, und zwar der Schlüssel *Software \ TreeDesigner \ NameDerSymbolleiste*.

Zur leichteren Wiederverwendung wurde das Speichern und das Wiederherstellen der Leisten in Form von allgemeinen Prozeduren in die Unit *TDUtil* ausgelagert. Sie können diese Prozeduren nach folgendem Muster für andere Symbolleisten wiederverwenden:

```
SaveToolbarDockStatus(IniPath+'Name', Symbolleiste); // Speichern
...
RestoreToolbarDockStatus(IniPath+'Name', Symbolleiste); // Wiederherst.
```

Da diese Prozeduren eher arbeitsaufwändig als interessant sind, beschränken wir uns hier auf die Schlüsselposition im Programm-Code zum Wiederherstellen der Symbolleistenkonfiguration. Dazu ist zu beachten, dass sich nach jedem neuen Programmstart alle Symbolleisten an der zur Entwurfszeit festgelegten Stelle befinden. Das Programm muss also eventuell selbst dafür sorgen, dass einige der Symbolleisten abgetrennt werden.

Falls es sich um eine abgetrennte Symbolleiste handelt (was in Form eines booleschen Wertes natürlich in der Registry gespeichert werden muss), werden in *RestoreDockStatus* die vier Koordinaten der Bildschirmposition der Komponente aus der Registry ausgelesen und die Komponente wird programmgesteuert aus der ControlBar herausgenommen. Hierfür stellt die VCL die Methode *ManualFloat* zur Verfügung:

```
WindowRect:=Rect(R.ReadInteger('ScreenLeft'), R.ReadInteger('ScreenTop'),
             R.ReadInteger('ScreenRight'), R.ReadInteger('ScreenBottom'));
Toolbar.ManualFloat(WindowRect);
```

Die im *ManualFloat*-Parameter angegebene Bildschirmposition gibt die exakte Positionierung der frei schwebenden Komponente am Bildschirm an, also nicht die Positionierung des äußeren Hilfsfensters.

Beim Speichern der Position, also in der Prozedur *SaveDockStatus*, wird dieses Rechteck mit den beiden folgenden Zeilen ermittelt:

```
ScreenRect.TopLeft := Toolbar.ClientToScreen(Control.ClientRect.TopLeft);
ScreenRect.BottomRight :=
              Toolbar.ClientToScreen(Control.ClientRect.BottomRight);
```

Hinweis: Im Falle des TreeExplorer-Fensters liegt wieder eine andere Situation vor. Wenn es nicht angedockt ist, braucht beim nächsten Programmstart kein *ManualFloat* für dieses Fenster aufgerufen zu werden, da es standardmäßig schon frei ist. Ein zusätzlicher Eingriff des Programms wird nur dann erforderlich, wenn es als angedocktes Fenster wiederhergestellt werden soll, in diesem Fall ist *ManualDock*, das Gegenstück zu *ManualFloat*, aufzurufen. Details hierzu finden Sie im Programmcode, weitere Besonderheiten des Dockings des TreeExplorers werden in Kapitel 5.8.2 beschrieben.

5.2.3 Menüleisten im Toolbar-Stil

Ein letzter Blick auf das Vorbild der Delphi-IDE zeigt, dass eine ControlBar für den Benutzer noch komfortabler wird, wenn auch das Menü in einer der verschiebbaren Leisten untergebracht wird, denn dann kann der freie Platz neben dem Menü beispielsweise für eine Symbolleiste genutzt werden (außerdem könnte eine solche Menüleiste auf die in Kapitel 5.2.2 beschriebene Weise sogar abtrennbar gemacht werden, was jedoch weder in der Delphi-IDE noch im TreeDesigner vorgesehen ist).

Eingebaute Funktionen von TToolBar

Die *ToolBar*-Komponente von Delphi 6 verfügt über die Fähigkeit, automatisch den Inhalt einer *MainMenu*-Komponente anzuzeigen. Sie müssen dazu das *MainMenu* lediglich im *Menu*-Property der *ToolBar* angeben und – wenn Sie die übliche Menü-Optik erhalten wollen – das Property *Flat* einschalten.

Die Toolbar erzeugt dann für jeden Menüpunkt der obersten Ebene einen ToolButton und verknüpft ihn mit dem Menüpunkt. Wenn zur Laufzeit Änderungen an diesem Pulldown-Menü vorgenommen werden (z.B. weil eine Liste von zuletzt geöffneten Dateien oder eine Liste offener Fenster aktualisiert wird), dann wirken sich diese Änderungen auch aus, wenn Sie das Menü über die Toolbar aufrufen. Allerdings aktualisiert die Toolbar ihre Schalter nicht, wenn die Menüpunkte der obersten Ebene verändert werden. Und für den TreeDesigner wäre besonders die Tatsache ungünstig, dass *TToolBar* keine automatische Menüverschmelzung unterstützt, wie sie bei MDI-Anwendungen benötigt wird.

Hinweis: Auch vor Delphi 6 können Sie *TToolBar* leicht für die Anzeige von Menüs verwenden, indem Sie in der ToolBar *ShowCaptions* auf *True* setzen und für jeden Menüpunkt der obersten Ebene einen *ToolButton* erzeugen, in dessen *MenuItem*-Property Sie den Menüpunkt angeben. Um die Funktionsweise des ToolBar-Menüs zu perfektionieren, schalten Sie noch die Properties *Grouped* und *AutoSize* des *ToolButtons* ein.

Einen kleinen »Schönheitsfehler« hat die automatische Darstellung von Menüs durch *TToolBar* noch: Die standardmäßig von *TToolBar* verwendete Schriftart stimmt nicht mit der Systemschriftart für Menüs überein (diese wird in der Systemsteuerung unter den ANZEIGE-Eigenschaften auf der Seite DARSTELLUNG eingestellt). Wie Sie das ändern, lesen Sie im Abschnitt *Auf Änderungen der Systemeinstellungen reagieren* weiter unten.

Die Komponente TMenuToolbar

Unter den Komponenten zum Buch finden Sie auch eine Klasse namens *TMenuToolBar*. Diese erledigt wie die erweiterte Delphi-6-ToolBar alle Schritte zur Erzeugung einer ToolBar-Menüleiste automatisch und ist darüber hinaus in der Lage, die in Kapitel 5.7.3 beschriebene Menüverschmelzung von MDI-Anwendungen durchzuführen. Nachteil gegenüber *TToolBar* ist, dass die Schalter noch nicht zur Entwurfszeit angezeigt werden können und Sie Code schreiben müssen, um die Komponente mit dem Menü zu verknüpfen.

Um diese Komponente zu verwenden, wie im TreeDesigner demonstriert,

- benötigen Sie wieder eine *TMainMenu*-Komponente, in der Sie das Menü auf herkömmliche Art und Weise editieren können. Das Menü bleibt jedoch unsichtbar (das *Menu*-Property des Formulars bleibt also ungesetzt).
- Fügen Sie dann eine *TMenuToolbar*-Komponente an die gewünschte Position ein. Um das Hauptmenü in dieser Komponente anzuzeigen, setzen Sie das *MainMenu*-Property zur Laufzeit, also beispielsweise in *OnCreate* auf die *TMainMenu*-Komponente.

In einer MDI-Anwendung sind weitere Schritte notwendig, um die Menüverschmelzung zu aktivieren. *TMenuToolbar* verfügt über das Property *ChildMenu*, in dem Sie eine zweite *TMainMenu*-Komponente angeben können (üblicherweise die eines MDI-Kindfensters), die dann nach dem in Kapitel 5.7.3 beschriebenen Prinzip mit dem *MainMenu* verschmolzen wird. In einer MDI-Anwendung muss *ChildMenu* jedes Mal, wenn ein anderes Dokumentfenster aktiviert wird, auf das Menü dieses Fensters gesetzt werden. Wird das letzte Kindfenster geschlossen, muss *ChildMenu* auf *nil* gesetzt werden, damit die *MenuToolbar* nur noch das Hauptmenü des Hauptformulars anzeigt. Die entsprechenden Schritte können Sie auf der CD in den Methoden *WMMDIActivate* und *FormClose* der Klasse *TDocumentForm* nachvollziehen.

Auf Änderungen der Systemeinstellungen reagieren *R149*

Nun geht es zum Abschluss um die Anpassung der Toolbar-Schrift an die im System eingestellte Menüschriftart. Wenn Sie *TMenuToolbar* verwenden, ist dies am einfachsten: Sie stellt Ihnen die Methode *UpdateFont* zur Verfügung, die die Menü-Schriftart mit der Windows-API-Funktion *SystemParametersInfo* ermittelt. Da *TMenuToolbar* diese

Methode automatisch aufruft, brauchen Sie theoretisch gar nichts zu unternehmen. Es gibt jedoch eine gute Gelegenheit *UpdateFont* auch manuell aufzurufen, und zwar wenn die Systemeinstellungen während der Programmausführung geändert werden. Windows informiert alle Anwendungen über solche Änderungen, indem es die Botschaft *WM_SETTINGCHANGE* sendet. Sie können diese Nachricht in einem Delphi-Formular einfach abfangen, indem Sie das Ereignis *TApplication.OnSettingChange* bearbeiten (ab Delphi 6) oder auf Formularebene eine Methode wie die Folgende in die Deklaration des Formulars aufnehmen:

```
procedure SettingChange(var Msg: TMessage); message WM_SETTINGCHANGE;
```

Für die Aktualisierung der Toolbar-Schriftart kann diese Methode wie folgt implementiert werden (Auszug aus dem Testprogramm für die *TMenuToolbar*, auf der CD unter `Complib\MenuToolbarTest.dpr`):

```
procedure TForm1.SettingChange(var Msg: TMessage);
begin
  MenuToolbar1.UpdateFont;
  inherited; // Weitergabe an die Standardverarbeitung der Nachricht
end;
```

Es ist zwar theoretisch möglich, dass der Nachrichtenparameter *Msg.LParam* weitere Informationen darüber enthält, welche der vielen Systemeinstellungen überhaupt geändert wurde, jedoch empfiehlt die Win32-Dokumentation selbst, bei einem *WM_SETTINGCHANGE* alle von der Anwendung verwendeten Einstellung neu zu lesen und zu aktualisieren (dies gilt natürlich auch für das in Delphi 6 eingeführte *OnSettingChange*-Ereignis, das lediglich eine »Verpackung« für das *WM_SETTINGCHANGE*-Ereignis darstellt).

5.2.4 Komponenten für die Zeichenfläche

Da die wichtigste Aufgabe der Zeichenfläche, das Ausgeben der Grafik, in den Kapiteln 5.5 und 5.6 besprochen wird, werfen wir hier nur einen kurzen Blick auf den zur Entwurfszeit festgelegten Aufbau der Zeichenfläche.

Bereiche für die Grafikausgabe

Das Dokumentformular ist ein Beispiel für ein Formular, dessen *Canvas*-Zeichenfläche nicht für die Grafikausgabe in Frage kommt, da Teile des Formulars und damit der Zeichenfläche durch automatisch angeordnete Komponenten wie die Lineale belegt werden. Selbst wenn sich ein fester Bereich zur Grafikausgabe finden ließe, würde dieser vielleicht bei einer Weiterentwicklung der Anwendung verändert, wenn z.B. eine neue Leiste in das Fenster integriert werden würde. (Die VCL passt die Zeichenfläche des *Canvas*-Properties des Formulars nicht wie eine mit *alClient* ausgerichtete Komponente automatisch in den freien Bereich des Formulars ein.)

Aus diesem Grund werden Sie wahrscheinlich immer eine weitere Komponente in das Formular einfügen, die den Bereich für die Grafikausgabe klar definiert und die auch über ein eigenes *Canvas*-Property verfügt. Unter den Standardkomponenten bietet sich hierzu die Klasse *TPaintBox* an.

Im Formular des TreeDesigners sind zunächst einmal zwei einfache Lineal-Komponenten der Klasse *TRuler* eingebunden. *TRuler* wird auf der CD-ROM mitgeliefert und soll hier lediglich als »Black Box« betrachtet werden. Die Lineale werden später zur maßstabsgerechten Positionsanzeige beim Scrolling verwendet (Kapitel 5.5.4). An dieser Stelle genügt es zu wissen, dass die beiden Komponenten mit *alTop* bzw. mit *alLeft* an den oberen und den linken Rand des Formulars geheftet wurden.

Die *TPaintBox*-Zeichenfläche füllt den gesamten noch freien Bereich aus. Für den TreeDesigner genügt eine einfache Zeichenfläche jedoch noch nicht, denn sie wäre für größere Hierarchiegrafiken viel zu klein. Es müssen also noch eine horizontale und eine vertikale Bildlaufleiste hinzukommen, die sich in einem praktischen Zweierpack bereits in der Komponente *TScrollBox* befinden.

Es ist diese *TScrollbox*-Komponente, die den gesamten noch freien Platz des Formulars einnimmt, also mit *alClient* ausgerichtet ist. Die *TPaintBox*-Komponente ist dieser Scrollbox als Kindfenster untergeordnet. Ihre Größe ist im TreeDesigner standardmäßig auf 3000 mal 3000 Pixel eingestellt, so dass bei einem Vergrößerungsfaktor von 1 das gesamte Grafikdokument hineinpasst. Die ScrollBox stellt dazu automatisch passende Scrollbars bereit. Weitere Erläuterungen zum Scrolling und zur Vergrößerung der Grafikdarstellung überlassen wir den Kapiteln 5.5 und 5.6.

Hinweis: Der TreeDesigner verwendet für seine Scrollbox nicht die Klasse *TScrollBox*, sondern die in Kapitel 6.5.2 erläuterte *TScrollboxEx*.

5.2.5 Wichtige Menübefehle

Normalerweise unverzichtbare Bestandteile eines Dokumentformulars sind auch Menüpunkte, die das Speichern und Laden des Dokuments gestatten. Da *TGraphicDoc*, die Dokument-Klasse des TreeDesigners, ähnlich einfache Methoden zum Speichern und Laden zur Verfügung stellt wie verschiedene Klassen der VCL mit *SaveToFile* und *LoadFromFile*, bleiben die Ereignismethoden für die *OnClick*-Ereignisse der Menüpunkte sehr überschaubar.

Hier geht es um den Code der Menüpunkte SPEICHERN, SPEICHERN UNTER..., ÖFFNEN... und SCHLIESSEN. Grundlage für diese Methoden ist, dass sich das Programm merkt, welche Datei gerade bearbeitet wird. Nur so kann es die Datei sofort unter dem richtigen Namen speichern, ohne den Benutzer vorher mit dem *Speichern-unter*-Dialog danach zu fragen.

Speichern

Der TreeDesigner speichert den Dateinamen eines Dokuments im zugehörigen Dokumentobjekt. Das Dokument*fenster* kann über *TGraphicDoc.FileName* auf diesen zugreifen. Falls die Datei noch unbenannt ist, enthält dieses Property einen Leerstring.

Die *Speichern-unter*-Dialogbox wird aufgerufen, wenn der Benutzer den entsprechenden Menüpunkt aufruft oder eine bisher unbenannte Datei gespeichert wird. In beiden Fällen erledigt die folgende Methode den Aufruf:

```
procedure TDocumentForm.CmSaveAsClick(Sender: TObject);
{ OnClick für "Speichern unter..." }
begin
  SaveDialog1.FileName := Document.FileName;
  if SaveDialog1.Execute then begin
    Document.SaveToFile(Filename);
    // den neuen Dateinamen an die Liste der zurückliegenden
    // Dateien anhängen:
    FileList.AddString(SaveDialog1.FileName);
  end;
end;
```

Wenn der Benutzer den Menüpunkt SPEICHERN wählt, kann die Datei sofort gespeichert werden, falls sie schon einen Namen hat. Ansonsten wird einfach die eben gezeigte Methode für den *Speichern-unter*-Dialog aufgerufen:

```
procedure TDocumentForm.CmSaveClick(Sender: TObject);
{ OnClick für "Speichern" }
begin
  { Testen, ob Datei schon benannt ist: }
  if Document.FileName <> '' then
    { Wenn ja, unter diesem Namen speichern }
    Document.SaveToFile(Document.FileName);
  { wenn nicht, Methode für "Speichern unter" aufrufen: }
  else CmSaveAsClick(Sender);
end;
```

Aktualisierung von Dateiname, Fenstertitel und Änderungsflag

Beim Laden oder Speichern eines Dokuments müssen normalerweise drei Dinge berücksichtigt werden:

- Eine Flag oder ein Zähler, der speichert, ob an dem Dokument Änderungen vorgenommen worden sind, die noch nicht gespeichert wurden. Dieses Flag bzw. der Zähler muss sowohl beim Laden einer neuen Datei als auch beim Speichern zurückgesetzt werden, so dass beim Schließen des Fensters keine unnötige Sicherheitsabfrage durchgeführt wird.

- Der Dateiname muss intern gespeichert werden, damit der Benutzer ihn beim nächsten Speichern nicht noch einmal angeben muss.
- Der Fenstertitel muss aktualisiert werden, sofern er den Dateinamen angibt.

In den obigen Methoden gab es von alledem nichts zu sehen, weil Dateiname und Änderungszähler im Dokumentobjekt verwaltet werden, die genannten Dinge werden oben also innerhalb von *Document.SaveToFile* berücksichtigt. Das Dokumentobjekt veranlasst die Aktualisierung des Fenstertitels bei einem geänderten Dateinamen und enthält auch einen Änderungszähler, den es bei jeder Änderung am Dokument erhöht.

Damit das Dokument auch alle Änderungen kontrollieren kann, deklariert es alle Variablen als *private* und lässt Zugriffe darauf nur über Methoden zu, in denen dann unter anderem auch der Änderungszähler richtig gesetzt wird. Beispiele für solche Methoden gibt Kapitel 5.3.2.

Hinweis: Die Dokumentklasse *TGraphicDoc* ist von der Klasse *TList* abgeleitet. Damit auch in den *TList*-Methoden *Add* und *Delete* der Änderungszähler berücksichtigt wird, muss *TGraphicDoc* die geerbten Methoden *Add* und *Delete* überschreiben und das geerbte Verhalten (ohne Änderungszähler) entsprechend ergänzen. Eigentlich müsste *TGraphicDoc* sogar *alle* den Listeninhalt ändernden Methoden von *TList* überschreiben und durch die Änderungszählung erweitern; im TreeDesigner werden jedoch nur *Add* und *Delete* von außen aufgerufen, und so wurde auf diesen Schritt verzichtet.

Der Dateiname des Dokuments (*TGraphicDoc.FileName*) wird innerhalb der Methoden *SaveToFile* und *LoadFromFile* auf den als Parameter angegebenen Dateinamen gesetzt. Bleibt noch die Frage, wie der Dateiname in den Fenstertitel gelangt. *TGraphicDoc* speichert für das Dokumentfenster einen Zeiger namens *UpdateCaptionProc*, der auf eine Formularmethode weist, die den Titel aktualisiert. Da diese Methode im TreeDesigner sehr stark mit dem Dokument-View-Konzept zusammenhängt, wird sie erst in Kapitel 5.3.1 besprochen.

Lesen von Dateien

Das Lesen von Dateien funktioniert ähnlich simpel wie das Schreiben, als Dialog kommt ein Exemplar von *TOpenDialog* zum Einsatz:

```
procedure TDocumentForm.LoadFromFile(NewFileName: String);
begin
  Document.FreeAll; // alle Objekte löschen
  Document.LoadFromFile(NewFileName); // neue Objekte laden
  Invalidate; // bewirkt die Neuausgabe der Grafik
end;
```

```
procedure TDocumentForm.CmLoadClick(Sender: TObject);
{ OnClick für "Öffnen..." }
begin
  if OpenDialog1.Execute then
    LoadFromFile(OpenDialog1.FileName);
end;
```

Schließen

Obwohl man ein Windows-Fenster über Alt+F4 und über einen Klick auf das Schließen-Feld schließen kann, verzichtet kaum ein Dokumentfenster auf einen eigenen Menüpunkt SCHLIESSEN im Dateimenü. Die zugehörige Methode ist mit sechs Zeichen Inhalt die kürzeste in diesem Buch:

```
procedure TDocumentForm.CmCloseClick(Sender: TObject);
begin
  Close;
end;
```

Der Aufruf von *Close* bewirkt indirekt, dass es zu einer Sicherheitsabfrage kommt, falls die Datei nicht gespeichert wurde. Das damit zusammenhängende *OnCloseQuery*-Ereignis ist ein Thema des nächsten Kapitels.

5.2.6 Typische Formularaufgaben

Um das allgemeine Management des Dokumentformulars abzuschließen, behandelt dieses Kapitel

- die Sicherheitsabfrage vor dem Schließen eines ungespeicherten Dokuments (eine der nahezu unverzichtbaren Funktionen),
- die Anpassung von Menü-Properties zur Laufzeit und
- Popup-Menüs für lokal anwendbare Funktionen.

Die Sicherheitsabfrage *R60*

Durch die marktüblichen Anwendungen ist der Anwender bereits so stark an die Sicherheitsabfrage vor dem Schließen einer ungespeicherten Datei gewöhnt, dass es sich eine Anwendung kaum leisten kann, darauf zu verzichten. In Delphi lässt sich eine solche Sicherheitsabfrage sehr schnell mit Hilfe des *OnCloseQuery*-Events realisieren.

Als Erstes benötigen Sie jedoch eine Variable, in der Sie speichern, ob das aktuelle Dokument bereits verändert wurde. Beim Speichern setzen Sie diese Variable auf *False* zurück, bei jeder Änderung wieder auf *True*. Der TreeDesigner verwendet statt einer booleschen Variable einen Zähler namens *Changed*, der größer als Null ist, falls die Datei seit der letzten Speicherung verändert wurde.

Das *OnCloseQuery*-Ereignis tritt immer auf, wenn der Benutzer versucht, ein Fenster zu schließen, oder wenn dieses Fenster indirekt geschlossen wird, weil die Anwendung oder sogar die gesamte Windows-Sitzung beendet wird. Die Methode für das *OnCloseQuery*-Event erhält von der VCL einen booleschen Variablenparameter *CanClose*, der per Voreinstellung mit *True* vorbelegt ist, was bedeutet, dass das Formular geschlossen werden kann.

Falls die Funktion *IsChanged* des Dokuments auf eine ungespeicherte Änderung hinweist, startet die Methode *FormCloseQuery* des TreeDesigners mit der *MessageDlg*-Funktion der *Dialogs*-Unit eine Anfrage, ob die Änderungen gespeichert werden sollen. Der Benutzer hat die Wahl zwischen den Schaltern *mbYes*, *mbNo* und *mbCancel*. Wählt er *Ja*, so simuliert die Methode den Aufruf des Menüpunkts DATEI | SPEICHERN, indem sie die Methode *CmSaveClick* aufruft:

```
procedure TDocumentForm.FormCloseQuery(Sender: TObject;
            var CanClose: Boolean);
var
  Answer: Integer;
  DocName: string;
begin
  if Document.IsChanged then begin
    { "DocName" gibt im MessageDlg-Text den Dateinamen an: }
    DocName := Document.FileName;
    if DocName = '' then DocName := 'unbenannter Datei';
    Answer := MessageDlg(
      Format('Soll %s gespeichert werden?', [DocName]),
      mtConfirmation, [mbYes, mbNo, mbCancel], 0);
    case Answer of
      mrYes: begin
        try
          CmSaveClick(Self);
        except
          on E: Exception do begin
            Application.HandleException(E);
            CanClose := False;
          end;
        end;
        { testen, ob im Sichern-Dialog Abbruch gedrückt wurde -
          kann an der Variable "Change" erkannt werden: }
        CanClose := (Changed = 0);
      end;
      mrCancel: CanClose := False;
      else CanClose := True;
    end;
  end;
end;
```

FormCloseQuery erlaubt nur dann das Schließen des Fensters, wenn die Datei gar nicht verändert wurde, wenn der Benutzer den Schalter *mbNo* gewählt hat oder wenn er den Schalter *mbYes* gewählt hat *und* die Variable *Changed* durch das Speichern auf Null zurückgesetzt wurde. Ist *Changed* trotz Drückens des Ja-Schalters ungleich Null, so bedeutet das, dass *CmSaveClick* zwar den *Speichern unter*-Dialog aufgerufen hat, dass der Benutzer in diesem jedoch *Abbruch* gedrückt hat.

Schließlich kann es auch noch vorkommen, dass die Speichern-Funktion aufgerufen wurde, es dabei aber zu einer Exception gekommen ist. Die obige Methode fängt alle Exceptions ab und zeigt dem Benutzer über *Application.HandleException* eine Meldung darüber an. Außerdem verhindert sie das Schließen des Fensters, indem Sie *CanClose* auf *False* setzt (dabei wurde angenommen, dass eine Exception meist bedeutet, dass die Datei nicht korrekt gespeichert werden konnte; dies könnte bei Interesse natürlich über den Typ der Exception genauer abgefragt werden: z.B. kann es zu einer *EFCreateError*-Exception kommen, wenn die Datei in einem Verzeichnis erzeugt wird, für das keine Schreibberechtigung vorliegt).

Ohne diesen Exception-Handler würde die Methode, falls es beim Speichern zu einer Exception kommt, einfach abgebrochen, das Fenster würde geschlossen werden und der Benutzer hätte eventuell seine Änderungen am Dokument verloren.

Menümarkierung und -deaktivierung

Um die Markierung eines Menüpunkts, wie z.B. des Menüpunkts ANSICHT | GERÄTEKOORDINATENSYSTEM, ein- und auszuschalten, muss zur Laufzeit das Property *Checked* geändert werden. Ab Delphi 6 wird dies automatisch erledigt, wenn Sie das *AutoCheck*-Property des Menüpunkts einschalten. In früheren Delphi-Versionen erledigen Sie dies mit einer Zeile Code selbst: Den Namen des Menüpunkts, im Folgenden *NoMapMode*, erfahren Sie am schnellsten aus dem Objektinspektor, wenn Sie den Menüpunkt im Menüeditor ausgewählt haben. Die folgende Zeile kehrt also den bisherigen Markiert-Status des genannten Menüpunkts um:

```
NoMapMode.Checked := not NoMapMode.Checked;
```

Entsprechend der Deaktivierung von Steuerelementen (siehe Kapitel 3.3.1) sollten auch Menüpunkte manchmal deaktiviert werden, falls sie gerade nicht sinnvoll anzuwenden sind. Auch bei Menüpunkten heißt das entsprechende Property *Enabled*.

Aktionslisten im TreeDesigner *R20*

Im TreeDesigner gibt es nur ein Menü, in dem Menüpunkte deaktiviert werden, das Menü *Ansicht*. Wenn Sie die Option ANSICHT | GERÄTEKOORDINATENSYSTEM einschalten, sollen unter anderem die Menüpunkte zum Vergrößern und Verkleinern sowie der entsprechende Scrollbar in der Werkzeugleiste deaktiviert werden. Deaktiviert werden

insbesondere auch die Menüpunkte SEITENANZEIGE und 100%, die in einer Toolbar durch Schalter repräsentiert sind, welche ebenfalls deaktiviert werden müssen.

Zur gleichzeitigen Behandlung von Menüpunkten und zugehörigen Toolbar-Schaltern verwendet der TreeDesigner für das Ansichts-Menü eine Aktionsliste (zum Aufbau von Aktionslisten siehe Kapitel 4.6.4). Statt die *Checked*- und *Enabled*-Properties der Menüpunkte und Schalter direkt zu setzen, setzt er einfach die gleichnamigen Properties der entsprechenden Aktion (mit welcher Menüpunkte und Schalter über das *Action*-Property verknüpft sind). Statt wie in der oben gezeigten Zeile auf *NoMapMode* greift er beispielsweise auf *ANoMapMode*, das zugehörige *TAction*-Objekt der Aktionsliste, zu:

```
procedure TDocumentForm.NoMapModeClick(Sender: TObject);
begin
{ Aktualisierung von Menü UND Toolbars durch Aktionen: }
  { Markierungs-Flag umkehren: }
  ANoMapMode.Checked := not ANoMapMode.Checked;
  { davon abhängige Aktionen aktivieren oder deaktivieren: }
  ASeitenAnzeige.Enabled := not ANoMapMode.Checked;
  AMaximaleVerkleinerung.Enabled := not ANoMapMode.Checked;
  AN100Prozent.Enabled := not ANoMapMode.Checked;
  AN200Prozent.Enabled := not ANoMapMode.Checked;
  { davon abhängiges Steuerelement aktivieren oder deaktivieren: }
  ZoomFactorBar.Enabled := not NoMapMode.Checked;
  { Durchführen des eigentlichen Menübefehls
    (Gerätekoordinatensystem einschalten): }
  if NoMapMode.Checked then begin
    FZoomFactor := 1;
    ZoomFactorBar.Position := 10;
  end;
  ResizePaintbox;
end;
```

Diese Vorgehensweise birgt gute Erweiterungsmöglichkeiten in sich: Auch wenn bisher nur zwei der Menüpunkte durch Toolbar-Schalter dupliziert werden, können in Zukunft auch alle anderen Punkte in der Toolbar repräsentiert werden. Wenn die neuen Schalter dann mit den bestehenden Aktionen verknüpft werden, muss kein neuer Code geschrieben werden, um die Schalter zu (de-)aktivieren oder gedrückt darzustellen (ein Schalter für einen markierten Menüpunkt wird gedrückt, also mit *Down*-Property = *True* dargestellt).

Optionen in Popup-Menüs

Popup-Menüs geben dem Anwender einen viel schnelleren Zugriff auf bestimmte Optionen als mehrseitige Dialogboxen oder geschachtelte Hauptmenüs, die erst nach der richtigen Option durchsucht werden müssen. Die Delphi-IDE gibt mit ihren lokalen Menüpunkten HINWEISE ANZEIGEN (für die Komponentenpalette) und IMMER IM

VORDERGRUND (für den Objektinspektor) selbst ein Beispiel für Einstellungen, die sehr nützlich sein können, aber nicht über das Hauptmenü vorgenommen werden können.

Da Sie ein Popup-Menü unter Delphi so schnell anlegen können wie ein Hauptmenü und da Popup-Menüs das Programm auch aus der Sicht des Programmierers übersichtlicher werden lassen, steht dem intensiven Gebrauch dieser Menüs eigentlich nichts im Wege.

Fügen Sie für die Komponenten Ihres Formulars, die ein Popup-Menü besitzen sollen, jeweils eine *PopupMenu*-Komponente von der ersten Seite der Komponentenpalette in das Formular ein, tragen Sie dieses Popup-Menü im *PopupMenu*-Property der Komponente ein, editieren Sie das Menü im Menüdesigner und verknüpfen Sie seine Punkte durch einen Doppelklick mit Programmcode.

Als kurzes Beispiel soll hier der Menüpunkt SCHRIFT WECHSELN... der Lineale im Dokumentfenster des TreeDesigners genügen. Nach seiner Betätigung wird ein Schriftauswahl-Standarddialog aufgerufen, durch den der Benutzer die Schrift beider Lineale (*VRuler* und *HRuler*) gleichzeitig ändert:

```
procedure TDocumentForm.CmChangeRulerFontClick(Sender: TObject);
begin
  FontDialog1.Font := VRuler.Font;
  if FontDialog1.Execute then begin
    VRuler.Font := FontDialog1.Font;
    HRuler.Font := FontDialog1.Font;
  end;
end;
```

Eine weitere Voraussetzung für den automatischen Aufruf eines Popup-Menüs ist, dass Sie sein *AutoPopup*-Property beim voreingestellten Wert *True* belassen.

Kapitel 5.8.3 beschreibt eine etwas kompliziertere Anwendung von Popup-Menüs: Die Zeichenfläche besitzt nicht nur ein, sondern zwei Popup-Menüs und muss daher auf den automatischen Aufruf durch die VCL verzichten.

5.3 Das Grafikdokument

Dialogfenster und viele andere Formulare stellen oft ihren gesamten Dateninhalt in einzelnen Komponenten dar. Ein Formular, dessen Inhalt eine Stringliste und ein zusätzlicher einzelner String ist, kann diese beiden Elemente beispielsweise in einer *TStringGrid*- und einer *TEdit*-Komponente speichern (ein Beispiel dafür ist der CD-Spieler aus Kapitel 3.6.4, der einen CD-Titel und eine Liste der einzelnen Spuren enthält).

Sobald der Inhalt eines Formulars nicht mehr nur aus dem Inhalt seiner Komponenten besteht, empfiehlt es sich, eine eigene Klasse für das Dokument anzulegen, die auch eine Methode zur Verfügung stellt, mit der das gesamte Dokument gespeichert werden kann.

5.3.1 Das Dokument-View-Konzept

Im Dokument-View-Konzept sorgt eine eigenständige Dokumentklasse dafür, dass alle zum Dokument gehörenden Daten von den Daten des Formulars (dem *View*) deutlich getrennt sind. So gehört beispielsweise die Information über den gerade eingestellten Vergrößerungsfaktor eher zum Formular als zum Dokument und könnte statt in der Dokumentdatei in einer Optionsdatei gespeichert werden. Ein Beispiel dafür gibt die Delphi-IDE selbst: Sie speichert die Projektoptionen weder im Pascal-Quelltext noch in der Projektdatei, sondern in eigenen Optionsdateien mit der Endung `.dof`.

Eine Trennung von Dokument und View wirkt sich nicht nur positiv auf die Verständlichkeit und Wartbarkeit eines Programms aus, sondern schafft auch die Grundlagen für weitere gern gesehene Funktionen von MDI-Anwendungen: So freuen sich Benutzer oft darüber, wenn sie dasselbe Dokument in verschiedenen Fenstern gleichzeitig ansehen können und sie in jedem Fenster eine andere Ansicht erhalten können (z.B. einen anderen Vergrößerungsfaktor, eine andere Perspektive, einen anderen Ausschnitt). Wenn jedoch die Daten des Dokuments fest in das Formular integriert sind, ist dem Benutzer diese Möglichkeit ohne programmiertechnische Verrenkungen kaum zu eröffnen.

Das Dokument-View-Konzept im TreeDesigner

In der MDI-Version des TreeDesigners können Sie ein Dokument in mehreren Views, also MDI-Kindfenstern, ansehen. Änderungen am Inhalt des Dokuments wie beispielsweise Farbänderungen von Grafikobjekten oder das Einfügen von neuen Objekten werden sofort in allen geöffneten Views sichtbar. Die Position der Bildlaufleisten und der Vergrößerungsfaktor wird für jedes Fenster getrennt verwaltet, so dass z.B. eine Vergrößerung der Dokumentdarstellung im ersten View nicht automatisch auch eine Vergrößerung im zweiten View verursacht. Dies erlaubt Ihnen, gleichzeitig verschiedene Ausschnitte der Grafik in verschieden Vergrößerungsstufen darzustellen.

Eine erfreuliche Nebenwirkung des Dokument-View-Konzepts ist, dass sich damit ganz leicht ein Übersichtsfenster realisieren lässt, in dem der TreeDesigner eine Gesamtansicht des jeweils aktuellen Dokuments zeigt, zusammen mit einer Andeutung des gerade sichtbaren Ausschnitts (siehe Abbildung 5.1). Dafür muss im Prinzip nur ein fast leeres Übersichtsfenster definiert werden, das dann bei jedem Dokumentwechsel als View zum aktuellen Dokument hinzugefügt wird.

Hinweis: Das MDI-Konzept ist unabhängig vom Dokument-View-Konzept und wird erst in Kapitel 5.7 behandelt. Im Folgenden werden daher die MDI-Kindfenster nur als »Dokumentfenster«, das MDI-Hauptfenster einfach als »Hauptfenster« bezeichnet.

Realisierung des Dokument-View-Konzepts *R42*

Die Grundlage der Trennung von Dokument und View besteht natürlich darin, dass diese beiden Objektarten in getrennten Klassen deklariert werden. Während die Formularklasse für das View zuständig ist, definiert der TreeDesigner die Klasse für das Dokument in einer eigenen Unit namens *GrDoc*. Die Klasse heißt *TGraphicDoc* und wird in Kapitel 5.3.2 detaillierter beschrieben. An dieser Stelle soll es nur um die Dokument-View-Eigenschaften dieser Klasse gehen.

Abbildung 5.3 zeigt die Zusammenhänge zwischen den TreeDesigner-Klassen, die an der Realisierung des Dokument-View-Konzepts beteiligt sind. Wäre der TreeDesigner nach dem Prinzip »ein Fenster – ein Dokument« aufgebaut, genügte es, mit jedem Kindfenster ein eigenes Dokument zu speichern (also jeweils ein Dokument zum »Besitz« des Formulars zu machen). Das Dokument-View-Konzept macht es erforderlich, eine von allen Views getrennte Liste von Dokumenten zu verwalten – naheliegenderweise im Hauptfenster.

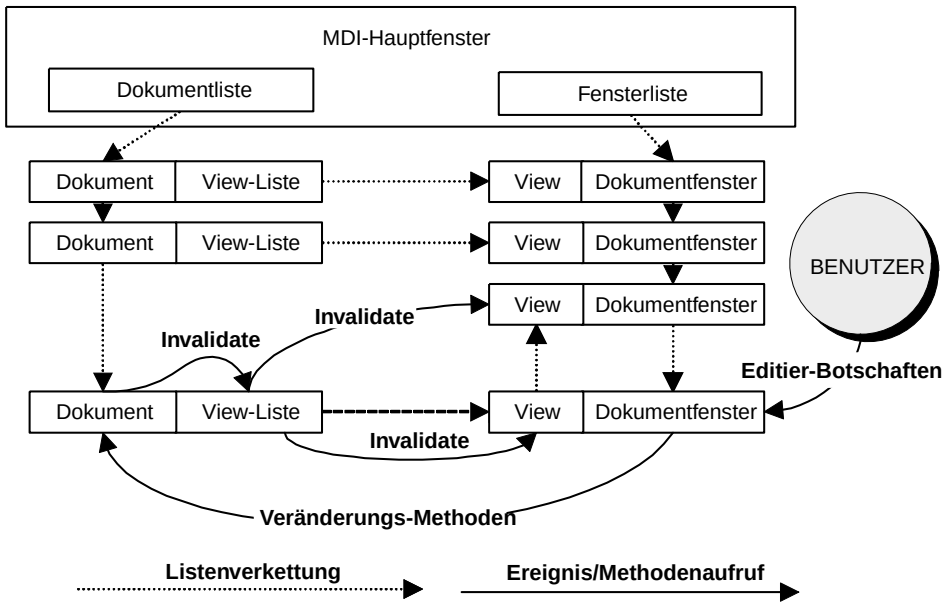


Abbildung 5.3: Eine Editieraktion des Benutzers löst eine Kette von Ereignissen und Methoden aus, an deren Ende erst die sichtbare Veränderung am Bildschirm steht.

Jedes dieser Dokumente verfügt über eine Liste von Views, bei denen es sich prinzipiell um die einzelnen Dokumentfenster handeln könnte. Im TreeDesigner wurde jedoch Wert darauf gelegt, dass die Dokumentklasse möglichst unabhängig von den Formularklassen ist – ein direkter Verweis vom Dokument auf ein Dokumentfenster sollte also vermieden werden. Daher wurde eine dritte Klasse, *TGraphicView* eingeführt, die vom Dokumentfenster initialisiert werden muss und alle View-Daten enthält, die von der Dokumentklasse benötigt werden. *TGraphicView* ist wie *TGraphicDoc* in der Unit *GrDoc* deklariert und unabhängig von der Formularklasse *TDocumentForm*.

Um das Zusammenwirken dieser verschiedenen Objekte und Klassen zu verdeutlichen, zeigt Abbildung 5.3 auch, was beim Editieren des Dokuments durch den Benutzer abläuft. Angenommen, er möchte einem Teil der Grafik eine neue Farbe zuweisen und bedient sich dazu der Technik des Drag&Drop. Für die Anwendung bedeutet dies letztendlich, dass ein Drop-Ereignis im Formular eintrifft (in der Abbildung als *Editier-Botschaft* dargestellt). Das Formular interpretiert diese als Farbänderungswunsch und ruft eine entsprechende Methode des Dokuments auf. Das Dokument ändert daraufhin in seinen Datenstrukturen die Farbe des betroffenen Grafikelements (bzw. der betroffenen Elemente). Jetzt muss noch dafür gesorgt werden, dass diese Änderung auch am Bildschirm sichtbar wird. Dafür muss das Dokument die Liste der mit ihm verbundenen Views durchgehen und jedes einzelne View zum Neuzeichnen der veränderten Bereiche veranlassen. Die hierzu verwendete Technik des Ungültig-Erklärens (*Invalidate*) wird in Kapitel 5.5 näher beschrieben.

Quelltext-Auszüge aus der Haupt-Formularklasse

Wie angekündigt wird die Liste der Dokumente im TreeDesigner in der Klasse des Hauptfensters deklariert. Die folgenden Zeilen zeigen die Deklaration der Liste und die für ihre Verwaltung wichtigen Methoden:

```
TMainForm = class(TForm)
// Ausschnitt: Doc-View-relevante Klassenelemente
  private
    FDocumentList: TList;
  public
    function FindDoc(FileName: String): TGraphicDoc;
    procedure DeleteDoc(Document: TGraphicDoc);
    function GetDoc(FileName: String): TGraphicDoc;
    function NewWindow(Document: TGraphicDoc;
                       ReuseEmptyWindow: Boolean = False): TDocumentForm;
```

Die Elemente der Liste sind die Dokumentobjekte, haben also den Typ *TGraphicDoc* und können mit *FDocumentList.Add(Document)* in die Liste eingefügt werden.

Damit nicht dieselbe Datei in mehreren verschiedenen Dokumentobjekten geöffnet wird, ist es Aufgabe der Methode *FindDoc* festzustellen, ob bereits ein Dokument eines gegebenen Dateinamens existiert:

```
function TMainForm.FindDoc(FileName: String): TGraphicDoc;
var
  i: Integer;
begin
  Result := nil;
  for i := 0 to FDocumentList.Count-1 do
    if TGraphicDoc(FDocumentList[i]).FileName = FileName then begin
      Result := TGraphicDoc(FDocumentList[i]);
      exit;
    end;
end;
```

Aufgabe der Methode *GetDoc* ist es dann, ein Dokumentobjekt zu einem gegebenen Dateinamen zurückzuliefern. Wenn bereits ein solches in der Dokumentliste existiert, wird dieses wiederverwendet, ansonsten wird ein neues erzeugt und aus der Datei geladen (zur Methode *LoadFromFile* siehe Kapitel 5.3.3):

```
function TMainForm.GetDoc(FileName: String): TGraphicDoc;
var
  Document: TGraphicDoc;
begin
  if FileName = '' then Document := nil // leeres Dokument gefordert
    else Document := FindDoc(FileName);
  if Document = nil then begin        // Dokument noch nicht geladen
    Document := TGraphicDoc.Create;
    if FileName <> '' then begin
      Document.LoadFromFile(FileName);
      // Einfügen in die Liste der zurückliegenden Dokumente
      // (siehe Kapitel 4.6.2, R46):
      FileList.AddString(OpenDialog1.FileName);
    end;
    FDocumentList.Add(Document);
  end;
  Result := Document;
end;
```

GetDoc wird beispielsweise beim Öffnen einer neuen Datei aufgerufen:

```
procedure TMainForm.CMFileOpen(var Msg: TMessage);
var
  Document: TGraphicDoc;
  NewChild: TDocumentForm;
begin
  if OpenDialog1.Execute then begin
    Document := GetDoc(OpenDialog1.FileName);
    NewChild := NewWindow(Document, True);
  end;
end;
```

Die Methode *NewWindow* gibt das gefundene *Document* an den Konstruktor des Dokumentfensters weiter und dieser speichert einen Verweis darauf in den Dokumentfenster-Daten. Eine detaillierte Darstellung dieser Vorgänge würde sich an dieser Stelle nicht lohnen; falls Sie sich dafür interessieren, ist ein schrittweises Verfolgen mit dem Debugger empfehlenswert.

Freigabe der Dokumente

Die Dokumente, die in *GetDoc* zur Liste hinzugefügt werden, müssen natürlich auch irgendwann aus der Liste gelöscht werden. Das Dokumentfenster selbst darf sein Dokument nicht löschen, da es nicht wirklich »sein« Dokument ist, sondern dieses noch in anderen Views dargestellt werden könnte. Vielmehr ist es Aufgabe des Hauptformulars, jedes Mal, wenn ein Kindfenster geschlossen wird, zu überprüfen, ob noch weitere Views an das Dokument angeschlossen sind. Nur wenn das nicht der Fall ist, darf das Hauptfenster das Dokumentobjekt freigeben.

Bei der Beantwortung der Frage, wie das Hauptfenster vom Schließen eines Kindfensters erfährt, soll im TreeDesigner wieder das Prinzip beachtet werden, dass das Kindfenster möglichst unabhängig vom Hauptfenster sein soll. Also ruft es nicht direkt eine Methode des Hauptfensters auf, sondern stellt ein Event zur Verfügung, über welches das Hauptfenster sich bei Bedarf vom Schließen des Fensters informieren lassen kann. Das Event heißt *OnDestroyDocWin* und das Hauptfenster verknüpft es nach jedem Erzeugen eines Kindfensters in *TMainForm.NewWindow* mit seiner folgenden Methode:

```
procedure TMainForm.ChildWindowDestroy(Sender: TObject);
begin
  // leicht gekürzte Version
  // (zur Verwaltung des Übersichtsfensters siehe CD-ROM)
  with Sender as TDocumentForm do begin
    if Assigned(Document) and (Document.ViewCount = 0) then
      DeleteDoc(Document);
  end;
end;
```

Die Erzeugung dieses Events funktioniert nach dem üblichen Muster (Kapitel 6.3.3) und soll hier nicht weiter erläutert werden.

Die Daten eines Views und deren Initialisierung

Wie schon erwähnt werden alle Daten des Dokumentfensters, die im Grafikdokument benötigt werden, zu der eigenen Klasse *TGraphicView* zusammengefasst:

```
TGraphicView = class
  Form: TForm; // Das Formular, in dem das Dokument dargestellt wird
  PaintBox: TPaintBox; // Zur Darstellung benutzte PaintBox
  ViewNumberCaption: String; // Nummer des Views für die Titelzeile
  // Die folgenden drei Variablen sind Zeiger auf
```

```
    // Methoden des Formulars, die vom Dokument oder den darin
    // enthaltenen Grafikelementen aufgerufen werden:
    SetMapModeProc: TSetMapModeProc; // (Kapitel 5.6)
    GetGripSizeFunc: TGetGripSizeFunc; // für die "Rahmengriffe"
    UpdateCaptionProc: TSetFileNameProc;
  end;
```

Es ist die Aufgabe des Dokumentformulars, eine solche Struktur zu erzeugen, seine »persönlichen« Daten darin zu speichern und die Struktur schließlich an das Dokumentobjekt weiterzugeben:

```
constructor TDocumentForm.Create(AOwner : TComponent; ...;
                                 ADocument: TGraphicDoc);
begin
  FView := TGraphicView.Create;
  FView.Form := self;
  FView.ViewNumberCaption := '';
  FView.SetMapModeProc := SetMapMode;
  FView.GetGripSizeFunc := GetGripSize;
  FView.UpdateCaptionProc := UpdateCaption;
  Document := ADocument; // Bewirkt den Aufruf von SetDocument
  ...
// Die PaintBox kann erst später beim OnCreate-Ereignis gesetzt werden:
procedure TDocumentForm.FormCreate(Sender: TObject);
begin
  ...
  FView.PaintBox:=PaintBox;

// Das Property Document wird durch folgende Methode beschrieben:
procedure TDocumentForm.SetDocument(NewDoc: TGraphicDoc);
begin
  ...
  FDocument := NewDoc;
  Document.AddView(FView); // Weitergabe der Struktur an das Dokument
```

Nach diesem *AddView*-Aufruf sind die Verknüpfungen zwischen Dokument und View vollständig aufgebaut und können verwendet werden, wie im nächsten Abschnitt am Beispiel von *TGraphicElement.InvalidateAllViews* gezeigt wird.

Hinweis: Die durch *AddView* hergestellte Verknüpfung wird am Ende innerhalb der Methode *TDocumentForm.FormDestroy* (Ereignis *TForm.OnDestroy*) wieder gelöst. Hierzu wird analog zu *Document.AddView* die Methode *Document.DeleteView* verwendet.

View-Verwaltung im Dokument

Die Dokumentklasse enthält ein *TList*-Objekt zur Speicherung aller mit ihm verbundenen Views. Es wird wie üblich im Konstruktor und Destruktor initialisiert (hier nicht

gezeigt) und erhält einige Hilfsmethoden, die ebenfalls so einfach sind, dass ihre Implementierung nicht abgedruckt zu werden braucht:

```
TGraphicDoc = class(TList)
  // Ausschnitt: Doc-View-relevante Klassenelemente
private
  FViews: TList;  { darstellende Views }
  function GetViews(Index: Integer): TGraphicView;
  function GetViewCount: Integer;
public
  procedure AddView(AView: TGraphicView);
  procedure DeleteView(AView: TGraphicView);
  property Views[Index: Integer]: TGraphicView read GetViews;
  property ViewCount: Integer read GetViewCount;
```

Eine beispielhafte Anwendung der View-Liste ist die Methode, die das Neuzeichnen aller mit dem Dokument verbundenen Views bewirkt. Es ist allerdings keine Methode des Grafikdokuments, sondern die eines einzelnen Grafik*elements*, das aber über sein Property *Document* auf das Grafikdokument und dessen View-Liste zugreifen kann:

```
procedure TGraphicElement.InvalidateAllViews;
var
  R : TRect;
  View: TGraphicView;
  i: Integer;
begin
  for i := 0 to Document.ViewCount-1 do begin
    View := Document.Views[i];
    GetPaintRect(R, View.GetGripSizeFunc);
    InvalidateRect(R, View);
  end;
end;
```

Das Ungültigerklären selbst wird in Kapitel 5.5.3 erläutert.

Setzen der Fenstertitel

Ein weiterer Fall, in dem die View-Liste benötigt wird, tritt ein, sobald die Titelzeilen der Views aktualisiert werden müssen, etwa weil der zweite von vier Views geschlossen wurde und die im Titel angegebenen View-Nummern der letzten beiden Views um eins verringert werden müssen. In jedem Fall muss der Titel angepasst werden, wenn sich der Dateiname des Dokuments ändert. Die Titelanpassung findet daher in der Property-Schreib-Methode für das *FileName*-Property von *TGraphicDoc* statt:

```
procedure TGraphicDoc.SetFileName(const Value: string);
var
  i: Integer;
begin
  FFileName := Value;
```

```
    for i:=0 to FViews.Count - 1 do
      with TGraphicView(FViews[i]) do
        if Assigned(UpdateCaptionProc) then
          UpdateCaptionProc(Value, FViews);
end;
```

SetFileName ruft also lediglich für jeden View die in der *TGraphicView*-Struktur gespeicherte Methode zur Aktualisierung des Titels auf[11]. Auch deren Implementation in *TDocumentForm* soll im Folgenden noch kurz betrachtet werden. Sie bekommt nämlich im zweiten Parameter die View-Liste übermittelt, damit sich das Dokumentfenster selbst in dieser Liste finden kann, um seine Nummer darin festzustellen. Diese Nummer wird allerdings nur dann im Titel angezeigt, wenn es mehr als ein Dokumentfenster in der Liste gibt (da auch das Übersichtsfenster als View in die Liste eingetragen wird, muss der Typ von *Form* auf *TDocumentForm* überprüft werden):

```
procedure TDocumentForm.UpdateCaption(NewFileName: string;
                                      ViewList: TList);
var
  i, ViewIndex, ViewCount: Integer;
begin
  ViewIndex := 0;
  ViewCount := 0;
  for i := 0 to ViewList.Count-1 do begin
    with TGraphicView(ViewList.Items[i]) do
      if Form is TDocumentForm then begin
        inc(ViewCount);
        if Form = self then ViewIndex := ViewCount;
      end;
  end;
  if ViewCount = 1 then FView.ViewNumberCaption := ''
    else FView.ViewNumberCaption := ' ['+IntToStr(ViewIndex)+']';
  // und nun Viewnummer und Dateinamen zusammensetzen:
  if (NewFileName='') then
    FCaptionVorgemerkt := 'unbenannt'+FView.ViewNumberCaption
  else FCaptionVorgemerkt := NewFileName+FView.ViewNumberCaption;
  if Showing then
    Caption := FCaptionVorgemerkt; // sonst kann es hier zu einer
      // Exception kommen, wenn das Formular noch nicht erzeugt ist.
end;
```

Falls noch kein Name für die Datei existiert, setzt diese Methode den Titel auf »unbenannt« , ansonsten auf den Dateinamen. (Bei einer Anwendung, in der das Dokumentfenster gleichzeitig auch das Hauptfenster ist, würde normalerweise vor dem Dokumentnamen noch der Name der Anwendung genannt werden.)

11 *UpdateCaptionProc* könnte auch *OnUpdateCaption* genannt werden, denn im Prinzip handelt es sich dabei um nichts weiter als um ein Ereignis des Dokuments, für das jedes Fenster, das dieses Dokument darstellt, eine Methode bereitstellen sollte.

Weitere Funktionen einer Dokument-View-Anwendung

Abschließend sei noch auf zwei weitere Besonderheiten hingewiesen, die im TreeDesigner implementiert sind, hier aber aus Platzgründen nicht detailliert mit Code-Auszügen besprochen werden können:

- Eine LISTE DER GEÖFFNETEN DOKUMENTE können Sie aus dem Menü ANSICHT aufrufen. Ein einfaches Dialogformular mit einer Listbox und eine Schleife, in der das Hauptformular seine *FDocumentList* durchläuft, sind die Basis für diese Funktion.
- Die in Kapitel 5.2.6 gezeigte *OnCloseQuery*-Methode muss etwas erweitert werden, da beim Schließen eines Fensters nur dann eine Sicherheitsabfrage zum Speichern des Dokuments durchgeführt werden muss, wenn kein weiteres Fenster für dasselbe Dokument offen ist. Eine Ausnahme stellt ein *OnCloseQuery* beim Schließen der gesamten Anwendung dar. Hier erhalten *alle* Views eine *OnCloseQuery*-Abfrage, bevor auch nur das erste Fenster geschlossen wird. In diesem Fall muss sichergestellt werden, dass für jedes ungespeicherte Dokument genau ein Fenster eine Sicherheitsabfrage durchführt.

Die im TreeDesigner gewählte Ausarbeitung des Dokument-View-Konzepts ist sicher nicht die einzig mögliche. So können Sie in kommerziellen Anwendungen wie Microsoft Word beispielsweise ausprobieren, wie man in verschiedenen Ansichten desselben Dokuments verschiedene Bereiche des Dokuments markieren kann. Die Markierung ist dort also eine Eigenschaft des Views und nicht des Dokuments. Im TreeDesigner wurde die Markierung der Einfachheit halber als Eigenschaft der einzelnen Grafikobjekte und damit als Teil des Dokuments realisiert, weshalb in allen Ansichten desselben Dokuments immer dieselben Objekte markiert sind.

Eine andere Erweiterungsmöglichkeit bestünde darin, aus einem in mehreren Views angezeigten Dokument mehrere verschiedene Dokumente zu machen, wenn das Dokument in einem der Views unter einem anderen Namen gespeichert wird. Die derzeitige Verhaltensweise des TreeDesigners führt nämlich noch dazu, dass bei einer solchen Speicherung einfach der Dokumentname des bestehenden Dokuments geändert wird, und zwar in allen angeschlossenen Views aktualisiert wird (deren Titel schließlich durch die oben abgedruckte *UpdateCaption*-Methode aktualisiert werden).

5.3.2 Eine Klasse für das Grafikdokument

Dieses Kapitel beschreibt das Dokument des TreeDesigners (definiert in der Unit *GrDoc*), zur Vereinfachung allerdings ohne Berücksichtigung der Verbindungslinien zwischen den Objekten. In den folgenden Quelltext-Auszügen wurden also die Zeilen weggelassen, die die Verbindungslinien betreffen. Einführende Informationen zu den Verbindungen finden Sie in Kapitel 5.3.4.

Ganz allgemein besteht das Grafikdokument aus einer Datenstruktur, die eine Ansammlung von Grafikobjekten speichert. Ohne selbst eine neue Datenstruktur schreiben zu müssen, haben wir hier die Auswahl zwischen den Arrays der Sprache Pascal und den Listenobjekten der VCL.

Im TreeDesigner fiel die Wahl auf die Listen, da diese im Programm erheblich einfacher zu handhaben sind: Wenn ein Objekt beispielsweise gelöscht wird, verwenden Sie einfach die Methode *TList.Delete* und müssen nicht die Lücken füllen oder verwalten, die bei einer Löschoperation in einem Array entstehen würden.

Ableiten einer spezialisierten Listenklasse

Der TreeDesigner verwendet hierzu also die Klasse *TList*. Das Grafikdokument könnte nun in einer neuen Klasse realisiert werden, die ein *TList*-Objekt enthält:

```
type
  TGraphicDoc = class
    Elements: TList;
    ...
```

Einfacher ist jedoch die im TreeDesigner gewählte Möglichkeit, die Klasse *TGraphicDoc* direkt von der VCL-Klasse *TList* (Kapitel 4.1.3) abzuleiten. *TGraphicDoc* deklariert zwar auch ein paar neue Variablen, diese werden jedoch nur als Hilfsvariablen zur Laufzeit verwendet. Der gesamte Inhalt des Dokuments befindet sich in der von *TList* verwalteten Liste, deren Elemente über das geerbte Property *Items* angesprochen werden können. Dies ändert sich übrigens auch dann nicht, wenn die Verbindungen zwischen den Objekten hinzukommen, denn diese werden als Bestandteil der einzelnen Objekte gespeichert, ohne dass sich die Liste darum kümmern muss. Fast alle von *TGraphicDoc* neu deklarierten Elemente sind daher Properties und Methoden:

```
type
  TGraphicElement = class;
  TShapeType = (stNone, stRect, stEllipse, stLine,
                stHexagon, stRhomb);
  TGraphicDoc = class(TList)
    private
    (* interne Property-Daten *)
      FPaintBox: TPaintBox;
      FWidth, FHeight: integer;
      FFileFormatVersion: char;
      FViews: TList;  { darstellende Views }
      FMarkedElementCount: integer;
      FFileName: string;
      FChangeCount: integer;
    (* Property-Zugriffsmethoden *)
      procedure SelectedSetText(s: string);
      procedure SelectedSetFontName(s: string);
      procedure SelectedSetFontSize(s: Integer);
```

```
    procedure SelectedSetFont(F: TFont);
    procedure SelectedSetPenWidth(w: Integer);
    procedure SelectedSetFontColor(c: TColor);
    procedure SelectedSetBrushColor(c: TColor);
    procedure SelectedSetPenColor(c: TColor);
    procedure SelectedSetShape(s: TShapeType);
    function GetGraphicElement(index: Integer): TGraphicElement;
    function GetViews(Index: Integer): TGraphicView;
    function GetViewCount: Integer;
    procedure SetFileName(const Value: string);
  public
  (* Konstruktor und Destruktor *)
    constructor Create;
    destructor Destroy; override;
  (* View-Verwaltung *)
    procedure AddView(AView: TGraphicView);
    procedure DeleteView(AView: TGraphicView);
    property Views[Index: Integer]: TGraphicView read GetViews;
    property ViewCount: Integer read GetViewCount;
  (* Die Markierung betreffende Properties und Methoden *)
    property MarkedElementCount: integer read FMarkedElementCount;
    procedure SelectRect(R: TRect);
    procedure DeleteSelected;
    function AnySelected: Boolean;
    procedure DeselectAll; { alle deselektieren }
  (* Elemente freigeben *)
    procedure FreeElement(Item: TGraphicElement);
    procedure FreeAll;
  (* Input/Output *)
    procedure SaveToStream(Stream: TStream; OnlyMarked: boolean;
                    SaveFormatVersion: char = CurrentFileVersion);
    procedure SaveToFile(Name: string);
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(Name: string);
    // weitere Ein- und Ausgabemethoden für XML siehe Kapitel 5.9.3
  (* Properties *)
    property Shape: TShapeType write SelectedSetShape;
    property Text: string write SelectedSetText;
    property FontColor: TColor write SelectedSetFontColor;
    property FontSize: Integer write SelectedSetFontSize;
    property FontName: string write SelectedSetFontName;
    property PenWidth: Integer write SelectedSetPenWidth;
    property PenColor: TColor write SelectedSetPenColor;
    property BrushColor: TColor write SelectedSetBrushColor;
    property Items[index: Integer]: TGraphicElement
                                          read GetGraphicElement;
    property Width: integer read FWidth write FWidth;
    property Height: integer read FHeight write FHeight;
    property FileFormatVersion: char read FFileFormatVersion
                                    write FFileFormatVersion;
    property ChangeCount: integer read FChangeCount;
```

```
      property FileName: string read FFileName write SetFileName;
  (* weitere Methoden *)
      function GetObjectByPoint(P: TPoint): TGraphicElement; overload;
      function GetObjectByPoint(P: TPoint; GripSize: Integer;
        var rm: TResizeMode) : TGraphicElement; overload;
      procedure PaintAll(Dest: TCanvas);
      procedure GetPaintRect(var R: TRect);
      procedure Scale(B, H: Extended);
    (* Einige in den jüngsten TreeDesigner-Versionen neu
       hinzugekommene Methoden sind hier der Übersicht
       halber nicht aufgeführt. *)
  end;
```

Wie die folgenden Abschnitte zeigen werden, erleichtern die zahlreichen Properties der Klasse *TGraphicDoc* den Umgang mit einem Objekt dieser Klasse (also einem Dokument) erheblich.

Nicht lesbare Properties *R121*

Die öffentlichen Properties von *TGraphicDoc* sind zum großen Teil relativ ungewöhnlich, da sie zwar beschrieben, aber nicht gelesen werden können. Mit dem Schreiben in diese Properties erreichen Sie, dass alle Grafikobjekte des Dokuments, die gerade markiert sind, mit einem neuen Attribut versehen werden. Schreiben Sie z.B. in das Property *BrushColor* des *TGraphicDoc*-Objekts einen Wert, so ändern Sie damit das Property *Brush.Color* aller ausgewählten Grafikelemente.

Im TreeDesigner findet diese Zuweisung beim *OnBrushColorChange*-Event der Farbpalette statt. Das Dokumentformular hat hier nur eine Zeile Code auszuführen:

```
procedure TDocumentForm.BrushColorChange(Sender : TObject; Color: TColor);
begin
  Document.BrushColor := Color;
end;
```

Sehen wir uns als Beispiel die Methode an, die den Schreibzugriff auf dieses Property ausführt:

```
procedure TGraphicDoc.SelectedSetBrushColor(c: TColor);
var
  i: Integer;
begin
  if MarkedElementCount > 0 then begin
    inc(FChangeCount);
    for i := 0 to Count-1 do
      with TGraphicElement(Items[i]) do
        if Marked then
          Brush.Color := c;
  end;
end;
```

Die Methode verwendet zwei von *TList* geerbte Properties: *Count*, das die Anzahl der Listenelemente enthält, und *Items*, das Array, in dem diese gespeichert sind. (Um diese Liste zu erweitern, verwenden andere Teile des Programms entsprechend die ebenfalls von *TList* geerbte Methode *Add*.)

Die anderen in der obigen Methode verwendeten Bezeichner gehören zur Klasse *TGraphicElement*, die in Kürze vorgestellt wird: *Marked* zeigt an, ob ein Element selektiert ist, *Brush* enthält den Pinsel, mit dem sich das Objekt zeichnet, und *Invalidate* bewirkt das Neuzeichnen des Elements.

MarkedElementCount ist ein Property des Dokumentobjekts; es enthält die Zahl der insgesamt markierten Elemente. Auch *FChangeCount* gehört zum Dokument, das darin die Zahl der seit dem letzten Speichern vorgenommenen Änderungen hochzählt.

Ein Typenumwandlungs-Property *R120*

Wie Sie schon am letzten Beispiel sehen konnten, können Sie mit einem *TGraphicDoc*-Objekt so arbeiten wie mit einem *TList*-Objekt, da die Verwaltung der Listenelemente vollständig von *TList* geerbt ist. Die oben gezeigte Methode *TGraphiDoc.SelectedSetBrushColor* zeigt jedoch eine kleine Unbequemlichkeit: Da *TList* nichts über den Typ der enthaltenen Elemente weiß, gibt sie die Elemente als *Pointer* zurück, so dass der Aufruf *Items[i]* zuerst in ein *TGraphicElement* zurückverwandelt werden muss. Aus diesem Grund definiert *TGraphicDoc* das indizierte Property *Items* neu (überschreibt es) und gibt seinen Elementen den Typ *TGraphicElement*:

```
{ Die Deklaration des Properties:
  property Items[index: Integer]: TGraphicElement
                                      read GetGraphicElement; }

function TGraphicDoc.GetGraphicElement(index: Integer)
                            : TGraphicElement;
begin
   { Endlos-Rekursion rekursion vermeiden -> Umwandeln in TList }
  Result := TGraphicElement(TList(self).Items[index]);
end;
```

Die Property-Lesemethode tut also nichts anderes, als den Typ des von *TList* zurückgegebenen Objekts umzuwandeln, so wie es oben die Methode *SelectedSetBrushColor* noch selbst gemacht hat. Wichtig ist dabei jedoch, dass sich das Objekt in dieser Methode erst in den Typ *TList* umwandelt (man könnte auch sagen »zurückverwandelt«, denn eigentlich handelt es sich ja um ein spezialisiertes *TGraphicDoc*-Objekt). Wenn es das nicht täte, würde der Lesezugriff auf das Property *Items* dazu führen, dass sich die Methode selbst aufruft, und zwar in einer endlosen Schleife. Die Umwandlung von sich selbst (*self*) in ein *TList*-Objekt führt dazu, dass der Compiler die *Items*-Lesemethode von *TList* aufruft, die ansonsten von der Neudeklaration des Properties *Items* und dessen Lesemethode *GetGraphicElement* verdeckt wäre.

Die obige Typenumwandlungsmethode wird also bei jedem normalen Lesezugriff auf *TGraphicDoc.Items* angesprochen. Daher war die Typenumwandlung von *Items[i]* in der Methode *SelectedSetBrushColor* unnötig. Die Methode *SelectedSetFontName* verzichtet auf diese Typenumwandlung und nutzt damit die Neudefinition des Properties *Items*:

```
procedure TGraphicDoc.SelectedSetFontName (s: string);
var
  i: Integer;
begin
  if MarkedElementCount > 0 then begin
    inc(FChangeCount);
    for i := 0 to Count-1 do
      with Items[i] do
        if Marked then
          Font.Name := s;
  end;
end;
```

Bemerkenswert ist noch, dass die Methode *GetGraphicElement* keinerlei Überprüfung dessen durchführt, ob das Listenelement wirklich den Typ *TGraphicElement* hat, in den es umgewandelt wird. Hätte es einmal einen anderen Typ, so käme es möglicherweise zu einem nicht leicht lokalisierbaren Programmfehler. Wir können uns aber an dieser Stelle darauf verlassen, dass das Programm nur Objekte des Typs *TGraphicElement* über die geerbte Methode *Add* zur Liste hinzufügt und dass daher die Abfrage des *Items*-Properties immer ein solches Objekt zu Tage fördert (Zugriffe außerhalb der Listengrenzen führen nicht zu einem *nil*-Wert, sondern zu einer Exception).

Weitere Klassenelemente

Die Schreibmethoden für die Properties *Shape*, *Text*, *FontColor*, *FontSize*, *FontName*, *PenColor*, *PenWidth* und *BrushColor* funktionieren ähnlich wie die Methode *SelectedSetFontName*, weshalb sie nicht abgedruckt zu werden brauchen. Ihre Bedeutung stimmt mit den entsprechenden Eigenschaften der Klasse *TGraphicElement* überein, die im nächsten Kapitel erläutert wird.

Die Properties *Height* und *Width* geben an, wie viele Einheiten die Zeichenfläche beinhalten sollte, auf der das Dokument dargestellt wird. Diese Größenangabe wird in keiner der Methoden von *TGraphicDoc*, sondern nur in den Formularmethoden berücksichtigt. Das Property *FileFormatVersion* wird während des Ladens einer Datei verwendet, um die Version des Dateiformats zu speichern (zur *Versionskennung* siehe Kapitel 5.3.3). Die Properties zur View-Verwaltung und *FileName* wurden bereits in Kapitel 5.3.1 vorgestellt.

Was bleibt, sind die folgenden Methoden von *TGraphicDoc*:

Element	**Bedeutung**
constructor Create	initialisiert *FWidth*, *FHeight* und *FViews* (ruft zuerst natürlich den geerbten Konstruktor auf).
destructor Destroy	gibt alle Grafikelemente und die Liste *FViews* frei (die Freigabe der Grafikelemente wird nicht automatisch durch den geerbten Listen-Destruktor erledigt, da *TList* die Listenelemente nicht als Objekte mit virtuellem *Destroy*-Destruktor, sondern nur als Zeiger speichert). Wird beim *OnDestroy*-Ereignis des Dokumentformulars aufgerufen.
procedure SelectRect	wählt alle Elemente, die sich innerhalb des als Parameter übergebenen Rechtecks befinden, aus (setzt deren *Marked*-Property auf *True*).
procedure DeleteSelected	löscht alle ausgewählten Elemente.
function AnySelected	überprüft, ob irgendwelche Elemente selektiert sind.
procedure DeselectAll	hebt die Auswahl aller Elemente auf.
procedure FreeElement	entfernt ein Element mit *TList.Remove* aus der Liste und gibt es mit *Free* frei.
procedure FreeAll	leert die Liste und gibt alle Grafikelemente mit *Free* frei.
procedure SaveToFile	erzeugt eine neue Datei oder überschreibt eine bestehende Datei und speichert darin den gesamten Dokumentinhalt, benutzt dafür die Methode *SaveToStream*.
procedure SaveToStream	speichert den gesamten Dokumentinhalt oder nur die selektierten Grafikelemente in einem Stream, der eine Datei sein kann oder ein MemoryStream (für das Kopieren der Daten in die Zwischenablage)
procedure LoadFromFile	liest das Dokument aus einer bestehenden Datei ein. Zuvor werden alle bestehenden Elemente mit *FreeAll* gelöscht.
procedure LoadFromStream	lädt Grafikelemente aus einem Stream.
function GetObjectByPoint	findet das angeklickte Objekt und den angeklickten Markierungspunkt heraus (siehe Kapitel 5.4.2).
procedure PaintAll	ruft die *Paint*-Methode aller Grafikelemente auf.
procedure Scale	skaliert die gesamte Grafik, wobei die Faktoren für horizontale und vertikale Skalierung in den beiden Parametern angegeben werden.
procedure GetPaintRect	liefert die Koordinaten des kleinsten Rechtecks, mit dem alle Grafikelemente des Dokuments überdeckt werden könnten. Diese Methode wird sowohl beim Exportieren des Dokuments als Metadatei als auch beim Drucken dazu benötigt, einen optimalen Ausschnitt aus der möglicherweise viel zu großen Zeichenfläche zu wählen.

Bevor wir nun die neu gezeichneten Objekte mit *TList.Add* hinzufügen können, benötigen wir eine zweite Klasse, in der diese Objekte definiert werden.

5.3.3 Eine Klasse für die Grafikobjekte

In einer ganz einfachen Anwendung, in der Sie beispielsweise nur Ellipsen und Kreise zeichnen können, deren Farbe und Umrandung vorgegeben ist, würde es genügen, für jedes Grafikobjekt einfach nur eine *TRect*-Struktur zu verwenden, in der das Rechteck gespeichert ist, das die Ellipse bzw. den Kreis umgibt.

Die Objekte im TreeDesigner sollen jedoch auch farbig gestaltbar sein und verschiedene Formen sowie eine Beschriftung haben können, daher benötigen wir hier mindestens eine eigene Datenstruktur, besser natürlich eine eigene Klasse. Diese Klasse heißt *TGraphicElement*, sie definiert alle zu einem Grafikelement gehörenden Daten, darunter auch eine *TRect*-Struktur für die Koordinaten.

Überlegungen zur Klassenstruktur

Vor der Implementierung der verschiedenen Grafiktypen gilt es, eine wichtige Entscheidung zur Programmstruktur zu treffen. Es konkurrieren hauptsächlich zwei Methoden, die unterschiedlichen Grafiktypen (Rechteck, Kreis etc.) in *TGraphicElement* zu vereinbaren:

- Für jeden Grafiktyp könnte eine eigene Klasse angelegt werden. Dies ist auf den ersten Blick die natürlichere Vorgehensweise. Jede der von *TGraphicElement* abgeleiteten Klassen könnte dann durch Überschreiben der Methode *Paint* große Freiheiten erreichen.
- Die zweite Möglichkeit ist, alle Grafiktypen bereits in der Klasse *TGraphicElement* abzudecken. Um welchen Grafiktyp es sich handelt, müsste dann in einer Variablen gespeichert werden. Dies scheint auf den ersten Blick ein nicht ganz OOP-gemäßer Trick zu sein, da er die Klasse *TGraphicElement* mit der Arbeit vieler einzelner Klassen (Typen grafischer Elemente) belastet.

  Indem Sie eine solche *TGraphicElement*-Klasse jedoch nicht ansehen als eine Klasse, die die Arbeit verschiedener speziellen Klassen übernimmt, sondern als eine Klasse, die eine rechteckige Fläche mit verschiedenen Formen auszufüllen vermag, kommen Sie jedoch wieder mit der OOP-Philosophie ins Reine. (Tatsächlich können Sie die Form der Grafikobjekte ja auch zur Laufzeit ändern, indem Sie eine andere Form von den Schaltern der Werkzeugleiste auf das zu ändernde Element ziehen.)

Der größte Vorteil der zuletzt genannten Möglichkeit – die im TreeDesigner auch realisiert ist – kommt erst in abgeleiteten Klassen zum Tragen: Angenommen, Sie wollten aus irgendeinem Grund sowohl Rechtecke als auch Ellipsen und die anderen Grafiktypen um eine besondere Funktionalität erweitern, indem Sie eine neue Klasse ableiten. Hätte jeder Grafiktyp seine eigene Klasse, so müssten Sie von jeder dieser Klassen eine neue Klasse ableiten und in dieser jeweils die neue (identische) Funktionalität imple-

mentieren, was durch die enorme Redundanz eine völlig unpraktikable Lösung wäre. Sind aber alle Grafiktypen in einer Klasse vereint, würde es genügen, eine einzige neue Klasse davon abzuleiten.

Dies soll selbstverständlich nicht heißen, dass Klassen immer mit möglichst viel Funktionalität vollgestopft sein sollten. Da sich die Unterschiede zwischen den Grafikobjekten jedoch nur auf die Form und damit die Variable *ShapeType* beschränken, lohnt sich die Zusammenfassung zu einer Klasse in diesem Fall.

Hinweis: Der TreeDesigner definiert selbst bereits zwei von *TGraphicElement* abgeleitete Klassen, die jedoch hier schon aus Platzgründen nicht weiter beschrieben werden können: *TGradientObject* definiert die Farbverlauf-Grafikelemente, die Sie durch den untersten Schalter der Werkzeugleiste am linken Fensterrand erhalten; die Klasse *TMultiLineElement* ist für die mehrzeilige Textausgabe zuständig, die in der Darstellung einer XML-Datenstruktur verwendet wird.

Übersicht über TGraphicElement

Ohne die in Kapitel 5.3.4 hinzukommenden Verbindungen zu anderen Objekten und ohne einige Erweiterungen der TreeDesigner-Versionen 2.5 – 3.5 sieht die Deklaration der Klasse *TGraphicElement* folgendermaßen aus:

```
type
  TShapeType = (stNone, stRect, stEllipse, stLine,
                stHexagon, stRhomb, stRoundRect);
  { In der Markierung eines Rechtecks werden acht Größenänderungspunkte
    hervorgehoben, die den folgenden Größenänderungs-Modi entsprechen: }
  TResizeMode = (rmNone, rmRB, rmLB, rmRT, rmLT, rmR, rmL, rmT, rmB);
  TGrObjectText = string[50]; // "historische" Beschränkung auf 50 Zeichen
    // (könnte auch auf einen normalen String variabler Länge erweitert
    // werden, dieser würde jedoch ein neues Dateiformat erfordern.)
  TGraphicElement = class(TPersistent)
    { Die Klasse zieht zurzeit keinen Vorteil aus ihrer
      Vorgängerklasse TPersistent. }
  private
  (* persistente Daten (werden in Datei gespeichert) *)
    Points: TRect;
    FShapeType: TShapeType;
    FText: TGrObjectText;
    Content: TElementContent; { Die Klasse TElementContent kapselt einen
       eventuellen zusätzlichen Inhalt des Grafikelements, beispielsweise
       eine Bitmap oder eine Metadatei (zur Anwendung siehe Kap. 5.1.3).
       Wird in den folgenden Quelltextauszügen nicht weiter beachtet. }
    (* Dies sind noch nicht alle persistenten Daten: Auch einige Attribute
       der öffentlichen Objekte Pen, Brush und Font werden gespeichert! *)
  (* Laufzeitvariablen *)
    FMarked: Boolean;
```

```
    Document: TGraphicDoc;
  (* Property-Zugriffsmethoden *)
    procedure SetMarked(val: Boolean);
    function GetMarked: Boolean;
    procedure SetShape(s: TShapeType);
    function GetCoord(Index: Integer): Integer;
    procedure SetText(s: TGrObjectText);
    procedure GraphicsObjectChanged(Sender: TObject);
  public
    Pen: TPen;
    Brush: TBrush;
    Font: TFont;

    property ShapeType: TShapeType read FShapeType write SetShape;
    property Text: TGrObjectText read FText write SetText;
    property Marked: Boolean read GetMarked write SetMarked;

    property Left: Integer index 1 read GetCoord;
    property Right: Integer index 2 read GetCoord;
    property Top: Integer index 3 read GetCoord;
    property Bottom: Integer index 4 read GetCoord;

  (* Konstruktoren, Destruktoren, Speichern und Laden: *)
    constructor Create(Doc: TGraphicDoc; InitRect: TRect;
                       SType: TShapeType); virtual;
    destructor Destroy; override;
    procedure LoadFromStream(Stream: TStream);
    procedure StoreToStream(Stream: TStream; Index: Integer);
  (* Veränderung der Koordinaten und Verschieben des Objekts *)
    procedure SetNewRect(NewRect: TRect);
    procedure Move(XTranslate, YTranslate: Integer);
  (* Methoden im Zusammenhang mit dem Zeichnen *)
    procedure Paint(CanvasRef: TCanvas); virtual;
    procedure PaintMark(CanvasRef: TCanvas);
    procedure Invalidate(View: TGraphicView);
    procedure InvalidateRect(var R : TRect; View: TGraphicView);
    procedure InvalidateAllViews;
   (* Zeichnen während einer Drag-Operation *)
    procedure XorPaint(CanvasRef: TCanvas;
       XorRect: TRect); virtual;
   (* Informationsmethoden über die verbrauchte Fläche *)
    procedure GetOriginalRect(var R: TRect);
    procedure GetNormalizedRect(var R: TRect);
    procedure GetPaintRect(var R: TRect);
    procedure GetInnerRect(var R: TRect);
    function PointInShape(TestPoint: TPoint): Boolean; overload;
    function PointInShape(TestPoint: TPoint;
                          var rm: TResizeMode): Boolean; overload;
    function ContainedIn(R: TRect): Boolean;
  (* sonstige Methoden *)
```

```
    function GetHMiddle: Integer;
    function GetVMiddle: Integer;
  end;
```

Die Klassenelemente bestehen aus verschiedenen Gruppen:

- Private Daten: Zu diesen gehören die Property-Variablen *FMarked*, *FShapeType* und *FText*, auf die von außen nur über die Properties zugegriffen wird, die Variable *Document*, die auf das *TGraphicDoc*-Dokument weist, in dem das Objekt enthalten ist, und *Points*, eine *TRect*-Struktur, die die Koordinaten des Objekts enthält. Letztere beiden Variablen werden bereits im Konstruktor initialisiert. Während *Document* normalerweise nur einmal gesetzt wird, können Sie die *Points* durch *SetNewRect* (siehe Tabelle) jederzeit ändern.
- Private Methoden: Diese dienen nur dem Zugriff auf die Properties und brauchen nicht einzeln erläutert zu werden.
- Grafikattribute: Zwei der Grafikattribute werden über Properties angesprochen: die grafische Form des Elements (*ShapeType*) und der Beschriftungstext (*Text*). Die Farben, die Liniendicke und die Schriftart werden in eigenen Objekten gespeichert: *Pen*, *Brush* und *Font*. Diese sind nicht als Properties implementiert, da sie nicht als Ganzes neu gesetzt werden. Vielmehr werden immer nur einzelne Attribute dieser Objekte gesetzt, wie oben z.B. in der Methode *TGraphicDoc.SelectedSetFontName*.
- Weitere Properties: Das Property *Marked* gibt an, ob das Element markiert ist oder nicht. *Left*, *Right*, *Top* und *Bottom* erlauben den Lesezugriff auf die einzelnen Koordinatenbestandteile. Sie sind ein Beispiel für Properties mit gemeinsamer Zugriffsmethode (trotz des Wortes *index* in ihrer Deklaration nicht zu verwechseln mit den indizierten Properties, bei denen es sich um die Array-Properties handelt). Ändern lassen sich die vier Werte nur gemeinsam über die Methode *SetNewRect*.
- Die öffentlichen Methoden sind in der folgenden Tabelle zusammengefasst.

Methode	**Beschreibung**
Create	Konstruktor, überschreibt den geerbten Konstruktor, initialisiert die Koordinaten, die Form und den Zeiger auf das *TGraphicDoc*-Dokument.
Destroy	Destruktor, gibt den dynamisch belegten Speicher frei.
LoadFromStream	liest die persistenten Daten des Objekts aus einem Stream.
SaveToStream	speichert die persistenten Daten in einem Stream.
GetHMiddle	liefert die X-Koordinate des Objektmittelpunkts zurück (verwendet bei der zentrierten Ausgabe der Beschriftung).
GetVMiddle	liefert die zu *GetHMiddle* fehlende Y-Koordinate.
XorPaint	übernimmt die Ausgabe des Objekts bei Ziehoperationen mit der Maus.
Paint	zeichnet das komplette Grafikelement in das übergebene *TCanvas*-Objekt.

Methode	Beschreibung
PaintMark	zeigt die Markierung des Elements am Bildschirm an.
InvalidateRect	weist ein View an, einen bestimmten Bereich neu zu zeichnen (Hilfsmethode für *InvalidateAllViews*).
InvalidateAllViews	bewirkt den Aufruf von *InvalidateRect* für alle Views, die an das übergeordnete Dokument angeschlossen sind (ein Verweis auf dieses Dokument befindet sich ja in der privaten Variablen *Document*).
Move	verschiebt das Element um bestimmte Beträge in horizontaler und vertikaler Richtung.
SetNewRect	ändert die Koordinaten und sorgt für eine Neuausgabe der Grafik an den veränderten Stellen.
GetOriginalRect	liefert die unveränderten Koordinaten, die in *Points* angegeben sind.
GetNormalizedRect	liefert das Originalrechteck, vertauscht die Koordinaten aber so, dass *Bottom>Top* und *Right>Left*.
GetPaintRect	liefert das Rechteck, das durch das Objekt belegt wird; zum Original-Rechteck kommen hier evtl. überstehende Umrandungslinien hinzu (wenn der Stift breiter als 1 Pixel ist).
GetInnerRect	liefert das innere Rechteck ohne die Umrandungslinien; dieses Rechteck entspricht dem inneren Bereich des Objekts, falls *ShapeType=stRect*.
PointInShape	entscheidet, ob der übergebene Punkt im Bereich des Grafikobjekts ist, sieht das Grafikobjekt jedoch immer als Rechteck an. Zur richtigen Behandlung von Linien und anderen Formen wären einige Erweiterungen dieser Methode notwendig. Die zweite (überladene) Version dieser Methode mit zusätzlichem *TResizeMode*-Parameter prüft auch, ob der Punkt in einer Randmarkierung zur Änderung der Größe liegt, und, wenn ja, welche Art der Größenänderung dies bedeutet.
ContainedIn	entscheidet, ob das Grafikelement innerhalb eines Rechtecks liegt – diese Methode funktioniert für alle geometrischen Formen gleichermaßen gut.

Einige dieser Methoden könnten übrigens durchaus auch als private Methoden deklariert werden, denn sie werden von außen nicht aufgerufen, so z.B. *PaintHexagon* und *PaintRhomb*.

Änderung der Zeichenattribute

Dadurch, dass die meisten Attribute von *TGraphicElement* als Properties implementiert sind, können die Methoden von *TGraphicDoc* diese Attribute durch einfache Zuweisung eines neuen Wertes ändern (z.B. durch die *TGraphicDoc.SelectedSetFontName* in Kapitel 5.3.2). Die Schreibmethode für diese Properties führt das erforderliche Neuzeichnen automatisch durch (wie *Invalidate* funktioniert, lesen Sie in Kapitel 5.5.1):

```
procedure TGraphicElement.SetShape(s: TShapeType);
begin
  FShapeType := s;
```

```
    InvalidateAllViews;
    ElementChangeNotify; // Benachrichtigung eventueller COM-Clients,
                         // siehe Kapitel 8.7.6
  end;
```

Der TreeDesigner ermöglicht es dem Benutzer über die Werkzeugleiste lediglich, Einfluss auf die Properties *Pen.Width*, *Brush.Color*, *Pen.Color*, *Font.Color*, *Font.Size* und *Font.Name* zu nehmen, die anderen Properties von *Pen*, *Brush* und *Font* bleiben ungenutzt.

Allerdings sind die Objekte *Pen*, *Brush* und *Font* keine Properties. Dies wäre hier auch nicht sinnvoll, da nicht Stift, Pinsel und Schrift als Ganzes, sondern nur einzelne ihrer Attribute geändert werden. Die Frage ist nun, wie eine Änderung dieser Attribute zu einem automatischen Neuzeichnen führen kann wie in der obigen Methode *SetShape*. Offenbar sind wir hier an den Grenzen der Properties angelangt, da durch die Zeile

```
Objekt.Brush.Color := clLightRed;
```

zwar das *Brush*-Objekt selbst über die Änderung informiert wird, das *TGraphicElement* jedoch keine Möglichkeit hat, das Setzen des Properties mit einer eigenen Methode zu verknüpfen.

Die OnChange-Ereignisse nicht-visueller Objekte

Borland hat dies bei der Entwicklung der VCL berücksichtigt und vielen nicht-visuellen Objekten ein *OnChange*-Ereignis mit auf den Weg gegeben. So lösen Stifte, Pinsel und Schriftobjekte immer dann ein *OnChange*-Ereignis aus, wenn eines ihrer Properties mit einem neuen Wert beschrieben wird. Die Klasse *TCanvas* verwendet diese Ereignisse beispielsweise, um ihr eigenes *OnChange*-Ereignis auszulösen.

Ähnlich wie *TCanvas* enthält auch *TGraphicElement* drei Objekte *Pen*, *Brush* und *Font*. Damit das Grafikobjekt neu ausgegeben wird, wenn sich eines der Attribute ändert, verknüpft es die folgende einfache Methode mit den *OnChange*-Ereignissen aller drei Objekte:

```
procedure TGraphicElement.GraphicsObjectChanged(Sender: TObject);
begin
  Document.IncChangeCount; // erhöht privaten Zähler Document.FChangeCount
  InvalidateAllViews;
  ElementChangeNotify;
end;
```

Die Verknüpfung findet im Konstruktor statt, dem wir uns als Nächstes zuwenden werden.

Konstruktor, Destruktor und dynamische Objekte *R130*

Die Vorgänge im Konstruktor der Klasse *TGraphicElement* sind ein gutes Beispiel für den dynamischen Aufbau von Objekten:

```
constructor TGraphicElement.Create(AList: TGraphicDoc;
                        InitRect: TRect; SType: TShapeType);
begin
  inherited Create;
  Document := Doc;
  FShapeType := SType;
  Points := InitRect;
(* Konstruktion der dynamischen Objekte: *)
  Pen := TPen.Create;
  Brush := TBrush.Create;
  Font := TFont.Create;
  Pen.OnChange := GraphicsObjectChanged;
  Brush.OnChange := GraphicsObjectChanged;
  Font.OnChange := GraphicsObjectChanged;
{ für die von diesem Objekt ausgehenden Verbindungen kommt
  später noch hinzu: }
{  Edges := TList.Create;   }
  { die anderen Daten werden automatisch auf 0 gesetzt. }
end;
```

Der Code des Konstruktors *Create* erfüllt drei verschiedene Aufgaben:

- Er ruft den geerbten Konstruktor auf, damit sich die Basisklasse initialisieren kann.
- Da alle Variablen bereits automatisch mit Nullen initialisiert werden, braucht er nur einige Werte zu setzen, deren Voreinstellung von 0 verschieden ist, wie *FShapeType*, *Document* und *Points*, also genau die Werte, die als Parameter im Aufruf des Konstruktors enthalten sind.
- Er erzeugt durch weitere Konstruktoraufrufe die dynamischen Objekte, die im Grafikelement enthalten sind, und initialisiert diese (explizit initialisiert werden müssen lediglich deren Verknüpfungen zu den *OnChange*-Ereignissen – wie im letzten Abschnitt besprochen).

Die wichtigste Aufgabe des Konstruktors, das Reservieren des Speichers für das *TGraphicElement*-Objekt selbst, wird automatisch ausgeführt, und zwar schon bevor die erste Zeile des Konstruktors erreicht wird (zu weiteren Details von Konstruktoren siehe Kapitel 2.3.1).

Der dynamische Aufbau eines Grafikobjekts beginnt damit, dass die in Kapitel 5.4.1 noch folgende Methode zum Event *OnMouseDown* ein neues *TGraphicElement* erstellt:

```
MouseObject := TGraphicElement.Create(
         Document, { so heißt das TGraphicDoc-Objekt des Formulars }
         Rect(x, y, x, y),  { das Anfangsrechteck ist ein Punkt   }
```

```
            CurShapeType);  { CurShapeType speichert die Auswahl des
                              Benutzers in den Schaltern der Mauspalette }
```

Was jetzt noch fehlt, ist der Destruktor. Er muss lediglich die dynamischen Objekte freigeben und den geerbten Destruktor aufrufen:

```
destructor TGraphicElement.Destroy;
begin
  Pen.Free;
  Brush.Free;
  Font.Free;
{ später kommt hinzu: }
{  Edges.Free; }
  inherited Destroy;
end;
```

Speicherverbrauch

Der Aufruf von *TGraphicElement.Create* belegt in der fertigen Version (mit den Verbindungen zwischen den Objekten) bereits 132 Bytes für die Variablen, die in *TGraphicElement* deklariert sind (dieser Wert wurde über die Methode *InstanceSize* ermittelt). Der Konstruktor erzeugt dann die Stift-, Pinsel- und Schriftobjekte, die weitere 28, 24 und 36 Bytes benötigen, sowie die Liste der Verbindungen, die ähnlich viel Platz verbraucht. Nicht mitgerechnet sind allerdings Speicherbereiche, die in den Klassen *TList*, *TBrush*, *TFont* und *TPen* dynamisch reserviert werden.

Eine effektivere Möglichkeit, den Speicherbedarf zu reduzieren, bestünde darin, die Beschriftung der Objekte dynamisch zu verwalten (z.B. durch Verwendung der normalen langen Object-Pascal-Strings) und die Farben, die Liniendicke und den Namen der Schrift in einzelnen Variablen zu speichern statt innerhalb der Objekte *Pen*, *Brush* und *Font*:

```
BColor, PColor, FColor: TColor;
PWidth: Integer;
FontName: string;
```

Solange der TreeDesigner weniger als 1000 oder 10000 Objekte verwalten soll, ist der dadurch entstehende Zusatzaufwand jedoch kaum notwendig.

Dateioperationen *R107*

Die nächste wichtige Aufgabe, die *TGraphicElement* zu erfüllen hat, ist, seine Daten in einen Stream zu schreiben, und zwar so, dass sie später genauso wieder zurückgelesen werden können.

Hinweis: Die tatsächlichen Methoden des TreeDesigners weichen in Teilen von den hier gezeigten Methoden ab, um das Dateiformat kompatibel zur 16-Bit-Version zu halten. Auf diese Kompatibilitäts- und Portierungsfragen geht die Datei *D1und2.rtf* auf der CD näher ein.

Zuständig für die einzelnen Objekte sind die *TGraphicElement*-Methoden *SaveToStream* und *LoadFromStream* (zu den einzelnen Methoden von *TStream* siehe Kapitel 4.3):

```
procedure TGraphicElement.SaveToStream(Stream: TStream;
     Index: Integer); (* Index wird für das Speichern der Verbindungen
           benötigt, siehe "Speichern von Zeigern" in Kapitel 5.3.4. *)
  procedure WriteColorToStream(s: TStream; c: TColor);
  begin
    S.Write(c, sizeof(c));
  end;

var
  i: Integer;
  s: string[LF_FACESIZE - 1];
begin
  { einfach zu schreibende Daten }
  Stream.Write(Points, sizeof(Points));
  Stream.Write(FShapeType, sizeof(FShapeType));
  Stream.Write(FText, length(FText)+1);
  { Farben }
  WriteColorToStream(stream, Pen.Color);
  WriteColorToStream(stream, Brush.Color);
  WriteColorToStream(stream, Font.Color);
  { umständlich zu schreibende Daten, weil es sich nicht
    um eigene Properties handelt: }
  i := Pen.Width;
  Stream.Write(i, sizeof(i));
  i := Font.Size;
  Stream.Write(i, sizeof(i));
  s := Font.Name;
  Stream.Write(s, length(s)+1);

  FileIndex := Index; { Für Speicherung der Verbindungen merken }
end;

procedure TGraphicElement.LoadFromStream(Stream: TStream);
var
  C: TColor;
  i: Integer;
  s: string[LF_FACESIZE - 1];;
begin
  FMarked := False;
  Visited := False;
  { einfach zu lesende Daten }
  Stream.Read(Points, sizeof(Points));
```

```
    Stream.Read(FShapeType, sizeof(FShapeType));
    Stream.Read(FText[0], 1);
    Stream.Read(FText[1], Byte(FText[0]));
    { Farben }
    Stream.Read(C, sizeof(TColor));
    Pen.Color := C;
    Stream.Read(C, sizeof(TColor));
    Brush.Color := C;
    Stream.Read(C, sizeof(TColor));
    Font.Color := C;
    { umständlich zu lesende Daten: }
    Stream.Read(i, sizeof(i));
    Pen.Width := i;
    Stream.Read(i, sizeof(i));
    Font.Size := i;
    Stream.Read(s[0], 1);
    Stream.Read(s[1], Byte(s[0]));
    Font.Name := s;
  end;
```

SaveToStream und *LoadFromStream* schreiben und lesen nur die wichtigen Daten, speichern also nicht die kompletten Objekte *Pen*, *Brush* und *Font*, sondern wählen aus diesen die verwendeten Daten aus.

Die Properties zeigen sich in diesen Methoden von ihrer problematischen Seite, denn sie sind nicht als Variablenparameter verwendbar – und die Stream-Methoden *Read* und *Write* benötigen einen solchen Variablenparameter. Die beiden obigen Methoden müssen daher Hilfsvariablen bereitstellen, in denen sie den Wert des Properties zwischenspeichern und die sie an die Stream-Methoden übergeben können.

Die Methode *SaveToStream* geht bei den Farben einen etwas anderen Weg. Sie verwendet die lokale Prozedur *WriteColorToStream*, die keine Variablenparameter erwartet. Wenn *SaveToStream* ihr die Properties direkt übergibt, wird durch den Prozeduraufruf eine Kopie der Property-Werte auf dem Stack angefertigt, so dass *WriteColorToStream* diese Kopie als Variablenparameter an den Stream weitergeben kann.

Strings

Weitere Umstände bereitet der String zur Beschriftung. Da die obigen Methoden aus Gründen der Effektivität keine *TFiler*-Objekte verwenden, müssen sie den String per Hand speichern. Da es außerdem sehr platzverschwenderisch wäre, für jeden String eine feste Zahl von Zeichen zu speichern, schreibt *SaveToStream* nur die tatsächlich existierenden Zeichen des Strings in die Datei. Als Erstes ist daher die Speicherung der Zeichenzahl erforderlich, die sich immer im nullten Zeichen eines kurzen Pascal-Strings befindet. Dementsprechend liest die Lesemethode zuerst die Länge des Strings, um dann im zweiten Schritt die richtige Bytemenge für alle Zeichen des Strings lesen zu können.

Hinweis: Die gezeigten Anweisungen zur String-Speicherung funktionieren nur mit kurzen Pascal-Strings. Die Länge von langen Strings wird über die *length*-Funktion ermittelt; und da dann auch Längen über 255 Zeichen in Frage kommen, reicht ein Byte zur Speicherung der exakten Länge nicht mehr aus. Weitere Informationen zum Thema kurze und lange Pascal-Strings finden Sie in Kapitel 2.4.4.

Dateioperationen für das Dokument

In der Klasse des Dokuments *TGraphicDoc* sorgen die Methoden *SaveToFile* und *LoadFromFile* für das Speichern der gesamten Elementliste. Um die Lese- und Schreibmethoden nicht an die Verwendung von Dateien zu binden, sondern sie auf jeden abstrakten *TStream* anzuwenden (wichtig z.B. für das Kopieren von Grafikobjekten in die Zwischenablage), werden die eigentlichen Lese- und Schreiboperationen in die separaten Methoden *ReadFromStream* und *WriteToStream* ausgelagert, die dann von *SaveToFile* und *LoadFromFile* aufgerufen werden:

```
const
  NullRect: TRect = (Left:0; Top:0; Right:0; Bottom:0);
  CurrentFileFormatVersion = '2'; { Dateiformats-Version }
  TVFSignature: string = 'EWTD-V1'+CurrentFileFormatVersion;

procedure TGraphicDoc.SaveToFile(Stream: TStream);
var
  ObjCount: Integer;
  i: Integer;
begin
  Stream.Write(TVFSignature[1], 8); { Dateikennung schreiben }
  { Anzahl der Objekte vermerken }
  ObjCount := Count;
  Stream.Write(ObjCount, sizeof(ObjCount));
  { Schreiben der einzelnen Objekte }
  for i := 0 to Count-1 do
    TGraphicElement(Items[i]).SaveToStream(Stream, i);
end;

procedure TGraphicDoc.SaveToFile(Name : string);
var
  Stream : TFileStream;
begin
  Stream := TFileStream.Create(Name, fmCreate);
  WriteToFile(Stream, false);
  Stream.Free;
  FileName := Name;
  FChangeCount := 0;
end;
```

```
procedure TGraphicDoc.LoadFromStream(Stream: TStream);
var
  ObjCount: Integer;
  i: Integer;
  NextObject: TGraphicElement;
  TestSig: string[8];
begin
  { Dateikennung und -Version überprüfen }
  Stream.Read(TestSig[1],8);
  TestSig[0] := #7;
  TVFSignature[0] := #7;
  if not ( (TestSig = TVFSignature) and  { Kennung prüfen }
           (TestSig[8] = '0'))           { Version prüfen }
    then raise EFileFormatError.Create('Fehler im TVF-Dateiformat');
  FFileFormatVersion := TestSig[8]; { Formatversion kann von den
             Lademethoden der Grafikelemente abgefragt werden }
  { Zahl der Objekte lesen }
  Stream.Read(ObjCount, sizeof(ObjCount));
  { Objekte lesen }
  for i := 1 to ObjCount do begin
    NextObject := TGraphicElement.Create(self, NullRect, stRect);
    NextObject.LoadFromStream(Stream);
    Add(NextObject);
  end;
end;

procedure TGraphicDoc.LoadFromFile(Name : string);
var
  Stream : TFileStream;
begin
  FreeAll;
  Stream := TFileStream.Create(name, fmOpenRead);
  LoadFromStream(Stream);
  DeselectAll;
  Stream.Free;
  FileName := Name;
  FChangeCount := 0;
end;
```

Versionskennung

Um ein Dokument als das eigene zu erkennen, versieht eine Anwendung eine Datei üblicherweise mit einer Kennung, die meistens noch Angaben zur verwendeten Programmversion enthält. Wenn sich die Dateiformate so von Version zu Version ändern, ist es einer neueren Version über diese Kennung möglich, das alte Format automatisch in das neue umzuwandeln. Die Kennung für die TreeDesigner-Dateien ist im String *TVFSignature* angegeben, wobei die beiden letzten Zeichen desselben die Versionsnummer angeben.

Hinweis: In den TreeDesigner-Versionen 2.5 und 3.5 wurden neue Dateiformat-Versionen eingeführt – zuletzt, um die Grafikelemente verschiedener Klassen unter Nutzung der Polymporphie zu speichern. Dateien aus TreeDesigner-Versionen vor 2.5 (bei denen die Versionskennung mit »V10« endet), können weiterhin eingelesen werden, denn die vollständige *LoadFromStream*-Methode auf der CD berücksichtigt auch das alte Format. Beim Speichern werden die Dateien quasi automatisch in das Format »V11« konvertiert und können nicht mehr mit alten TreeDesigner-Versionen gelesen werden! Das neue Format »V12« wird auch von der aktuellen TreeDesigner-Version nur erzeugt, wenn das Grafikdokument Elemente enthält, die dieses Format erfordern (Farbverlaufs- oder Mehrzeilentext-Elemente). Für eine Behandlung in diesem Buch wären diese Erweiterungen zu umfangreich, daher beschränkt sich der im Buch abgedruckte Code auf die Daten, die schon seit der Dateiformats-Version des TreeDesigner 1.0 in die Datei geschrieben werden. Wenn Sie sich für die Nutzung der Polymorphie bei der Speicherung interessieren, sei auf *R108* (Seite 480) verwiesen, der TreeDesigner geht nach dem dort beschriebenen Prinzip vor.

Schreiben der Objektliste

Das Schreiben der Objektliste ist aus der Sicht des Dokuments kein Problem. Damit die Lesemethode weiß, wie viele Objekte zu lesen sind, schreibt *SaveToFile* zuerst die Zahl der Objekte in die Datei (wobei das Property *Count* wieder nicht direkt als Variablenparameter an den Stream übergeben werden kann). Für jedes Objekt muss *SaveToFile* dann lediglich die Schreibmethode des Objekts aufrufen. *ReadToFile* verwendet entsprechend die Lesemethode, muss aber zuvor per Konstruktoraufruf ein neues Objekt erzeugen.

5.3.4 Baumstrukturen

Dieses Kapitel beschreibt die Erweiterungen, die die oben beschriebenen Klassen erfahren, um die Baumstruktur speichern zu können. Aus Platzgründen kann neben einer Übersicht über die neue Klasse *TEdge* und über die Erweiterungen der Klasse *TGraphicElement* nur eine vollständige Methode zum Anordnen des Baums besprochen werden.

Bedienung

Wie schon in der Einführung erwähnt, können Sie im TreeDesigner zwei Objekte verbinden, indem Sie ein Objekt markieren und das andere Objekt in Verbindung mit den Tasten Shift und Strg anklicken. Das zuerst markierte Objekt wird dadurch zum *Nachfolger* des als Zweites angeklickten Objekts. Diese Reihenfolge widerspricht zwar der Intuition, derzufolge der Vorgänger *vor* dem Nachfolger angeklickt werden müsste, hat aber zum Vorteil, dass Sie *mehrere* Nachfolger markieren und sie mit einem

Vorgänger verbinden können. (Jedes Objekt darf beliebig viele Nachfolger, aber nur einen Vorgänger haben – so wie die Klassen in Object Pascal).

Weitere Möglichkeiten, die Grafik automatisch anzuordnen oder Teilbäume ein- und auszublenden, finden Sie im lokalen Menü der Grafikelemente. Klicken Sie dazu ein grafisches Element direkt mit der rechten Maustaste an und wählen Sie den gewünschten Menüpunkt. Das lokale Menü eines Grafikelements enthält auch Optionen, mit denen Sie die eingehenden und ausgehenden Verbindungslinien lösen können.

Knoten und Kanten

Wie jede Baumstruktur setzt sich auch die Baumstruktur des TreeDesigners aus *Knoten* und *Kanten* zusammen. Ein Knoten (engl. *Node*) ist in diesem Fall einfach eines der Grafikelemente und wird somit durch die Klasse *TGraphicElement* repräsentiert. Eine Kante (engl. *Edge*) ist die Verbindung zwischen zwei Knoten. Für sie definiert die Unit *GrDoc* eine neue Klasse:

```
TEdge = class
  StartNode, EndNode: TGraphicElement;
public
  procedure SaveToStream(Stream: TStream);
  constructor LoadFromStream(Stream: TStream; List: TList);
  constructor StandardConnection(n1, n2: TGraphicElement);
  function GetEndNode: TGraphicElement;
end;
```

Die einzigen Daten einer Kante sind also die beiden Knoten, die verbunden werden sollen. Mit dieser allgemeinen Struktur ließen sich die verschiedensten in der Informatik verwendeten Typen von Graphen definieren, im TreeDesigner müssen diese theoretischen Möglichkeiten aus Gründen der Komplexität jedoch etwas eingeschränkt werden:

- Zunächst einmal behandelt der TreeDesigner den Graphen als *gerichteten Graphen* – das heißt, dass in jeder Verbindung ein Knoten der Ausgangsknoten (*StartNode*) und der andere der Endknoten (*EndNode*) ist. Bei der Darstellung der Kassenhierarchie ist die Basisklasse jeweils der Anfangsknoten, die davon ausgehenden Verbindungen führen in *Richtung* der abgeleiteten Klassen, die die Endknoten bilden.
- Als zweite Beschränkung darf jeder Knoten nur einen Vorgängerknoten haben, so wie jede Klasse in Object Pascal nur eine Vorfahrklasse haben kann. Die Zahl der Nachfolger ist nur durch die Kapazität der VCL-Klasse *TList* beschränkt.

An diesen beiden Beschränkungen führt im TreeDesigner kein Weg vorbei. Falls Sie versuchen, einem Knoten, der schon einen Vorgängerknoten hat, einen weiteren Vorgängerknoten zuzuweisen, erzeugt die Klasse *TEdge* eine Exception.

Für die automatische Anordnung der verknüpften Elemente gelten weitere Regeln: Es werden jeweils nur die Elemente angeordnet, die von einem bestimmten, von Ihnen ausgewählten Wurzel-Element aus erreichbar sind. Darüber hinaus sollte die Grafik keine Zyklen haben, d.h. dass kein Grafikelement sein eigener Vorgänger sein sollte. Dies ist immer sichergestellt, wenn die Grafik aus einem normalen Hierarchiebaum besteht. Da jeder Knoten nur einen Vorgänger haben darf, könnte in einem solchen Baum nur ein Zyklus entstehen, wenn Sie die Wurzel an einen ihrer Nachfolger anhängen (bei der Klassenhierarchie würde das bedeuten, die Klasse *TObject* beispielsweise von *TForm* abzuleiten).

Wenn Sie Grafikelemente manuell verknüpfen, überprüft der TreeDesigner *nicht*, ob dadurch Zyklen entstehen. Bei der Darstellung der Baumstruktur spielt es keine Rolle, ob Zyklen vorhanden sind (da die Darstellung nicht rekursiv erfolgt). Rekursiv arbeiten allerdings die Methoden zum Anordnen des Baums. Sie vermeiden eine Endlosschleife durch Verwendung der Variable *Visited* und ignorieren einen Zyklus stillschweigend.

Die Erweiterungen von TGraphicElement

Die wichtigsten neuen Datenelemente von *TGraphicElement* sind die Liste der Verbindungen zu den Nachfolgerknoten (*Edges*) und der Zeiger auf den Vorgängerknoten (Property *PrevNode*). Die Liste *Edges* enthält Objekte der Klasse *TEdge*, um also an einen Nachfolgerknoten zu kommen, müssen Sie aus einem dieser *TEdge*-Objekte das Element *EndNode* auslesen:

```
EndNode := TEdge(Edges[i]).EndNode;
```

Diese scheinbar umständliche Speicherung der Verbindungsdaten ermöglicht es, später weitere Daten mit jeder Verbindung zu speichern. So könnte die Klasse *TEdge* beispielsweise um Daten wie die Farbe der Verbindungslinie oder die Option, einen Pfeil statt einer Linie zu zeichnen, erweitert werden.

Damit man auch ohne den oben gezeigten komplizierten Ausdruck sofort an einen Nachfolgerknoten gelangen kann, definiert *TGraphicElement* außerdem das Array-Property *ChildNodes*, zu finden im folgenden Listing:

```
TGraphicElement = class(TPersistent)
 { Auszug der Elemente, die für die Baumstruktur
   hinzugefügt wurden }
private
  Edges: TList; { Liste der TEdges, die hiermit verknüpft sind }
  PreviousNode: TGraphicElement;
                { ansprechbar unter dem Property PrevNode }
  Visited: Boolean; { Endlosschleifen-Blocker }
  FCollapsed: Boolean; { True, wenn Verbindung zu
           Nachfolgerknoten nicht gezeichnet wird }
```

```
(* Methoden für Properties *)
  procedure SetPreviousNode(N: TGraphicElement);
  function GetChildNode(Index: Integer): TGraphicElement;
public
  procedure AddNode(n: TEdge);
  property PrevNode: TGraphicElement read PreviousNode
              write SetPreviousNode; { Vorgängerknoten }
  property Collapsed: Boolean read FCollapsed write FCollapsed;
  { Zahl der Nachfolgerknoten }
  property OutCount: Integer read GetChildCount;
  { Array der Nachfolgerknoten }
  property ChildNodes[Index: Integer]: TGraphicElement
                                        read GetChildNode;
(* Speichern und Laden der Verbindungsdaten *)
  procedure SaveEdgesToStream(Stream: TStream);
  procedure LoadEdgesFromStream(Stream: TStream; Items: TList);
(* Funktionen zum Anordnen *)
  function ArrangeTree(var t: TTreeFormat;
     RangeStart, RangeEnd, Depth: Integer): Boolean;
   {  TTreeFormat ist ein Record, der das Format des Baums angibt:
      horizontal/vertikal, Ebenenabstand, Ebenentiefe und
      Knotenbreite }
  procedure LocalExpand;
  procedure MoveTree(XTranslate, YTranslate: Integer);
  { Anzahl der Blätter dieses Teilbaums zählen: }
  function CountLeafs: Integer;
(* Verbindungs-Abbau *)
  procedure DisconnectInEdge;
  procedure DeleteInEdges;
  procedure DeleteOutEdges;
  procedure DeleteAllEdges;
  procedure DisconnectFrom(Item: TGraphicElement);
end;
```

Verbindungsaufbau

Alle Variablen, die Daten der Verbindungen speichern (*TEdge.Start/EndNode*, *TGraphicElement.PreviousNode* und *TGraphicElement.Edges*), sind als privat deklariert, so dass Sie eine neue Verbindung nur über den *TEdge*-Konstruktor *StandardConnection* aufbauen können, wobei Sie diesem die zu verbindenden Objekte als Parameter übergeben:

```
{Connection := }TEdge.StandardConnection(Vorgaenger, Nachfolger);
```

Das neu erstellte *TEdge*-Objekt *Connection* ist hier auskommentiert, weil es nicht extra gespeichert zu werden braucht: Es trägt sich bereits selbst in der *TEdge*-Liste des Vorgängerknotens ein. Falls *Nachfolger* bereits einen Vorgänger hat, kommt es in der obigen Anweisung zu einer *ETreeTypeError*-Exception und das *TEdge*-Objekt wird automatisch wieder freigegeben.

An diesem Verbindungsaufbau sind die folgenden beiden Methoden beteiligt:

```
procedure TGraphicElement.SetPreviousNode(N: TGraphicElement);
begin
  if Assigned(PreviousNode) then
    raise ETreeTypeError.Create('Eingangsknoten schon vorhanden');
  PreviousNode := N;
end;

constructor TEdge.StandardConnection(n1, n2: TGraphicElement);
begin
  { folgende Zuweisung kann eine Exception auslösen:
    Falls das der Fall ist, wird der Rest des Konstruktors
    übersprungen inklusive des Aufrufs StartNode.AddNode.
    Es ist kein try..finally-Block notwendig. Das Objekt self
    wird im Fehlerfall automatisch wieder freigegeben. }
  n2.PrevNode := n1; { SetPreviousNode wird aufgerufen }

  StartNode := n1;
  EndNode := n2;
  StartNode.AddNode(self);
end;
```

Anordnen

Als Beispiel für die Handhabung der Baumstruktur soll hier die Methode *LocalExpand* dienen, die alle Nachfolgerknoten des Grafikelements links neben diesem in gleichmäßigen Abständen anordnet (Abbildung 5.4). Sie können diese zur Laufzeit über das Popup-Menü der Grafikelemente aufrufen.

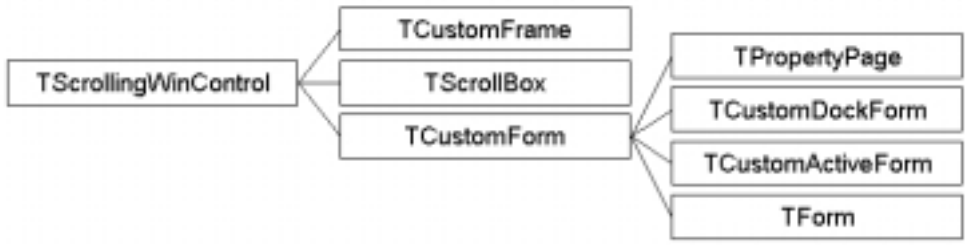


Abbildung 5.4: Das Ergebnis eines zweifachen Aufrufs der Methode LocalExpand

LocalExpand muss neben einigen Berechnungen im Wesentlichen nur alle *Edges.Count* Nachfolgerknoten durchlaufen, die sie über das Property *ChildNodes* einfach ansprechen kann. Jeder Knoten erhält mit *SetNewRect* seine neue Position und Größe.

```
procedure TGraphicElement.LocalExpand;
{ nur für vertikal angeordnete Bäume }
```

```
var
  Middle, { vertikale Mitte von Self }
  RangeStart, RangeEnd, { vertikaler Bereich für die Nachfolger }
  CellHeight, CellStart, CellEnd, { Bereich für einen einzelnen
              Nachfolger, mit Zwischenraum zwischen den Rechtecken}
  RectHeight, RectWidth, { Daten des Nachfolger-Rechtecks }
  RectDist, { Zwischenraum zwischen Bereichsgrenze und Rechteck }
  i: Integer;
begin
  FCollapsed := False;
 { vorbereitende Berechnungen }
  Middle := Top+(Bottom-Top) div 2;
  CellHeight := trunc((Bottom-Top)*1.2);
  RangeStart := Middle-Edges.Count*CellHeight div 2;
  RangeEnd := Middle+Edges.Count*CellHeight div 2;
  { eigene Breite und Höhe für den Nachfolgerknoten übernehmen: }
  RectHeight := Bottom-Top;
  RectWidth := Right-Left;
  RectDist := (CellHeight-RectHeight) div 2;
 { Durchlaufen aller Nachfolgerknoten }
  for i := 0 to Edges.Count-1 do begin
    CellStart := RangeStart+i*CellHeight;
    CellEnd := RangeStart+(i+1)*CellHeight;
    { Nachfolger des Nachfolgers verstecken: }
    ChildNodes[i].FCollapsed := True;
    { Nachfolgerknoten anordnen }
    ChildNodes[i].SetNewRect(
      Rect(Right+20, CellStart+RectDist,
           Right+20+RectWidth, CellEnd-RectDist));
  end;
end;
```

LocalExpand berechnet zu Beginn den vertikalen Platzbedarf der Nachfolgerknoten. Die Knoten werden vertikal innerhalb des Bereichs *RangeStart* bis *RangeEnd* angeordnet, wobei *LocalExpand* für jeden Nachfolgerknoten das 1,2-fache seiner eigenen Höhe berechnet (*CellHeight*). Darin ist bereits der vertikale Zwischenraum zwischen den Nachfolgerknoten enthalten (der halbe Zwischenraum nach oben und nach unten beträgt *RectDist* Einheiten).

Zeiger speichern *R109*

Problematisch ist es, die Baumstruktur auch in einer Datei zu speichern, denn die Daten innerhalb der *TEdge*-Objekte weisen auf *TGraphicElement*-Objekte, die bereits innerhalb der Objektliste von *TGraphicDoc* gespeichert werden. Um eine doppelte Speicherung zu vermeiden, muss statt des Objektinhalts eine Art von Adresse des Objekts gespeichert werden. Da die Speicheradresse eines Objekts jedoch nicht vorhersehbar ist, brauchen wir hier eine sicherere Adresse. Als solche bietet sich der Index des

Objekts innerhalb der Liste *TGraphicDoc* an, der sich beim Einlesen nicht ändert, da das Einlesen in derselben Reihenfolge stattfindet und die Elemente mit *Add* immer am Schluss der Liste angehängt werden (Methode *TGraphicDocument.LoadFromFile*).

Wenn nun das Element mit dem Index 10 gespeichert wird und eine Verbindung zum Element an Position 20 vorhanden ist, kann diese Verbindung beim Lesen nicht sofort hergestellt werden, weil Element 20 zum Lesezeitpunkt von Nummer 10 noch gar nicht eingelesen wurde. Daher findet das Lesen in zwei Durchläufen statt: Zuerst werden die Grafikelemente ohne die Verbindungsdaten geschrieben. Dann folgt ein zweiter Durchlauf durch alle Grafikelemente, allerdings werden nun nur die Verbindungsdaten gespeichert:

```
{ Ausschnitt aus TGraphicDoc.WriteToStream }
  for i := 0 to count-1 do
    TGraphicElement(Items[i]).SaveToStream(Stream, i);
  for i := 0 to count-1 do
    TGraphicElement(Items[i]).SaveEdgesToStream(Stream);
```

SaveEdgesToStream bewirkt, dass jede Ecke der *Edges*-Liste sich mit ihrer *SaveToStream*-Methode selbst im Stream speichert. Diese Speicherung sieht so aus, dass statt des Zeigers auf die beiden verbundenen Knoten die beiden Indizes dieser Knoten in den Stream geschrieben werden. Dies ist auch der Grund dafür, dass die Methode *TGraphicElement.SaveToStream* immer einen Index als Parameter erhält, den sie sich dann in der Variablen *FileIndex* merkt.

Zwar stimmt dieser Index beim Speichern einer Datei immer mit dem Index des Grafikelements in der *TList*-Listenstruktur überein, müsste also in diesem Fall gar nicht in einer extra Variablen abgelegt werden. Sollten aber einmal nicht alle Objekte des Dokuments in die Datei geschrieben werden, sondern z.B. nur die markierten Objekte, so weichen Dateiindex und Listenindex voneinander ab, daher wird hier die Variable *FileIndex* benötigt. Im TreeDesigner tritt dieser Fall ein, wenn die markierten Objekte in die Zwischenablage kopiert werden (hierfür werden die markierten Objekte mitsamt ihrer Verbindungen in einen MemoryStream geschrieben).

5.4 Mausaktionen und Zeichnen

Dieses Kapitel beschreibt die Durchführung von Mausziehaktionen, bei denen es nicht um Drag&Drop geht, sondern um das Zeichnen eines Objekts oder einer Grafik. Derartige Mausaktionen kommen nicht nur in Grafik- und Malprogrammen, sondern beispielsweise auch in der Delphi-IDE beim Einzeichnen der Komponenten in das Formular vor.

Da das Zeichnen mit der Maus eine etwas komplexere Aktion als das Betätigen eines Schalters ist, müssen Sie drei verschiedene Ereignisse bearbeiten, um den gesamten Vorgang kontrollieren zu können.

Die Ereignisse des Mauszieh-Vorgangs *R80*

In einem Zeichenprogramm, bei dem ein neues Objekt erst dann erstellt wird, wenn der Benutzer die Maustaste loslässt, können Sie die drei Ereignisse wie folgt bearbeiten:

- Wenn Sie beim Ereignis *OnMouseDown* die Position der Maus speichern, können Sie diese beim Abschluss der Aktion als Startpunkt für ein neu erstelltes Grafikobjekt verwenden.
- Während der Mausbewegung erhält der Benutzer normalerweise eine visuelle Rückmeldung über die Größe des eingezeichneten Objekts. Hierfür können Sie bei jedem *OnMouseMove*-Event einen invertierten Rahmen, ein invertiertes Rechteck oder einen anderen grafischen Hinweis im Formular ausgeben.
- Wenn schließlich das *OnMouseUp*-Ereignis anzeigt, dass die Maustaste wieder in ihren Ausgangszustand zurückgekehrt ist, können Sie die provisorische Größenanzeige, die bei den *OnMouseMove*-Events ausgegeben wurde, wieder entfernen und durch die endgültige Grafik ersetzen.

Etwas anders verhält es sich beim Freihandzeichnen in einem Malprogramm, denn dort wird normalerweise bei jeder Mausbewegung eine neue endgültige Linie gezeichnet.

MouseCapture

Normalerweise tritt das *OnMouseMove*-Ereignis nur bei dem Fenster auf, über dem sich die Maus bewegt hat. Da ein Zeichenvorgang jedoch nicht dadurch unterbrochen werden soll, dass der Benutzer die Maus über die Fenstergrenze hinwegbewegt, wird die Maus während des Zeichenvorgangs üblicherweise für das Fenster, in dem die Zeichnung stattfindet, reserviert.

Die VCL erledigt diese Reservierung automatisch bei jedem *OnMouseDown*-Event und hebt sie beim *OnMouseUp*-Event wieder auf. Ob die Mausnachrichten gerade reserviert sind, können Sie über das Property *MouseCapture* des Formulars in Erfahrung bringen.

5.4.1 Die Methoden im TreeDesigner

Nachdem in Kapitel 5.3 die TreeDesigner-Unit *GrDoc* im Mittelpunkt des Interesses stand, wenden wir uns jetzt der Unit *DocForm* zu. Die folgenden Variablen der Formularklasse *TDocumentForm* sind am Ziehvorgang beteiligt:

- *DragStartX* und *DragStartY* speichern die Position, bei der der Mauszieh-Vorgang begonnen hat.
- *LastX* und *LastY* speichern die letzte Position der Maus im Laufe der Bewegung.
- *MouseObject* enthält das Grafikobjekt, das gerade erzeugt wird (bzw. später das Objekt, das von der Maus angeklickt wurde).
- *DragAction* gibt an, ob eine für den TreeDesigner interessante Ziehoperation aktiv ist.
- *DragMode* enthält eine Konstante, die die Art des Ziehvorgangs bezeichnet: *dmCreate* (Erzeugen eines neuen Elements), *dmMark* (Aufziehen eines Rahmens zur Markierung von Elementen), *dmMove* (Verschieben eines einzelnen Elements), *dmMoveTree* (Verschieben eines Elements mit allen als Kind verbundenen Elementen) oder *dmResize* (Änderung der Größe eines Elements).
- *ResizeMode* gibt im Fall von *DragMode=dmResize* an, welcher Punkt des Grafikelement-Rahmens von der Maus zur Änderung der Größe gegriffen wurde.

Da im TreeDesigner nicht direkt auf dem Formular, sondern auf einer *TPaintBox*-Komponente namens *PaintBox* gezeichnet wird, werden nur die drei Mausereignisse dieser Komponente bearbeitet.

PaintBoxMouseDown

Die insgesamt sehr umfangreichen Methoden von *TDocumentForm* sind an dieser Stelle auf die Teile beschränkt, die für das Einzeichnen neuer Objekte verantwortlich sind. Das hier beschriebene Einzeichnen eines neuen Objekts wird nur dann ausgeführt, wenn in der Werkzeugleiste eine der grafischen Formen gewählt ist bzw. wenn der Zeigerschalter nicht gedrückt ist (*SpeedBtnArrow.Down=False*). Das Gerüst der Methode für das *OnMouseDown*-Ereignis sieht wie folgt aus:

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  // Einstellung des virtuellen Koordinatensystems und Umrechnen
  // der Mausposition, siehe Kapitel 5.6.3
  SetMapMode(PaintBox.Canvas, PaintBox);
  MapMousePos(x, y);
  // Speichern der Mausposition für OnMouseMove/OnMouseUp
  DragStartX := x;
  DragStartY := y;
  LastX := x;
  LastY := y;
  if Button = mbRight then begin
    { Code für den Klick mit der rechten Maustaste, siehe CD }
  end else
```

```
      DragAction := True;

    if SpeedBtnArrow.Down then begin
      { Code für den Markierungsmodus, folgt später }
    end else if DragAction begin
      { Hier startet das Neuzeichnen eines Grafikelements }
      DragMode := dmCreate;
      Document.DeselectAll;
      ShapeText.Text := ''; { Platz machen für neue Beschriftung }
      MouseObject :=
        TGraphicElement.Create(Document, Rect(x, y, x, y), CurShapeType);
      SetAttributes(MouseObject);
      MouseObject.Text := ShapeText.Text;
      MouseObject.Marked := True;
    end;
    if DragAction then begin
      MouseObject.XorPaint(Paintbox.Canvas,
                      Rect(DragStartX, DragStartY, X, Y));
    end;
  end;
```

Die Methode hält sich an den oben beschriebenen Ablauf, speichert also die Mausposition in den Variablen *DragStartX* und *DragStartY*. Sie erstellt bereits ein neues Grafikobjekt und stellt dessen Attribute ein, wie in Kapitel 5.6.3 beschrieben.

Zeichnen beim Ziehen der Maus *R96*

Die Methode *XorPaint* der Grafikelemente zeichnet eine besondere Version des Objekts in der Art, dass sich zwei gleiche Aufrufe dieser Methode auf der Zeichenfläche gegenseitig aufheben. Damit sich zwei gleiche Zeichenaktionen gegenseitig aufheben, müssen Sie lediglich das Property *Canvas.Pen.Mode* auf *pmNotXor* einstellen. Dies wirkt sich dann auch auf die mit dem *Canvas.Brush* gezeichneten Bereiche aus (so z.B. die Füllung eines Rechtecks).

XorPaint setzt also die Zeichenattribute des *Canvas*-Objekts auf die Attribute des Grafikelements und passt *Pen.Mode* an. Dann ruft es die Prozedur *PaintShape* auf, die die grafische Form (ohne Text) ausgibt. Wie die anderen Zeichenmethoden von *TGraphicElement*, benötigt auch *XorPaint* einen Parameter *CanvasRef*, in dem die Zeichenfläche angegeben wird, auf der das Element ausgegeben werden soll:

```
procedure TGraphicElement.XorPaint(CanvasRef : TCanvas; XorRect : TRect);
var
  OldPenMode: TPenMode;
begin
  { Pen.Mode muss am Schluss wiederhergestellt werden,
    damit es bei anderen Zeichenvorgängen nicht zu Störungen kommt. }
  OldPenMode := CanvasRef.Pen.Mode
  { Vorbereitungen, damit sich zwei Zeichnungen gegenseitig aufheben: }
  CanvasRef.Brush := Brush;
```

```
  CanvasRef.Pen := Pen;
  CanvasRef.Pen.Mode := pmNotXor;
  { Ausgabe der Grafik, siehe Kapitel 5.5.2: }
  PaintShape(CanvasRef, ShapeType, XorRect, Document.DrawEnhanced);
  { Canvas-Attribute wieder zurücksetzen: }
  CanvasRef.Pen.Mode := OldPenMode;
end;
```

Hinweis: Um die Füllung der Objekte während des Ziehvorgangs abzuschalten, können Sie *Brush.Style* auch auf *bsClear* setzen.

Bewegung der Maus

PaintBoxMouseMove sorgt bei jeder Mausbewegung für die schon beschriebene Aktualisierung der von *XorPaint* gezeichneten Grafik und speichert die neue Mausposition in *LastX/LastY*:

```
procedure TDocumentForm.PaintBoxMouseMove
  (Sender: TObject; Shift: TShiftState; X, Y: Integer);
begin { vereinfachte Version }
  (... Einstellung des virtuellen Koordinatensystems weggelassen ...)
  MapMousePos(x, y);
  // Nebenbei: Anzeige der aktuellen Koordinaten in der Statuszeile:
  Paintbox.Hint := Format('%d : %d', [x, y]);
  if DragAction then begin
    { Aufheben des alten Rechtecks }
    MouseObject.XorPaint(Paintbox.Canvas, Rect
                  (DragStartX, DragStartY, LastX, LastY));
    { Zeichnen des neuen Rechtecks }
    MouseObject.XorPaint(Paintbox.Canvas, Rect
                  (DragStartX, DragStartY, X, Y));
    end;
    LastX := X;
    LastY := Y;
  end;
end;
```

Hinweis: In der vollständigen Version ist *PaintBoxMouseMove* auch dafür zuständig, die Mauszeigerform anzupassen, wenn die Maus sich über einem Größenänderungsgriff eines Objektrahmens befindet; ferner sorgt sie beim Verschieben mehrerer gleichzeitig markierter Objekte für eine Vorschau all dieser Objekte (*XorPaint* muss dabei für jedes verschobene Objekt aufgerufen werden). Eine vollständige Vorschau beim Verschieben von Teilbäumen ist allerdings noch nicht implementiert.

Fertigstellung des Objekts

Schließlich kann *PaintBoxMouseUp* dem Grafikobjekt mit *SetNewRect* seine endgültige Größe zuweisen. *SetNewRect* sorgt auf dem Weg der Ungültig-Erklärung (siehe Kapitel 5.5.3) dafür, dass das Objekt sich in seiner endgültigen Gestalt ausgibt.

```
procedure TDocumentForm.PaintBoxMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin { vereinfachte Version }
  (... Einstellung des virtuellen Koordinatensystems weggelassen ...)
  if DragAction then begin
    MapMousePos(x, y); { nur für virtuelle Koordinatensysteme }
    { letzte XOR-Zeichnung aufgeben: }
    MouseObject.XorPaint(PaintBox.Canvas, Rect
                  (DragStartX, DragStartY, LastX, LastY));
    MouseObject.SetNewRect(Rect(DragStartX, DragStartY, X, Y));
    if DragMode = dmCreate then
      Document.Add(MouseObject);
    DragAction := False;
  end;
end;
```

5.4.2 Shift und Variationen eines Ziehvorgangs

Die im letzten Abschnitt gezeigten Methoden werden jetzt noch etwas komplizierter, denn auch die Mausaktionen zum Objekt-Verschieben und zum Ändern der Größe hängen mit den drei schon bearbeiteten Ereignissen zusammen.

Außerdem kommt ein weiterer *SpeedButton* ins Spiel, und zwar der mit dem Zeigersymbol versehene (Name der Komponente: *SpeedBtnArrow*). Wenn dieser selektiert ist, wird die Maus zum Manipulationswerkzeug bestehender Objekte, während sie bei allen anderen Schaltern als Zeichengerät funktioniert (ähnlich wie die Schalter in Delphis Komponentenpalette mit dem Unterschied, dass Delphi normalerweise nach jedem neu gezeichneten Objekt wieder zurück zum Zeigersymbol schaltet).

Die folgende Tabelle zeigt, wie Sie die Mausaktion im Manipulationsmodus mit zusätzlichen Tasten der Tastatur beeinflussen können. (Der TreeDesigner beachtet nur, welche Tasten beim Drücken der Maustaste gedrückt sind. Wenn Sie während des Ziehvorgangs weitere Tasten drücken oder loslassen, so hat das keinen Einfluss mehr.)

Tasten	Bedeutung	Ziehvorgang
keine	Wenn *MouseDown* innerhalb eines Objekts stattfindet: Einzelnes Objekt markieren und mitsamt den verbundenen Nachfolger-Objekten verschieben. Wenn *MouseDown* auf der Randmarkierung eines Objekts stattfindet: Größe des Objekts ändern. Wenn *MouseDown* außerhalb aller Objekte stattfindet: Einen Bereich zum Markieren mehrerer Objekte aufziehen.	ja

Tasten	Bedeutung	Ziehvorgang
Shift	Mehrere Objekte markieren und angeklicktes Objekt verschieben, ohne die verbundenen Nachfolger-Objekte mitzubewegen.	ja
Strg	das angeklickte Objekt mit allen bisher markierten Objekten verbinden.	nein

Da die Taste Alt für Systemfunktionen vorgesehen ist (ihre Betätigung lässt den Tastaturfokus zum System- oder Hauptmenü wandern), kann sie sich übrigens nicht an diesen Variationen beteiligen, aber Shift und Strg sind ja für die Zwecke des Tree-Designers bereits ausreichend.

Shift und TShiftState

Bei der Umsetzung der obigen Pläne in lauffähigen Quelltext hilft der Parameter *Shift*, der bei jedem der drei Mausereignisse an die Bearbeitungsmethode gesendet wird. Er gibt den Status der Tasten Shift, Strg und Alt in Form einer Menge des Typs *TShiftState* an. Um zu überprüfen, ob die Tasten Shift und Strg beim Drücken der Maustaste ebenfalls gedrückt waren, testen die Ereignismethoden, ob die Konstanten *ssShift* bzw. *ssCtrl* in dieser Menge enthalten sind (neben diesen Werten und *ssAlt* kann *Shift* auch noch die Flags *ssLeft*, *ssRight* und *ssMiddle* enthalten, die besagen, welche der Maustasten gedrückt sind).

Angeklicktes Objekt herausfinden *R81*

Im Manipulationsmodus ist es besonders wichtig, herauszufinden, welchem Grafikobjekt der Mausklick gegolten hat, denn dieses Grafikobjekt wird in den meisten Fällen markiert und in der folgenden Aktion womöglich noch verschoben oder anderweitig manipuliert. *TGraphicDoc* stellt dazu die Methode *GetObjectByPoint* zur Verfügung, die herausfindet, welches Objekt sich an einer gegebenen Position befindet:

```
function TGraphicDoc.GetObjectByPoint(P: TPoint): TGraphicElement;
var
  i: Integer;
begin
  Result := nil;
  { von oben nach unten durchsuchen }
  for i := Count-1 downto 0 do
    if Items[i].PointInShape(P) then begin
      Result := Items[i];
      exit;
    end;
end;
```

Da die Objekte am Anfang der Liste eventuell durch nachfolgende Objekte verdeckt werden, durchsucht *GetObjectByPoint* die Liste der Objekte von hinten nach vorne. Sie findet so das Objekt, das von allen an der Position *P* befindlichen Objekten zu oberst liegt.

Um herauszufinden, ob ein Objekt den Punkt *P* enthält, bedient sich *GetObjectByPoint* einer weiteren *TGraphicElement*-Methode: *PointInShape*. Diese wiederum verwendet die Windows-API-Funktion *PtInRect*, um die gestellte Frage zu beantworten:

```
function TGraphicElement.PointInShape(TestPoint : TPoint) : Boolean;
{ Stellt fest, ob TestPoint im Grafikelement liegt, wird
  verwendet, um das zu einem Mausklick passende Objekt zu finden. }
var
  TestRect : TRect;
begin
  GetNormalizedRect(TestRect); { Reihenfolge von Left/Right
                                 und Top/Bottom überprüfen }
  Result := PtInRect(TestRect, TestPoint);
end;
```

Markierungspunkte zur Größenänderung *R82*

Aus dem Listing der Klassendeklarationen von *TGraphicElement* und *TGraphicDoc* ging bereits hervor, dass es von *GetObjectByPoint* noch eine zweite Version gibt:

```
function GetObjectByPoint(P: TPoint; GripSize: Integer;
                      var rm: TResizeMode) : TGraphicElement; overload;
```

Diese Variante gibt im dritten Parameter zurück, welcher der Markierungspunkte zur Größenänderung angeklickt wurde (oder *rmNone*, falls kein Punkt angeklickt wurde), und benötigt dafür als Eingabe im Parameter *GripSize* die Größe eines Markierungspunkts. Zur Erinnerung noch einmal die Deklaration von *TResizeMode*:

```
type TResizeMode = (rmNone, rmRB, rmLB, rmRT, rmLT, rmR, rmL, rmT, rmB);
                                    // RLBT = Right/Left/Bottom/Top
```

Die mit diesen Markierungspunkten zusammenhängenden Programmteile können hier aus Platzgründen nicht abgedruckt werden, die folgenden Zeilen sollen jedoch eine Idee von der groben Funktionsweise vermitteln. Sie stammen aus einer zweiten Version von *PointInShape*, wo sie direkt auf die beiden Anweisungen folgen, die oben schon für die einfache Version von *PointInShape* gezeigt wurden:

```
if Marked then begin // Markierungspunkte nur sichtbar, wenn markiert
  for GripIterator := TResizeMode(1) to high(TResizeMode) do begin
    if PtInRect(GetGripRect(GripIterator, GripSize), TestPoint) then begin
      rm := GripIterator;
      Result := true;
      exit;
    end;
```

```
    end;
  end;
```

Wichtigstes Element ist die Hilfsfunktion *GetGripRect*, die für einen übergebenen *TResizeMode*-Wert und eine Größenangabe für die Markierungspunkte die Koordinaten für das entsprechende Markierungsrechteck berechnet. Diese Koordinaten werden beim Zeichnen der Markierung (siehe CD) oder in diesem Fall für das Überprüfen der Mausposition verwendet.

Der TreeDesigner-Quelltext

Da die vollständigen Methoden etwas umfangreich sind, soll hier ein Auszug aus der *OnMouseDown*-Bearbeitung genügen. Das angeklickte Objekt wird dabei in *MouseObject* gespeichert:

```
{ Aus TDocumentForm.PaintBoxMouseDown: }
{ Mausposition in logische Koordinaten umwandeln }
MapMousePos(x, y);
{ Objekt an logischer Klick-Position feststellen }
MouseObject := Document.GetObjectByPoint(Point(X, Y), GetGripSize,
                                         ResizeHandle);
...
if SpeedBtnArrow.Down then begin
  if MouseObject = nil then begin
    { Es wurde auf die leere Zeichenfläche geklickt. }
    if not (ssShift in Shift) then
      Document.DeselectAll; { alle Objekte deselektieren }
    DragMode := dmMark; { Markierung mehrerer Objekte durch
                          Aufziehen eines Rechtecks. }
  end
  else if Shift = [ssLeft] then begin
    { ein Objekt (MouseObject) wurde mit links angeklickt }
    if ResizeHandle <> rmNone then begin
      { ein Markierungspunkt zur Größenänderung wurde "getroffen" }
      Document.DeselectAll;
      DragMode := dmResize;
      ResizeMode := ResizeHandle;
    end else begin
      { das Innere eines Grafikelements wurde getroffen }
      if not MouseObject.Marked then begin
        { Objekt ist nicht markiert -> als Unterbaum verschieben }
        Document.DeselectAll;
        MouseObject.Marked := True;
        DragMode := dmMoveTree;
        LoadAttributes(MouseObject);
        Document.Exchange(Document.Count-1,
                          Document.IndexOf(MouseObject));
      end else
```

```
        { Anklicken eines bereits markierten Objekts -> Alle markierten
          Objekte verschieben }
        DragMode := dmMove;
    end;
  end else if Shift = [ssLeft, ssShift] then begin
    { Shift gedrückt -> kumulativ markieren und evtl. verschieben }
    MouseObject.Marked := not MouseObject.Marked;
    DragMode := dmMove;
    { weitere Anweisungen, die sicherstellen, dass bei einem markierten
      Objeke 8 Markierungspunkte zur Größenänderung, bei mehreren
      markierten Objekten jeweils nur 4 Punkte gezeichnet werden (wie
      im Formulardesigner der Delphi-IDE) siehe CD }
  end else if Shift = [ssLeft, ssCtrl] then begin
... { Strg-Taste -> Verbindung definieren }
    ...
  end else begin
 { SpeedBtnArrow.Down = False -> der TreeDesigner befindet sich im
   Zeichenmodus. Das Neuzeichnen eines Objekts wurde bereits
   in Kapitel 5.5.2 gezeigt... }
  ...
```

5.5 Grafikausgabe und Scrolling

Dieses Kapitel setzt die in Kapitel 4.4 beschriebenen Grundlagen der Grafikausgabe und der Klasse *TCanvas* voraus und beschreibt die Realisierung einer zunächst nur scrollbaren Grafik am Beispiel des TreeDesigners. Im folgenden Kapitel 5.6 wird die scrollbare Zeichenfläche durch ein virtuelles Koordinatensystem erweitert, mit dessen Hilfe sich die Grafik vergrößern und verkleinern lässt.

5.5.1 Ereignisgesteuerte Grafikausgabe

Unter Windows und anderen GUI-Systemen findet auch die Ausgabe von Grafik ereignisgesteuert statt, wobei die Grafik normalerweise nicht sofort ausgegeben wird, sobald eine Änderung eingetreten ist, sondern erst dann, wenn ein Paint-Ereignis die Anwendung explizit zum Zeichnen der Grafik auffordert. Die folgenden Situationen bzw. Ereignisse machen die Neuausgabe der Grafik erforderlich:

- Mausereignisse, die auftreten, wenn Sie die Grafik direkt mit der Maus verändern, im TreeDesigner also entweder neue Objekte einzeichnen oder bestehende manipulieren.
- Befehlsereignisse, die die Anwendung dazu veranlassen, die Grafik zu verändern. So bieten z.B. Bildverarbeitungsprogramme verschiedene Filter und Effekte an, und Textverarbeitungsprogramme stellen automatische Formatierfunktionen zur

Verfügung. Die vergleichbaren Funktionen des TreeDesigners sind die Funktionen zum Anordnen der verbundenen Grafikelemente in einer gleichmäßigen Baumdarstellung.

- Zeichenereignisse, die dann auftreten, wenn Teile der Grafik sichtbar werden, die bisher von einem anderen Fenster verdeckt waren oder am Rand des übergeordneten Fensters bzw. des Bildschirms abgeschnitten wurden. Da Windows diese verdeckten Teile in den meisten Fällen nicht speichert (Ausnahmen bilden Verdeckungen durch Menüs und spezielle Fenster), muss die Anwendung die Grafik neu ausgeben, auch wenn diese sich nicht geändert hat.

  Es genügt daher nicht, die Grafik nur bei den zuerst genannten Maus- und Befehlsereignissen anzuzeigen; der Aufbau der Grafik muss jederzeit rekonstruierbar sein. Bei dokumentbasierten Anwendungen beispielsweise wird die Grafik aus den Daten des Dokuments und gegebenenfalls aus weiteren Daten rekonstruiert. Zu diesen weiteren Daten gehören in einem Editor beispielsweise die Anfangs- und Endmarken der Textmarkierung, die sich in der grafischen Ausgabe in einer Hervorhebung der entsprechenden Zeilen und Zeichen auswirken.

Diese Aufzählung erhebt zwar keinen Anspruch auf allgemeine Vollständigkeit, deckt aber zumindest die im TreeDesigner vorkommenden Anlässe zur Grafikausgabe ab.

Die Ungültigerklärung *R91*

Sie können die Grafik zwar entsprechend der obigen Aufzählung in den Methoden für die Maus- und Befehlsereignisse (*OnClick*, *OnMouseDown*, etc.) direkt ausgeben, meistens ist es jedoch wesentlich effektiver, die Grafikausgabe nur in einer einzigen, mit dem Ereignis *OnPaint* verknüpften Methode zu implementieren. Bei Maus- und Befehlsereignissen, die die Grafik verändern, genügt es dann, indirekt ein solches *OnPaint*-Ereignis hervorzurufen, indem Sie den veränderten Bereich der Grafik für ungültig erklären. Hierzu gibt es zwei Möglichkeiten:

- Die Methode *TWinControl.Invalidate* erklärt den gesamten Bereich des Formulars bzw. der Komponente für erneuerungswürdig, was zur vollständigen Neuzeichnung führt und in vielen Fällen ein unvertretbar hoher Aufwand ist.
- Die Windows-API-Funktion *InvalidateRect* erklärt nur einen Teil des Fensters für ungültig. *InvalidateRect* erwartet als Parameter das Handle des *Canvas*-Objekts und eine *TRect*-Struktur, die den ungültigen Bereich angibt.

Nach dem Aufruf von *Invalidate* oder *InvalidateRect* prüft Windows, ob der ungültige Bereich überhaupt sichtbar ist. Wenn das der Fall ist, sendet Windows eine *WM_Paint*-Botschaft an die Anwendung, die von der VCL schließlich zu einem *OnPaint*-Ereignis umgewandelt wird. Kapitel 5.5.3 beschreibt den Aufruf von *InvalidateRect* im TreeDesigner.

Grafikausgabe außerhalb der *OnPaint*-Bearbeitung ist selten notwendig. Im TreeDesigner kommt sie nur während der Mausziehoperationen aus Kapitel 5.4 vor.

5.5.2 Zeichnen von Objekten und Text

Im TreeDesigner sind das Dokumentfenster, das *TGraphicDoc*-Objekt und die einzelnen Grafikobjekte an der Grafikausgabe beteiligt. Durch diese Aufteilung wird die Aufgabe für alle Beteiligten leicht überschaubar. Da die Grafikausgabe im TreeDesigner nicht auf der Fläche des Formulars, sondern in der Paintbox-Komponente stattfindet, bearbeitet die Formularklasse *TDocumentForm* nicht das eigene *OnPaint*-Ereignis, sondern das der Paintbox-Komponente.

Die Grafikausgabe beginnt also mit der Methode *PaintBoxPaint* der Klasse *TDocumentForm*. Diese muss zunächst das Koordinatensystem einstellen (siehe Kapitel 5.6.3) und teilt dann den weiteren Vorgang auf drei Methoden auf: *PaintBackground* (zeichnet den schattierten Umriss und die Fläche des Zeichenblattes), *PaintAll* (zeichnet die Objekte selbst, also den eigentlichen Grafikinhalt, wie er auch gedruckt wird) und *PaintAllMarks* (zeichnet nur die Markierungen von Objekten und soll im Folgenden, wie auch *PaintBackground*, nicht weiter beachtet werden):

```
procedure TDocumentForm.PaintBoxPaint(Sender: TObject);
var
  VCanvas: TVirtualCanvas;
begin
  VCanvas := TVirtualCanvas.Create(PaintBox.Canvas);
  Document.PaintBackground(VCanvas);
  Document.PaintAll(VCanvas);
  Document.PaintAllMarks(VCanvas);
end;
```

Bevor wir zu den weiteren Zeichenmethoden kommen, bedarf die im obigen Listing verwendete Klasse *TVirtualCanvas* einer Erklärung.

Erweiterung von TCanvas

TVirtualCanvas ist nicht Teil der VCL, sondern in der Unit *VCanvas* auf der CD definiert, und dient dazu, die von *TCanvas* bekannten und zusätzlich alle weiteren im TreeDesigner benötigten Grafikoperationen durch einfache Methodenaufrufe bereitzustellen. Gleichzeitig ist sie auch eine Schicht, die diese *TCanvas*-Erweiterungen bibliotheksunabhängig zur Verfügung stellt. Die Plattformunabhängigkeit könnte zwar bereits durch die seit Delphi 6 verwendbare Qt-Bibliothek gewährleistet werden, jedoch soll der TreeDesigner sich unter Windows mit der VCL ohne die Qt kompilieren lassen. Daher verwendet *TVirtualCanvas* unter Delphi Windows-API-Funktionen, unter Kylix Methoden von Qt.

Sie können dieses *TVirtualCanvas*-Objekt im Folgenden einfach als »die Zeichenfläche« ansehen, als Ersatz des vorgegebenen *TCanvas*-Objekts. Alle Methoden von *TCanvas* stehen auch in *TVirtualCanvas* zur Verfügung. Weitere Informationen zu dieser Klasse finden Sie in Kapitel 5.6.2.

PaintBoxPaint muss den drei *Document*-Methoden im obigen Listing die Zeichenfläche, also das *VirtualCanvas*-Objekt übergeben, auf dem sich das Dokument zeichnen soll, denn da die Klasse *TGraphicElement* (wie die gesamte Unit *GrDoc*) völlig unabhängig von jeglichen Formular-Units ist, kann sie natürlich nicht wissen, wo sich die Ausgabefläche befindet.

Zeichnen der Grafikelemente

Kommen wir nun wieder zurück zur Zeichenmethode. Da für *PaintBackground* und *PaintAllMarks* bereits auf die CD verwiesen wurde, bleibt noch die Methode *PaintAll*. Sie ruft lediglich die *Paint*-Methoden der einzelnen Grafikelemente auf:

```
procedure TGraphicDoc.PaintAll(Dest: TVirtualCanvas);
var
  i: Integer;
begin
  for i := 0 to Count-1 do
    Items[i].Paint(Dest);
end;
```

In *TGraphicElement.Paint* findet eine weitere Untergliederung in einzelne Teile der Grafikausgabe statt:

- Für alle Elemente werden zunächst die Pinsel- und Stiftattribute eingestellt.
- *PaintContent* mit dem zweiten Parameter *psBeforeFill* zeigt einen eventuellen zusätzlichen Inhalt des Grafikelements an. Wie in Kapitel 5.1.3 erwähnt, können Sie über das Popup-Menü des Grafikelements Bilddateien in das Element laden. Die Implementierung dieser Funktion soll hier nicht weiter untersucht werden, wichtigster Aufruf für die Grafikausgabe ist der Aufruf von *StretchDraw*, um die Bilddatei genau in die Umrisse des Elements einzupassen.
- Die Methode *PaintShape* gibt die eigentliche grafische Form aus, bestehend aus dem Umriss, der mit dem Stift, sowie der Füllung, die mit dem Pinsel gezeichnet wird. Dafür werden die in Kapitel 4.4.3 aufgeführten *TCanvas*-Methoden zum Zeichnen von Linien, (abgerundeten) Rechtecken und Ellipsen verwendet, so dass *PaintShape* für dieses Kapitel ebenfalls keine neue Herausforderung darstellen würde.
- *SetTextClipRegion* stellt eine Clipping-Region ein, durch die die Ausgabe des Textes auf den Bereich des Elements begrenzt wird. Dies wird Thema von Kapitel 5.5.5 sein.

- *PaintText* nimmt die eigentliche Textausgabe vor, diese Methode wird weiter unten abgedruckt.
- Mit *ClippingOff* (eine Methode der erweiterten *TVirtualCanvas*-Klasse) wird das vorher eingestellte Clipping wieder abgeschaltet. Es ist wichtig, dass dieser Aufruf in jedem Fall vor dem Ende der *Paint*-Methode stattfindet, weil sonst darauf folgende Zeichenmethoden an dem alten Clipping-Bereich abgeschnitten werden und somit wirkungslos bleiben. *ClippingOff* wird daher durch einen *try… finally*-Abschnitt gesichert.
- Der zweite Aufruf von *PaintContent* dient dem gleichen Zweck wie der erste, allerdings wird hier durch den Parameter *psAfterFill* angezeigt, dass dieser Aufruf nach dem Ausfüllen der Elementfläche durch die »reguläre« Grafik (*PaintShape* und *PaintText*) stattfindet. Grafikausgabe von *PaintContent* an dieser Stelle wird also die vorherigen Ausgaben übermalen. (Zurzeit zeichnet *PaintContent* übrigens nur den Zeitpunkt *psAfterFill*, der erste Aufruf von *PaintContent* mit *psBeforeFill* als Parameter dient lediglich als Ansatzpunkt für spätere Erweiterungen.)
- *PaintEdges* schließlich zeichnet die Verbindungslinien des Elements zu anderen Elementen. Kapitel 5.3.4 gibt eine kurze Einführung in diese Verbindungen, für den Code der Methode *PaintEdges* muss hier aber auf die CD verwiesen werden.

```
procedure TGraphicElement.Paint(CanvasRef : TVirtualCanvas);
begin
  CanvasRef.Brush := Brush;
  CanvasRef.Pen := Pen;

  PaintContent(CanvasRef, psBeforeFill);
  PaintShape(CanvasRef, ShapeType, Points);
  SetTextClipRegion(CanvasRef);
  try
    PaintText(CanvasRef);
  finally
    CanvasRef.ClippingOff;
  end;
  PaintContent(CanvasRef, psAfterFill);
  PaintEdges(CanvasRef);
end;
```

Polygone *R95*

Da das Windows-API für Dreiecke und alle Formen mit mehr als vier Ecken keine speziellen Ausgabefunktionen hat, müssen die spezielleren Formen des TreeDesigners in etwas aufwändiger Weise berechnet werden. *PaintShape* übergibt diese Arbeit den Prozeduren *PaintRhomb* und *PaintHexagon*. Als Beispiel sei hier die Methode *PaintHexagon* gezeigt, die ein Punkte-Array für das Sechseck berechnet und dieses mit der Methode *Polygon* ausgibt:

```
procedure PaintHexagon(CanvasRef: TCanvas; Points: TRect);
var
  dpoints: array[0..5] of TPoint; { Koordinaten des Sechsecks }
begin
{ Die Reihenfolge der Punkte:
        ____
       /0   1\
      5       2
       \4___3/
}
  dpoints[0].x := points.left+(points.right-points.left) div 3;
  dpoints[1].x := points.right-(points.right-points.left) div 3;
  dpoints[2].x := points.right;
  dpoints[3].x := dpoints[1].x;
  dpoints[4].x := dpoints[0].x;
  dpoints[5].x := points.left;

  dpoints[0].y := points.top;
  dpoints[1].y := points.top;
  dpoints[2].y := points.top+(points.bottom-points.top) div 2;
  dpoints[3].y := points.bottom;
  dpoints[4].y := points.bottom;
  dpoints[5].y := dpoints[2].y;
  CanvasRef.Polygon(dpoints);
end;
```

Die weiteren Methoden finden Sie auf der CD. Als Erweiterung wäre hier zu empfehlen, Formen mit beliebig vielen (gleichmäßig verteilten) Ecken zu implementieren, wobei die Eckenzahl beispielsweise in der Werkzeugleiste eingestellt werden könnte.

Textausrichtung *R97*

Nun wenden wir uns der Beschriftung der Grafikelement zu, die durch die Methode *PaintText* vorgenommen wird.

```
procedure TGraphicElement.PaintText(CanvasRef: TVirtualCanvas);
var
  R: TRect;
begin
  CanvasRef.SetTextAlign(TA_CENTER or TA_VCENTER);
  CanvasRef.Font := Font;
  CanvasRef.Font.Height := Font.Height;
  GetInnerRect(R);
  CanvasRef.Brush.Style := bsClear;
  CanvasRef.TextRect(R, Text);
end;
```

Die Textausgabemethoden von *TCanvas*, *TextOut* und *TextRect*, erwarten als Parameter die Koordinaten der linken oberen Ecke des Textes, was zur zentrierten Beschriftung des Rechtecks noch nicht ideal ist.

Wenn Sie die normale *TCanvas*-Klasse der VCL verwenden, können Sie die Ausrichtung des Textes ändern, indem Sie vorher die Windows-Funktion *SetTextAlign* aufrufen. Diese setzt das Textausrichtungs-Attribut der Zeichenfläche, das für die folgenden Aufrufe von *TextOut* und *TextRect* gültig, aber nicht als *Canvas*-Property ansprechbar ist. Sie übergeben *SetTextAlign* das Canvas-Handle (zum Thema Handles siehe Seite 694) und eine Kombination von Ausrichtungskonstanten:

```
Windows.SetTextAlign(Canvas.Handle, TA_CENTER or TA_BOTTOM);
```

Diese Zeile bewirkt, dass der nächste mit *TextOut* gezeichnete Text horizontal an seiner Mitte und vertikal an seiner Unterseite ausgerichtet wird (die Mitte der Unterseite des Textes wird sich also an der Position befinden, die Sie als Koordinaten an *TextOut* übergeben). Andere horizontale Ausrichtungen sind *TA_LEFT* (voreingestellte linksbündige Ausrichtung) und *TA_RIGHT* (rechtsbündig). Weitere vertikale Ausrichtungen sind *TA_TOP* (voreingestellt) und *TA_BASELINE* (richtet den Text an der Grundlinie der Schriftart aus).

Was Windows nicht bietet, ist ein Ausrichtungs-Flag *TA_VCENTER*, um den Text auch vertikal zu zentrieren. Im obigen Listing von *TGraphicElement.PaintText* sehen Sie, dass der *SetTextAlign*-Methode dennoch ein Flag namens *TA_VCENTER* übergeben bekommt. Dies ist eine Erweiterung der Klasse *TVirtualCanvas*.

Textgröße messen

Die Implementierung dieser *TVirtualCanvas*-Erweiterung gibt uns nun ein Beispiel für eine häufig vorkommende Aufgabe: das Messen der Größe eines Textstrings.

Zunächst müssen Sie noch wissen, dass die Methode *TVirtualCanvas.SetTextAlign* die als Parameter übergebenenen Alignment-Flags in der privaten Variablen *FTextAlign* zwischenspeichert. Dort können sie später von der Methode *TVirtualCanvas.TextRect* abgerufen werden. Diese kann die meisten dieser Flags direkt an die Windows-Funktion *SetTextAlign* weitergeben, nur das Flag *TA_VCENTER* muss gesondert behandelt werden. Ist *TA_VCENTER* angegeben, muss *TVirtualCanvas* die Y-Koordinate berechnen, die an die Windows-Textausgabefunktionen übergeben werden muss, damit der Text vertikal zentriert erscheint. Die standardmäßige vertikale Ausrichtung der Windows-Textausgabe ist *TA_TOP* - es wird also die Oberkante des Textes an die gewünschte Position gesetzt. Um die Y-Koordinate dieser Oberkante zu errechnen, muss

- im Standard-Koordinatensystem, in dem die Koordinaten von oben nach unten ansteigen, die Hälfte der Texthöhe von der Mitte des Rechtecks abgezogen werden
- in Koordinatensystemen, in denen die Koordinaten von oben nach unten ansteigen, muss die halbe Texthöhe zur Y-Koordinate der Rechteck-Mitte hinzugezählt werden.

Wie Sie dem folgenden Listing entnehmen können, berücksichtigt *TVirtualCanvas.TextRect* auch das Flag *TA_CENTER*, indem sie die X-Koordinate der Rechteckmitte berechnet und an die Windows-*TextRect*-Funktion weitergibt. Dies vereinfacht den Aufruf von *TVirtualCanvas.TextRect*: Während Sie der gleichnamigen Windows-Funktion im ersten Parameter ein umgebendes Rechteck und dann noch die Position für den Text übergeben müssen, berechnet *TextRect* die Position selbst; es genügt also die Angabe des umgebenden Rechtecks, innerhalb dessen dann auf Wunsch zentriert wird:

```
procedure TVirtualCanvas.TextRect(Rect: TRect; const Text: string);
var
  x, y: Integer;
begin
  // Rect.left und Rect.right sowie top/bottom können im TreeDesigner
  // vertauscht sein. x/y soll in jedem Fall auf die linke obere Kante
  // gesetzt werden:
  x := Rect.left; if Rect.right < x then x := Rect.right;
  y := Rect.top; if Rect.Bottom < y then y := Rect.bottom;
  if (FTextAlign and ta_VCenter) <> 0 then begin
    // ta_VCenter gesetzt, muss manuell berechnet werden,
    // da von Windows nicht unterstützt:
    if MapMode = mmTEXT then
      y := y - (abs(Rect.Top-Rect.Bottom) div 2) - TextHeight('A') div 2
    else
      y := y + (abs(Rect.Top-Rect.Bottom) div 2) + TextHeight('A') div 2;
  end;
  if (FTextAlign and ta_Center) <> 0 then begin
    if MapMode = mmTEXT then
      x := x - (abs(Rect.Right-Rect.Left) div 2)
    else
      x := x + (abs(Rect.Right-Rect.Left) div 2);
  end;
  // Setzen der Alignment-Flags, soweit von Windows unterstützt:
  SetTextAlign(FTextAlign and (not ta_VCenter));
  // In FVCLCanvas speichert TVirtualCanvas das TCanvas-Objekt
  // der VCL, auf dem die Grafikausgabe stattfindet.
  FVCLCanvas.TextRect(Rect, x, y, Text); // TCanvas-Methode aufrufen
end;
```

Um festzustellen, in welcher Richtung die Y-Koordinaten ansteigen, fragt *Paint* die Koordinatensystemeinstellungen ab, auf die wir in Kapitel 5.6.3 zurückkommen werden.

Hinweis: Die vollständige *TVirtualCanvas.TextRect*-Methode enthält Code sowohl für Windows als auch für Linux, der per *$ifdef*-Compileranweisungen unterschieden wird. Das obige Listing zeigt nur die Windows-Teile des Codes. Falls *TVirtualCanvas* unter Linux übersetzt wird, wird die vertikale Ausrichtungsfunktion der Qt-Bibliothek verwendet.

5.5.3 Effiziente Grafikausgabe

Wenn eine Grafikanwendung wie der TreeDesigner bei jeder kleinen Änderung der Grafik, beispielsweise beim Verschieben eines Objekts, das gesamte Fenster mit *Invalidate* für ungültig erklären würde, hätte das zwei Nachteile:

- Solange wenig Grafikobjekte vorhanden sind und das Neuzeichnen sehr schnell geht, würde das ständige Neuzeichnen zu einem Flackern des Bildes führen.
- Falls sehr viele Objekte zu zeichnen sind, würde das komplette Neuzeichnen unter Umständen zu viel Zeit in Anspruch nehmen.

Wie schon erwähnt, sollten Sie der Methode *Invalidate* in einem solchen Fall die Windows-Funktion *InvalidateRect* vorziehen.

Der TreeDesigner verwendet diese Funktion bei jeder geringfügigen Änderung der Grafik. Bevor wir zum konkreten Aufruf von *InvalidateRect* kommen, sehen Sie im Folgenden den allgemeinen Ablauf des Neuzeichnens.

Falls ein Objekt beispielsweise verschoben wird, müssen sowohl der Bereich, der vorher vom Objekt verdeckt war, als auch der gegenwärtig belegte Bereich neu gezeichnet werden. Dazu erklärt *TGraphicElement* zuerst das alte und dann das neue Rechteck für ungültig. Dieser Vorgang sieht in der Methode *SetNewRect* wie folgt aus:

```
procedure TGraphicElement.SetNewRect(NewRect: TRect);
begin
  InvalidateAllViews;  { alten Bereich für ungültig erklären }
  Points := NewRect;   { Koordinaten neu setzen }
  InvalidateAllViews;  { neuen Bereich für ungültig erklären }
  ElementChangeNotify; { Benachrichtigung von COM-Clients (Kap. 8.7.6) }
end;
```

InvalidateAllViews ist eine Methode der Klasse *TGraphicElement* und erklärt den durch das Element abgedeckten Fensterbereich in allen Views (Dokumentfenstern), in denen das Element angezeigt wird, für ungültig. Wichtig ist hierbei, dass das Neuzeichnen erst nach Abschluss der Methode *SetNewRect* stattfindet. Würde es bereits im Aufruf von *InvalidateAllViews* stattfinden, würden ja das Element erneut mit seinen alten Koordinaten gezeichnet werden, da die neuen Koordinaten erst nach *InvalidateAllViews* gesetzt werden.

Hinweis: Das Neuzeichnen findet nicht sofort nach der Beendigung der *SetNewRect*-Methode statt, sondern erst, wenn die Bearbeitung des Ereignisses, das letztendlich den Aufruf von *SetNewRect* ausgelöst hat, abgeschlossen ist. Die VCL wird dann schrittweise die nächsten Nachrichten aus der Nachrichten-Warteschlange abrufen und darunter eine Aufforderung zum Neuzeichnen finden, die sie dann letztlich durch ein *OnPaint*-Ereignis an das Formular weitergibt.

Aufrufen von InvalidateRect

Während Sie den Quelltext der Methode *InvalidateAllViews* in Kapitel 5.3.1 bei der Besprechung des Dokument-View-Konzepts finden, geht es jetzt um die Methode *InvalidateRect*. Wenn Sie kein virtuelles Koordinatensystem verwenden, können Sie die Koordinaten des ungültigen Rechtecks direkt an *InvalidateRect* übergeben. Da *InvalidateRect* aber Gerätekoordinaten erwartet, müssen virtuelle Koordinaten, wie sie im TreeDesigner verwendet werden, vor dem Aufruf mit *LPtoDP* in Gerätekoordinaten umgewandelt werden:

```
procedure TGraphicElement.
              InvalidateRect(var R: TRect; View: TGraphicView);
var
  Control: TWinControl;
  PaintBox: TPaintBox;
  I: Integer;
  VCanvas: TVirtualCanvas;
begin
  PaintBox := View.PaintBox; { siehe nächster Abschnitt }
  if Assigned(View.SetMapModeProc) then { Koordinatensystem einstellen }
    VCanvas := View.SetMapModeProc(PaintBox.Canvas, PaintBox);
  { die PaintBox besitzt kein Window-Handle, daher wird auch
    das Elternfenster der Paintbox benötigt: }
  Control := PaintBox.Parent as TWinControl;
  { virtuelle Koordinaten in Gerätekoordinaten,
    nur bei virtuellem Koordinatensystem notwendig: }
      LPtoDP(PaintBox.Canvas.Handle, R, 2);
  { Rechteck in die richtige Richtung drehen: }
  NormalizeRect(R);
  { Bereich für ungültig erklären: }
  VCanvas.InvalidateRect(Control, R, True);
end;
```

Wichtig ist an dieser Stelle nur der Aufruf der Methode *InvalidateRect*. Hierfür wird zunächst eine *TWinControl*-Komponente benötigt, in der sich der ungültig zu erklärende Bereich befindet. Die Paintbox selbst ist kein *TWinControl*, daher muss die Elternkomponente der Paintbox verwendet werden. Die Paintbox lässt sich aus den *View*-Daten ermitteln (siehe *TGraphicView* in Kapitel 5.3.1). Der zweite Parameter von *SetWindowRect* ist das ungültig zu erklärende Rechteck, welches in Gerätekoordinaten angegeben wird und daher eventuell zuerst berechnet werden muss.

Der dritte Parameter *True* gibt in diesem Fall an, dass das Rechteck von Windows vor dem Neuzeichnen, d.h. vor dem *OnPaint*-Ereignis, gelöscht werden soll. (Der TreeDesigner zeichnet den außerhalb der weißen Zeichenfläche liegenden Bereich nicht selbst neu. Würde als dritter Parameter *False* übergeben werden, blieben auf diesem Bereich eventuell Reste von überstehenden Grafikelementen sichtbar.)

Da Sie die Methode *InvalidateRect* wahrscheinlich eher ohne *TVirtualCanvas* aufrufen werden, sei hier zum Abschluss noch gezeigt, wie der direkte Aufruf der API-Funktion im obigen Beispiel aussehen würde. Die letzte Zeile des Listings könnte dazu etwa durch die Folgende ersetzt werden:

```
Windows.InvalidateRect(Control.Handle, @R, True);
```

Rechteck-Orientierung

Ein letzter Punkt, der im TreeDesigner beachtet werden muss, ist folgender: Beim Einzeichnen der Grafikelemente übernimmt der TreeDesigner die Rechteck-Koordinaten wie der Benutzer sie einzeichnet: Die Position, an der die Maus geklickt wurde, wird zur Ecke (*Left*, *Top*), die Position des Loslassens wird zu (*Right*, *Bottom*). Wenn auf diese Weise ein Rechteck entsteht, bei dem *Left* > *Right* oder *Top* > *Bottom*, dann macht das bei der Ausgabe der Grafik mit *Rectangle* und *Ellipse* nichts aus. Für Linien-Objekte ist es sogar wichtig, dass die Koordinaten auch in diesen Verhältnissen vorliegen können, denn sonst würden alle Linien-Objekte von links oben nach rechts unten gezeichnet[12].

Einige Funktionen wie *InvalidateRect* erwarten jedoch Rechtecke, bei denen sich durch die Berechnung von *Bottom-Top* und *Right-Left* positive Werte ergeben. Andere Rechtecke werden je nach Funktion als nicht vorhanden oder als nach einer Seite unbegrenzt etc. interpretiert. Für diese Funktionen müssen die Rechteck-Koordinaten zuerst in die richtige Reihenfolge gebracht werden, weshalb in der oben gezeigten Methode die Prozedur *NormalizeRect* aufgerufen wird, die wie folgt arbeitet:

```
procedure NormalizeRect(var R: TRect);
  procedure Swap(var x, y: Integer);
  var h: Integer;
  begin
    h := x;  x := y;  y := h;
  end;
begin
  with R do begin
    if right < left then Swap(right,left);
    if bottom < top then Swap(bottom,top);
  end;
end;
```

Unabhängigkeit des Grafikdokuments vom Fenster

In den bisher beschriebenen Funktionen hat *TGraphicElement* die Zeichenfläche des Fensters, auf der es sich ausgeben sollte, immer in einem Parameter namens *CanvasRef* erhal-

12 Das betrifft natürlich nicht die automatisch gezeichneten Verbindungslinien *zwischen* den Grafikobjekten.

ten. Zum Ungültig-Erklären des Rechtecks wird die Zeichenfläche ebenfalls benötigt, da mit ihrer Hilfe die virtuellen in die logischen Koordinaten umgewandelt werden.

Da es sehr umständlich wäre, jeder *TGraphicElement*-Methode, in der das Grafikelement neu gezeichnet werden muss, einen Parameter für die Zeichenfläche zu geben (die Properties *ShapeText* und *Text* könnten so überhaupt nicht mehr geändert werden, da deren Schreibmethoden gar keine zusätzlichen Parameter haben dürfen), schlägt die Unit *GrDoc* einen anderen Weg ein, bei dem die Klassen *TGraphicDoc* und *TGraphicElement* weiterhin unabhängig von jeglichen Formular-Units bleiben.

Und zwar verlangt sie von jedem Fenster, das ein Grafikdokument des Typs *TGraphicDoc* darstellen und verändern will, einen *TGraphicView*-Record, in dem sich unter anderem die oben benötigten Daten *PaintBox* und *SetMapModeProc* befinden. Die Deklaration von *TGraphicView*, die Konstruktion dieses Records und die Weitergabe an das *TGraphicDoc*-Objekt wurden bereits in Kapitel 5.3.1 im Zusammenhang mit dem Dokument-View-Konzept gezeigt.

Das Neuzeichnen in OnPaint

Es stellt sich nun noch die Frage, was passiert, wenn nur ein Teil des Fensters für ungültig erklärt wird. Schließlich haben wir die *PaintboxPaint*-Methode nicht geändert, so dass diese nach wie vor *alle* Objekte zeichnet, selbst wenn nur ein kleiner Bereich der Grafik verändert wurde.

Die Antwort lautet: Zwar gibt *PaintBoxPaint* alle Objekte neu aus; Windows begrenzt diese Neuausgabe jedoch auf den ungültigen Bereich. Mit anderen Worten beschneidet Windows die Grafikausgabe im *OnPaint*-Ereignis nicht nur an den Grenzen des Fensters, sondern an den Grenzen dieses Bereichs. Das hat zur Folge, dass die Grafik außerhalb dieses Bereichs nicht erneuert wird, wodurch das Flackern der Anzeige ganz auf den ungültigen Bereich beschränkt wird. (Sie können das im TreeDesigner ausprobieren, indem Sie eine große Zahl von großen Ellipsen übereinander zeichnen und auf diese ein kleines Rechteck legen. Wenn Sie dieses dann verschieben, erhalten Sie eine anschauliche Demonstration dieses Neuzeichnen-Vorgangs: Selbst bei allerschnellsten Grafikkarten sollte das Neuzeichnen der Ellipsen im verschobenen Bereich des Rechtecks wahrnehmbar sein.)

Vermeidung unnötigen Zeichnens *R90*

Der erste Nachteil von *Invalidate*, das Bildflimmern, ist damit also gelöst. Weitere Optimierungen sind möglich, wenn Sie Objekte, die sich vollständig außerhalb des Clipping-Bereichs befinden, gar nicht erst zu zeichnen versuchen. Dazu müssen Sie in der *OnPaint*-Methode feststellen können, welcher Bereich überhaupt neu gezeichnet wird.

TCanvas stellt Ihnen diesen Bereich als Rechteck im Property *ClipRect* zur Verfügung. Sie könnten mit Hilfe der API-Funktion *IntersectRect* feststellen, ob ein Objekt überhaupt gezeichnet werden muss.

Die bereits in Kapitel 5.5.2 gezeigte *PaintAll*-Methode des TreeDesigner-Dokuments wurde für das Clipping durch die im Folgenden kursiv dargestellten Zeilen und um den Parameter *ClipRect* erweitert:

```
procedure TGraphicDoc.PaintAll(Dest: TVirtualCanvas; ClipRect: TRect);
var
  i: Integer;
  DummyRect, PaintRect: TRect;
begin
  for i:=0 to Count-1 do begin
    Items[i].GetPaintRect(PaintRect, 0);
    NormalizeRect(PaintRect);
    NormalizeRect(ClipRect);
    if IntersectRect(DummyRect, PaintRect, ClipRect) then
      Items[i].Paint(Dest);
  end;
end;
```

Hier bestimmt *IntersectRect*, ob das Rechteck, in dem das Grafikelement gezeichnet wird (*PaintRect*), sich in irgendeiner Weise mit dem neu zu zeichnenden Bereich (*ClipRect*) überschneidet. Das von *IntersectRect* ermittelte Überschneidungsrechteck wird hier nicht benötigt und in der nicht weiter beachteten Variablen *DummyRect* abgelagert.

Entsprechend muss auch der Aufruf von *PaintAll* um den neuen Parameter ergänzt werden. Für *ClipRect* wird einfach das *ClipRect*-Property des *Canvas*-Objekts der Zeichenfläche angegeben:

```
Document.PaintAll(VCanvas, Paintbox.Canvas.ClipRect);
```

Auf diese Weise können Sie einen Teil des Clippings, der sonst automatisch von Windows vorgenommen wird, selbst durchführen. Eigenes Clipping der oben gezeigten einfachen Form lohnt sich normalerweise nur, wenn Sie durch eine einzelne Clipping-Abfrage mehrere Windows-Clippings oder andere zeitaufwändige Aktionen vermeiden (beispielsweise, wenn ein einziger Aufruf von *TGraphicElement.Paint* 20 einzelne Grafikausgabebefehle verursacht, die von Windows einzeln geclippt würden, mit den obigen Anweisungen aber auf einen Schlag vermieden werden könnten).

Während der TreeDesigner 3.0 noch auf dieses eigene Clipping verzichtet hat, weil die einfachen Grafikelemente auch so schnell genug gezeichnet wurden, hat sich dies in der Version 3.5 mit den neuen mehrzeiligen Textelementen zur Darstellung von XML-Dateien (siehe Kapitel 5.9) geändert. Die Ausgabe dieser Textelemente beansprucht nämlich auch dann spürbar Zeit, wenn sie gar nicht im sichtbaren Bereich stattfindet.

Durch die obige Abfrage mit *IntersectRect* werden solche vergeblichen und zeitaufwändigen *Paint*-Bemühungen von vorneherein vermieden.

Hinweis: Wenn Sie Grafik außerhalb der *OnPaint*-Methode ausgeben, ist Ihre Grafikausgabe nicht auf einen Clipping-Bereich beschränkt. Selbst wenn zu diesem Zeitpunkt nur ein bestimmter Bereich ungültig ist, bleibt diese Grafikausgabe im gesamten Fenster bzw. auf der gesamten Zeichenfläche sichtbar.

5.5.4 Scrolling

Scrolling von Grafiken wird dann erforderlich, wenn die Grafik größer ist als die auf dem Bildschirm zur Verfügung stehende Fläche. Zwar ist es meistens möglich, die Grafik so zu verkleinern, dass sie ganz in das Fenster passt, jedoch ist bei einer solchen Übersichtsansicht oft kein Text mehr lesbar.

Dieses Kapitel stellt allgemeine Methoden des Scrollings sowie die im TreeDesigner gewählte Möglichkeit vor. Ein Ziel beim Scrollen ist in jedem Fall, dass die schon geschriebenen Grafikausgabemethoden nicht geändert werden müssen, und zwar nicht aus Bequemlichkeit, sondern um ihre Unabhängigkeit vom Scrolling zu gewährleisten.

Die VCL stellt einen Komponententyp namens *TScrollBox* zur Verfügung, der das Scrollen vereinfacht, wenn Sie nach dem unten beschriebenen Muster vorgehen.

TreeDesigner-Versionen

Dieses Kapitel beschreibt das Verhalten des TreeDesigners ohne die Verwendung des erst in Kapitel 5.6 beschriebenen virtuellen Koordinatensystems. Die fertigen TreeDesigner-Versionen auf der CD arbeiten per Voreinstellung zwar mit eingeschaltetem virtuellen Koordinatensystem, Sie können dieses jedoch mit dem Menüpunkt ANSICHT | GERÄTEKOORDINATENSYSTEM aus- und wieder einschalten. Allerdings dreht sich die Grafik dadurch nahezu komplett um, denn die Y-Koordinaten sind im virtuellen Koordinatensystem so eingestellt, dass sie von unten nach oben ansteigen im Gegensatz zum voreingestellten Fensterkoordinatensystem. Das Einzige, was sich nicht umdreht, ist der zentrierte Beschriftungstext der Grafikobjekte.

Für den Programmcode auf der CD bedeutet das, dass Sie an den Code-Stellen, die beim später eingeführten virtuellen Koordinatensystem anders funktionieren als bei dem in diesem Kapitel vorausgesetzten Gerätekoordinatensystem, eine Fallunterscheidung der folgenden Art finden:

```
if NoMapMode.Checked then
  { Code bei Verwendung des Gerätekoordinatensystems }
else ...
```

Dabei ist *NoMapMode* der Name des Menüpunkts ANSICHT | GERÄTEKOORDINATENSYSTEM. Es kann aber schon verraten werden, dass es nicht viele solcher Stellen gibt; im Prinzip wird in Kapitel 5.6 einfach das virtuelle Koordinatensystem hinzukommen und alles andere funktioniert weiter wie gehabt.

Hinweis: Bei einer gegebenen Zeichenfläche lässt sich auch über die API-Funktion *GetMapMode* feststellen, ob sie gerade im Gerätekoordinatensystem angesteuert wird. *GetMapMode* liefert dann das Ergebnis *MM_TEXT*. Bei Verwendung der Tree-Designer-Klasse *TVirtualCanvas* ist diese Einstellung entsprechend über das Property *TVirtualCanvas.MapMode* in Erfahrung zu bringen.

Funktionsweise des Scrollings

Damit die eigentlichen Grafikdaten beim Verschieben der Bildlaufleisten nicht verändert werden müssen, ist es erforderlich, in zwei verschiedenen Koordinatensystemen zu arbeiten: Im Koordinatensystem der Grafik (virtuelles, logisches oder Weltkoordinatensystem) und in dem des Fensters. Wenn die Grafikausgabefunktionen die Weltkoordinaten automatisch in Fensterkoordinaten umrechnen, können Sie diese Funktionen immer mit den gleichen Originalkoordinaten der Grafik aufrufen, unabhängig davon, welcher Grafikausschnitt gerade durch die Bildlaufleisten eingestellt ist. Verschiebt der Benutzer die Bildlaufleisten, so muss lediglich der Ursprung des Fensterkoordinatensystems relativ zum Weltkoordinatensystem verschoben werden (Abbildung 5.5a).

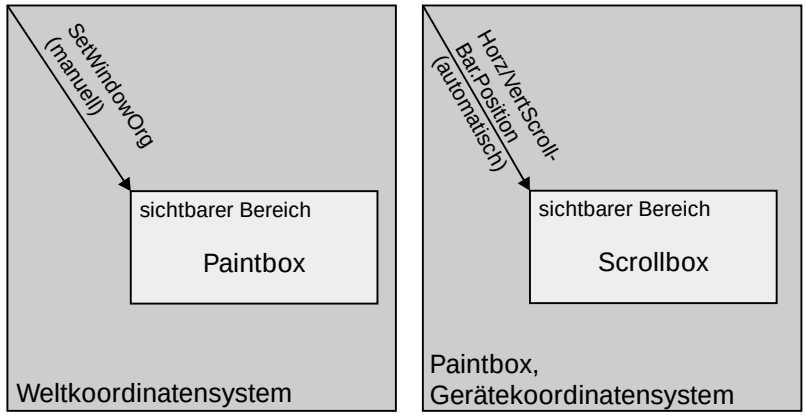


Abbildung 5.5: Der Fensteraufbau für zwei Arten des Scrollens

Praktische Umsetzung

Im Folgenden sind drei verschiedene Realisierungen des Scrollings kurz vorgestellt:

- Die Methode mit den wenigsten Beschränkungen ist die, dass Sie die Weltkoordinaten selbst in Fensterkoordinaten umrechnen. Auf diese Weise sind Sie nicht auf die 16-Bit-Weltkoordinaten von etwaigen 16-Bit-Versionen des Windows-Grafiksystems (wie es auch in Windows 98 noch anzutreffen ist) angewiesen, sondern können theoretisch 32 und mehr Bits verwenden. Die PaintBox, in der die Grafik ausgegeben wird, braucht dann nicht größer als der gesamte zur Verfügung stehende Bereich des Formulars zu sein, allerdings könnte sie dann nicht mehr ohne weiteres mit Hilfe einer ScrollBox gescrollt werden.
- Da das Windows-GDI optional bereits logische Koordinaten berücksichtigt, ist es einfacher, diese Option einzuschalten (mit Hilfe der API-Funktion *SetMapMode*) und daraufhin die Proportionen des logischen Koordinatensystems nach Bedarf einzustellen, wie in Kapitel 5.6.2 beschrieben. Auch bei dieser Vorgehensweise genügt eine Paintbox, die durch *alClient*-Ausrichtung den gesamten Bereich füllt.
- Die von *TScrollBox* begünstigte und damit bequemste Möglichkeit besteht darin, die Paintbox so groß zu machen, dass die gesamte Grafik hineinpasst (also so groß wie das Weltkoordinatensystem). Wenn die Paintbox der Scrollbox untergeordnet ist, aber größer als diese ist, wird sie automatisch am Rand der Scrollbox abgeschnitten, und Sie können mit den Bildlaufleisten der Scrollbox jeden Bereich der Paintbox sichtbar machen. Der grundlegende Unterschied zu den beiden vorher genannten Methoden ist, dass die Weltkoordinaten immer mit den Fensterkoordinaten der Paintbox übereinstimmen (solange keine zusätzlichen Koordinatentransformationen verwendet werden; Abbildung 5.5b).

Da *TScrollBox* die Verschiebung der Paintbox automatisch übernimmt, nehmen wir dieses Angebot dankend an, auch wenn das Weltkoordinatensystem dadurch auf die maximale Größe eines Fensters unter Windows beschränkt wird.

Der visuelle Entwurf *R4*

Um zur Entwurfszeit eine solche scrollbare Paintbox zu erhalten, fügen Sie Komponenten der Typen *TScrollBox* und *TPaintBox* in das Formular ein, wobei letztere Komponente Kindfenster der *ScrollBox* ist. Gehen Sie also so vor wie beim Einfügen von Komponenten in Panels, indem Sie die *PaintBox* direkt in die *ScrollBox* einzeichnen.

Damit die ScrollBox genau den Bereich füllt, der nach Abzug der Werkzeugleiste noch übrig bleibt, und die Größe automatisch mit der des Formulars verändert, ist sie im TreeDesigner mit der Ausrichtung *alClient* versehen.

Die *TPaintBox*-Komponente darf jedoch nicht auf diese Weise ausgerichtet sein, denn ihr Kennzeichen ist es ja gerade, dass sie viel größer ist als die Scrollbox – womit es schwierig wird, sie mit der Maus zu zeichnen. Entweder stellen Sie die Werte *Width* und *Height* also im Objektinspektor ein, oder Sie berechnen Sie zur Laufzeit. Im TreeDesigner werden Breite und Höhe zur Laufzeit bei der Initialisierung des Fensters in der Methode *ResizePaintbox* eingestellt, allerdings wurden die Werte *Left* und *Top* bereits zur Entwurfszeit endgültig auf 0 und 0 festgelegt (diese Position wird an verschiedenen Stellen des TreeDesigners vorausgesetzt, z.B. in *TDocumentForm.SetMapMode* und beim Einstellen der Lineale).

Ereignisse

Welche Ereignisse müssen nun noch bearbeitet werden? Erfreulicherweise keine. *TScrollBox* verschiebt den Zeichenbereich von alleine und die Koordinaten werden in beiden Richtungen automatisch umgewandelt:

- Vom Programm »empfangene« Koordinaten wie die Positionen von Mausklicks in den Methoden für die Mausereignisse werden automatisch relativ zur linken oberen Ecke PaintBox angegeben, die eventuell weit außerhalb des Bildschirms liegt, und
- vom Programm »gesendete« Koordinaten wie die Parameter für die Grafikausgabemethoden beziehen sich ebenfalls auf das PaintBox-System; alle Umrechnungen laufen automatisch ab.

Auch die Neuausgabe der Grafik, wenn sich die Position der Bildlaufleisten ändert, läuft automatisch ab. Dabei wird der Bereich, der sichtbar bleibt, innerhalb des Fensters verschoben und nur der vorher nicht sichtbare Bereich wird neu gezeichnet, so dass es lediglich an den Rändern zu einem Flackern des Bildes kommen kann.

Fließendes Scrolling

Wenn der Benutzer die Bildlaufleisten der Scrollbar mit der Maus verschiebt, kann die Aktualisierung der Grafik auf zwei verschiedene Weisen erfolgen: entweder nur einmal, wenn der Benutzer die Maustaste loslässt, oder fortlaufend, während die Maus bewegt wird. Die letztgenannte Option, das »fließende Scrolling«, erhalten Sie dadurch, dass Sie Properties *HorzScrollBar.Tracking* und *VertScrollBar.Tracking* der ScrollBox einschalten. Im Falle des TreeDesigners ist das aber noch nicht ausreichend, da nicht nur der Inhalt der Zeichenfläche verschoben, sondern auch die Positionsangabe in den Linealen während des Scrollens angepasst werden soll.

Der TreeDesigner verwendet daher die Klasse *TScrollBoxEx* aus Kapitel 6.5.2, die das im nächsten Absatz erläuterte *OnScroll*-Ereignis zur Verfügung stellt. Zum fließenden Scrollen selbst sei noch angemerkt, dass ein angenehmer Ablauf dieses Scrollens natür-

lich voraussetzt, dass die Grafik schnell genug neu aufgebaut werden kann. Für den TreeDesigner gilt hier, dass das Verschieben der Grafik nicht so zeitkritisch ist wie das fließende Zoomen mit dem Scrollbar der Mauspalette (siehe Kapitel 5.6.3), denn beim Verschieben müssen nur die Teile, die vorher unsichtbar waren, neu gezeichnet werden, während bei einer Änderung des Zoomfaktors die gesamte Zeichenfläche aktualisiert werden muss.

Lineale

Die Klasse *TScrollBoxEx* erzeugt bei jedem Scrollvorgang ein *OnScroll*-Ereignis, das der TreeDesigner dazu nutzen kann, die beiden Lineal-Komponenten anzupassen. Zwar wäre es auch möglich, ohne dieses Ereignis die Lineale auszugeben, und zwar immer dann, wenn auch das Ereignis *OnPaint* auftritt. Dies würde jedoch zu einer Reihe unnötiger Wiederholungen des Lineal-Zeichnens führen (unter anderem jedes Mal dann, wenn die Grafik verändert wird).

Die folgende Version der Methode gilt nur, solange noch keine Veränderung des Koordinatensystems durchgeführt wird (die vollständige Version finden Sie auf der CD). Sie gibt auch ein Beispiel dafür, wie die im Komponenten-Package des Buchs enthaltene *TRuler*-Komponente verwendet wird:

```
procedure TDocumentForm.ScrollBoxScroll(Sender: TObject; Code: Word;
  Horizontal: Boolean);
begin { Einstellen der durch die Lineale gezeichneten Grenzen }
  HRuler.LeftVal := ScrollBox.HorzScrollBar.Position;
  HRuler.RightVal := ScrollBox.HorzScrollBar.Position
                     +ScrollBox.ClientWidth;
  VRuler.LeftVal := ScrollBox.VertScrollBar.Position;
  VRuler.RightVal := ScrollBox.VertScrollBar.Position
                      +ScrollBox.ClientHeight;
end;
```

5.5.5 Clipping

Ein wichtiger Funktionsbereich des Windows-GDI, der nicht von der VCL abgedeckt wird, ist das *Clipping*. Mit dem Clipping können Sie die Grafikausgabe, die bereits standardmäßig auf den Bereich des Fensters beschnitten wird, weiter einschränken (Abbildung 5.6). Wenn Sie beispielsweise einen Kreis als Clipping-Bereich festlegen und die gesamte Zeichenfläche mit dem Aufruf der Methode *Rectangle* zu füllen versuchen, wird statt dessen nur der Kreis ausgefüllt, den Sie als Clipping-Bereich angegeben haben.

Abbildung 5.6: Text-Clipping im TreeDesigner-Dokument

Überblick

Windows erlaubt noch komplexere Clipping-Bereiche als Rechtecke und Kreise bzw. Ellipsen: Sie können verschiedene Kreise, Rechtecke, abgerundete Rechtecke und Polygone beliebig kombinieren, wobei Sie die einzelnen Teilbereiche schneiden, vereinigen und voneinander abziehen können.

Dieses Kapitel kann nur eine Einführung in das Clipping anhand des TreeDesigners sein. Mehr über die Möglichkeiten des Clippings erfahren Sie in der Online-Hilfe. Im Falle der Win32-Hilfe von Delphi finden Sie im Inhaltsverzeichnis beispielsweise die Kapitel *Clipping* und *Regions*.

Hinweis: Bei Einsatz der Clipping-Funktionen besteht keine Gefahr, dass Sie den vorgegebenen Clipping-Bereich ausschalten, der die Ausgabe auf das Fenster beschränkt, denn dieser wird von Windows immer zusätzlich berücksichtigt. Sie können jedoch außerhalb des Fensters zeichnen, wenn Sie sich mit API-Funktionen ein Gerätekontext-Handle auf den gesamten Bildschirm beschaffen (siehe die API-Funktionen *GetWindowDC* und *GetDesktopWindow*). Dies funktioniert sogar unter Windows NT, allerdings erreichen Sie dort wirklich nur den Desktop-Hintergrund und können auf diese Weise keine anderen Fenster übermalen, wie das unter Windows 98 noch möglich ist (siehe Programm *DesktopPaint* auf der CD).

Für das Clipping müssen wir nun Funktionen des Windows-API aufrufen. Zwar besitzt *TCanvas* ein Property namens *ClipRect*; dieses ist jedoch nicht beschreibbar und gibt auch nur das vorgegebene Clipping-Rechteck an, das die Ausgabe auf das Fenster oder auf den ungültigen Bereich beschränkt (siehe Kapitel 5.5.3). Für jedes zusätzliche Clipping müssen Sie zuerst einen Bereich bzw. eine *Region* definieren.

Die Region ist ein Windows-Objekt, das Sie von den oben erwähnten Clipping-Funktionen erhalten. Eine Ellipse ist beispielsweise das Ergebnis des folgenden Aufrufs:

```
var
  Rgn: THandle;
begin
  Rgn := Windows.CreateEllipticRgnIndirect(R);
```

Nachdem Sie die Region mit der Funktion *SelectClipRgn* als Clipping-Bereich aktiviert haben (dazu später mehr), sollten Sie den von der Region belegten Speicher wieder freigeben – mit der API-Funktion *DeleteObject*:

```
Windows.DeleteObject(Rgn);
```

API-Funktionen und Handles

Windows ist hierbei die Angabe der Unit, in der die Windows-API-Funktionen deklariert sind. Sie können die Funktionen auch ganz normal aufrufen, wenn Sie sicher sind, dass die VCL keine Methode oder Funktion definiert, die genauso heißt:

```
{ wenn kein Namenskonflikt besteht:}
CreateEllipticRgnIndirect(R);
```

Alle Windows-API-Funktionen zur Ausgabe von Grafik und auch die Funktion *SelectClipRgn*, die wir jetzt benötigen, erwarten im ersten Parameter ein *Handle auf einen Gerätekontext*. Unter Windows gibt es verschiedene Typen solcher Handles, bei denen es sich entweder um 16 oder um 32 Bit große Werte handelt, die nach außen keinen besonderen Sinn machen. Für Windows handelt es sich jedoch um eine Kennzahl bzw. um die Adresse einer internen Struktur, in diesem Fall des Gerätekontextes. Aus der Sicht der objektorientierten Programmierung ist ein Handle auch mit einem *self*-Zeiger vergleichbar.

Andere Objekte, auf die Handles verteilt werden, sind z.B. Fenster, Speicherblöcke oder Ausgabegeräte für Multimedia. In all diesen Fällen stellt die VCL jedoch Klassen zur Verfügung, die Ihnen meistens den Zugriff auf Handles und API-Funktionen ersparen.

Wenn eine Anwendung eine solche interne Windows-Struktur in Anspruch nehmen will, ruft sie eine spezielle API-Funktion auf (die oft mit *Create* oder mit *Get* beginnt) und erhält von dieser ein Handle zurück. Mit diesem bezieht sich die Anwendung solange auf das Windows-Objekt, bis dieses wieder gelöscht werden kann. Für das Löschen ist jeweils eine weitere API-Funktion zuständig. Nach der Löschung verliert das Handle seine Gültigkeit.

Canvas.Handle

Normalerweise ist es viel schwieriger, ein Handle von Windows zu erhalten, als dieses später zu verwenden, denn die Initialisierungsfunktionen zeichnen sich meistens durch eine lange Parameterliste und komplizierte Initialisierungsstrukturen aus. Glücklicherweise können Sie sich die Handles bei Delphi von den VCL-Objekten ausleihen. So wie jede *TWinControl*-Komponente im Property *Handle* ein Fenster-Handle bereitstellt, enthält jedes *TCanvas*-Objekt in seinem *Handle*-Property ein Gerätekontext-Handle. In der Natur des Properties liegt es, dass es immer gültig ist, wenn es abgefragt wird. Falls es vor der Abfrage des Properties noch kein gültiges Handle gab, lässt sich *TCanvas* von Windows schnell ein neues Handle geben.

Die automatische Handle-Erstellung der VCL-Klassen hat jedoch auch einen Nachteil: Sie erfahren nicht, wann ein Handle ungültig wird und damit auch alle von Ihnen vorgenommenen Einstellungen. Haben Sie beispielsweise in der *OnPaint*-Bearbeitung mit *SetTextAlign* die Ausrichtung des Textes verändert, so gilt diese Einstellung nur für das gerade gültige Gerätekontext-Handle. Beim nächsten *OnPaint*-Ereignis liefert die VCL einen neuen Gerätekontext, in dem wieder die Standardeinstellungen, beispielsweise der standardmäßige Clipping-Bereich des Fensters, die Textausrichtung *TA_LEFT or TA_TOP* und das voreingestellte Gerätekoordinatensystem gelten. Anders verhält es sich mit den Properties der VCL-Objekte: *TCanvas* sorgt natürlich dafür, dass die Einstellungen von *Pen*, *Brush* und *Font* wiederhergestellt werden, wenn ein neues Handle erzeugt wird.

Als Faustregel für die Gültigkeit eines Gerätekontext-Handles können Sie annehmen, dass der Gerätekontext erst am Ende der Nachrichtenbearbeitung (in *TWinControl.MainWndProc*, siehe Kapitel 3.1.4) aufgelöst wird bzw. dass er innerhalb der Methode zu *OnPaint* nicht gelöscht wird.

Aktivieren und Deaktivieren des Clipping-Bereichs *R99*

Wie schon erwähnt, können Sie nun mit dem *Canvas*-Handle und der Funktion *SelectClipRgn* die eben erhaltene Region *Rgn* als Clipping-Bereich aktivieren. Der Gesamtablauf eines einfachen Clipping-Vorgangs sieht also wie folgt aus:

```
R := Rect(100, 100, 200, 200);
{ Clippen am Rechteck R: }
Rgn := Windows.CreateEllipticRgnIndirect(R);
Windows.SelectClipRgn(Canvas.Handle, Rgn);
Windows.DeleteObject(Rgn);
```

SelectClipRgn erstellt eine Kopie der Region, so dass Sie die Original-Region, wie oben gezeigt, mit *DeleteObject* wieder löschen können.

Um das Clipping abzuschalten, rufen Sie *SelectClipRgn* erneut auf, übergeben aber eine 0 als zweiten Parameter:

```
Windows.SelectClipRgn(Canvas.Handle, 0);
```

Im TreeDesigner beispielsweise wird das Clipping nach Ausgabe der Beschriftung gleich wieder abgeschaltet, weil nur der Beschriftungstext abgeschnitten werden soll. Im Listing von *TGraphicElement*.Paint (Kapitel 5.5.2) wurde bereits der entsprechende Aufruf von *TVirtualCanvas.ClippingOff* gezeigt (diese ruft dann *SelectClipRgn* wie beschrieben mit einem zweiten Parameter von Null auf).

Gerätekoordinaten

Anders als die bisher besprochenen Grafikfunktionen arbeiten die Clipping-Funktionen mit Gerätekoordinaten. Die obigen Listings setzen solche Gerätekoordinaten voraus. Logische Koordinaten, wie sie im TreeDesigner verwendet werden, müssen jedoch vor der Übergabe an die *Create...RgnIndirect*-Funktion in Gerätekoordinaten umgewandelt werden (siehe Kapitel 5.6.3). Falls Sie keine der in Kapitel 5.6.2 besprochenen Funktionen verwenden, um das Koordinatensystem zu verändern, arbeiten Sie nur mit Gerätekoordinaten und benötigen *LPtoDP* nicht.

Der TreeDesigner-Code

Im TreeDesigner wurden die Clipping-Funktionen in die Klasse *TVirtualCanvas* ausgelagert, die wieder entsprechende Code-Alternativen für die Kompilierung mit Kylix enthält (es werden dann die Clipping-Funktionen der Qt-Bibliothek verwendet). *TVirtualCanvas* definiert die Methoden *SetClippingEllipse* und *SetClippingRectangle*, die beim Zeichnen der TreeDesigner-Grafikelemente wie folgt aufgerufen werden (die Code-Alternativen für Kylix sind im Folgenden nicht abgedruckt):

```
procedure TGraphicElement.SetTextClipRegion(CanvasRef : TVirtualCanvas);
var
  R: TRect;
begin
  GetInnerRect(R);
  NormalizeRect(R);
  if (ShapeType = stEllipse) then
    CanvasRef.SetClippingEllipse(R)
  else
    CanvasRef.SetClippingRectangle(R);
end;
```

Die obige Methode verwendet Clipping nur dazu, den Beschriftungstext für Ellipsen-Objekte am Rand der Ellipse abzuschneiden. Dies ist für den Fall gedacht, dass einmal eine sehr große Schriftart gewählt wird. Für alle anderen grafischen Formen (auch die nicht-rechteckigen) führt der TreeDesigner kein spezielles Clipping durch, sondern

stellt der Einfachheit halber lediglich ein Rechteck als Clipping-Region ein. Es ist jedoch wichtig, dass zumindest an diesem Rechteck geclippt wird, denn für das Neuzeichnen von Grafikelementen wird ja auch nur dieses Rechteck für ungültig erklärt. Ein völliger Verzicht auf das Clipping würde dazu führen, dass die Beschriftung des Grafikelements über den Rand des Rechtecks hinausragen könnte. Würde dann das Grafikelement verschoben, würde nur der Bereich des Rechtecks für ungültig erkärt und neu gezeichnet. Die überstehenden Textteile würden dann als Reste auf dem Bildschirm verbleiben.

Hinweis: Geht es nur um das einfache Clipping von *Text* an einem *Rechteck*, können Sie auf die in diesem Kapitel besprochenen Funktionen verzichten, denn die Funktion *TCanvas.TextRect* führt derartiges Clipping bereits automatisch durch.

5.6 Skalierung, virtuelle Koordinatensysteme und Druckerausgabe

Virtuelle Koordinatensysteme erleichtern die Ausgabe von geräteunabhängiger Grafik erheblich und erlauben eine Vergrößerung der Grafik auf eine einfache Weise, die grundsätzlich so funktioniert: Sie haben auf der einen Seite ein virtuelles *Weltkoordinatensystem*, das mehrere tausend Einheiten durchmisst und in dem sich Ihre Grafik befindet, auf der anderen Seite haben Sie ein Fenster des Bildschirms. Nun sagen Sie Windows, dass es im beispielsweise 1000*500 Pixel großen Fensterbereich einen 100*50 virtuelle Einheiten großen Ausschnitt des Weltkoordinatensystems darstellen soll. In diesem Beispiel erhalten Sie eine zehnfache Vergrößerung, denn jede virtuelle Einheit ist auf dem Bildschirm 10 Pixel groß.

Kapitel 5.6.3 zeigt den praktischen Einsatz eines solchen Koordinatensystems am Beispiel des TreeDesigners. Einen noch größeren Nutzen ergeben die virtuellen Koordinatensysteme jedoch auf einem anderen Gebiet: der Geräteunabhängigkeit. Wie Sie diese für die Druckerausgabe nutzen können, erfahren Sie in Kapitel 5.6.4. Das Kapitel 5.6.1 beginnt zunächst mit den Grundlagen. Während zum Scrollen lediglich Methoden der VCL notwendig sind, müssen wir zum Vergrößern der Grafik mit Hilfe von virtuellen Koordinatensystemen wieder ein paar Funktionen des Windows-API zu Hilfe nehmen, die in Kapitel 5.6.2 beschrieben sind.

5.6.1 Geräteunabhängigkeit

Die im obigen Beispiel angegebene Definition des Koordinatensystems (ein 100*50 Einheiten großer Bereich wird auf 1000*500 Pixel gezoomt) ist nur eine von vielen Möglichkeiten, die Windows bietet: Alternativ dazu können Sie einer virtuellen Einheit

eine fest definierte Größe zuordnen. Im Abbildungsmodus *mm_LoMetric* beispielsweise entspricht eine virtuelle Einheit einem Zehntelmillimeter. Die weiteren möglichen Modi sind in der folgenden Tabelle zusammengefasst:

Abbildungsmodus-Konstante	Größe einer Einheit
MM_TEXT	eine Geräteeinheit
MM_LOMETRIC	0,1 Millimeter
MM_HIMETRIC	0,01 Millimeter
MM_LOENGLISH	0.1 Zoll
MM_HIENGLISH	0.01 Zoll
MM_TWIPS	1/20 Punkt bzw. 1/1440 Zoll
MM_ISOTROPIC	frei definierbar, aber quadratische »Pixel«
MM_ANISOTROPIC	völlig frei definierbar

MM_TEXT steht für das voreingestellte Gerätekoordinatensystem. Im Zusammenhang mit den fünf anderen Modi passt auch die Bezeichnung als *logische* Koordinatensysteme, denn die darin definierten Abstände sind nicht mehr »virtuell«, sondern real. Wenn Sie in diesen Koordinatensystemen arbeiten, sollten Sie auf allen Ausgabegeräten gleich große Grafiken erhalten. Ein hochauflösender Drucker verwendet dann eben erheblich mehr Pixel dazu, um eine 1 cm lange Linie zu drucken, als ein Bildschirm Pixel dafür zur Verfügung stellen muss.

Während Sie beim Drucken eine sehr exakte Umsetzung Ihrer Größenangaben erwarten können, ist das bei Bildschirmen natürlich nicht so einfach: Drucker verfügen über genau definierte Auflösungen wie z.B. 600 oder 1440 dpi und darüber hinaus über spezifische Treiber, während die Auflösung der Bildschirme sehr variabel ist und sogar noch durch Verstellung des Bildes am Monitor geändert werden kann, ohne dass Windows etwas davon merkt. Noch nicht einmal die Größenklasse des Monitors ist Windows bekannt, solange der Monitortyp nicht in der Systemkonfiguration angegeben, ein Treiber für den Monitor installiert oder die Plug-and-Play-Funktionen moderner Monitore eingesetzt wird. Von welcher Bildschirmauflösung pro Zoll Windows ausgeht, können Sie in einer Delphi-Anwendung vom Property *Screen.PixelsPerInch* (siehe *TScreen*, Kapitel 3.2.4) erfahren.

Achsenrichtungen

Die Achsenrichtungen der Koordinatensysteme weisen leichte Unterschiede auf: Während die Y-Koordinaten im Modus *MM_TEXT* nach unten hin ansteigen, steigen sie in den Koordinatensystemen, die feste Maßeinheiten verwenden, von unten nach oben an. In jedem Fall befindet sich der voreingestellte Ursprung des Koordinatensystems in

der linken oberen Ecke, jedoch können Sie diesen in jedem Abbildungsmodus auch an eine andere Position verschieben.

Die einzigen Modi, in denen Sie die Richtungen der Koordinatenachsen umkehren können, sind *MM_ANISOTROPIC* und *MM_ISOTROPIC*. Sie benötigen dazu die Funktionen *SetWindowExtEx* und *SetViewportExtEx*, die im nächsten Kapitel beschrieben werden.

5.6.2 Die Funktionen des Windows-API

Die Einstellungen des logischen Koordinatensystems gehören zu den Einstellungen der Gerätekontexte, die nicht von der Klasse *TCanvas* berücksichtigt werden. Auch hier benötigen wir also wieder Funktionen des Windows-API sowie das *Handle*-Property der Klasse *TCanvas*. Mehr Informationen zu diesem Handle finden Sie auf Seite 694.

SetMapMode

Das Erste, was Sie zur Definition der logischen bzw. virtuellen Koordinatensysteme machen müssen, ist, einen der Abbildungsmodi aus der zuletzt gezeigten Tabelle einzustellen. Hierzu verwenden Sie die Funktion *SetMapMode*. Diese erwartet als ersten Parameter das Gerätekontext-Handle und im zweiten Parameter die den Modus beschreibende Konstante aus der Tabelle, beispielsweise:

```
SetMapMode(Canvas.Handle, MM_ISOTROPIC);
```

»Viewport« und »Window«

Die Modi *MM_ANISOTROPIC* und *MM_ISOTROPIC* werden unter Windows durch zwei Rechtecke näher beschrieben, deren Bezeichnung in den API-Funktionen *Window* und *Viewport* ist. Dabei hat »Window« nichts mit den Fenstern der Benutzerschnittstelle zu tun, sondern meint einen Ausschnitt aus dem Weltkoordinatensystem. Der »Viewport« befindet sich hingegen auf der physikalischen Abbildungsfläche (meistens das Bildschirmfenster oder der Drucker), ist aber nicht mit dieser gleichzusetzen.

Die Beziehungen dieser beiden Systeme zum Koordinatensystem des Ausgabegeräts zeigt Abbildung 5.7. In Abbildung 5.8 sehen Sie ein Testprogramm, das Sie unter dem Namen *MapModes* auf der CD-ROM finden. Es ermöglicht Ihnen eine interaktive Einstellung der Abbildungsmodi und deren Parameter und eignet sich somit gut zum Experimentieren mit den im Folgenden besprochenen Parametern. (Das Programm verknüpft alle *OnChange*-Events der Eingabeelemente mit einer Ungültig-Erklärung des gesamten Fensters durch *Invalidate*. Bei jedem *OnPaint*-Ereignis stellt es den Abbildungsmodus gemäß den Eingaben neu ein und gibt das Testbild neu aus.)

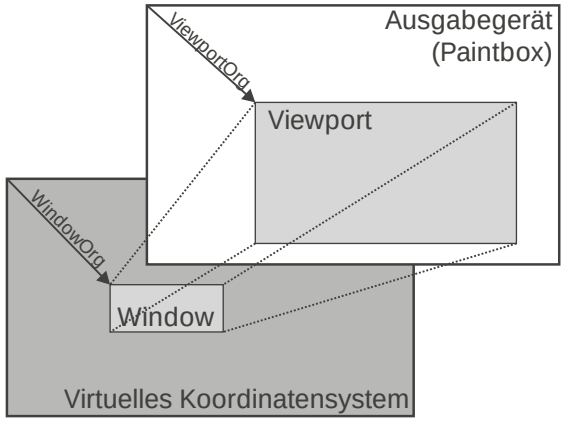


Abbildung 5.7: Window, Viewport und Gerätekoordinatensystem

Viewport und *Window* werden dadurch eingestellt, dass Sie den Ursprung mit *SetWindowOrgEx* / *SetViewportOrgEx* positionieren und die Größenverhältnisse mit *SetWindowExtEx* / *SetViewportExtEx* einstellen. Bei den Abbildungsmodi, die bereits auf festen Maßeinheiten basieren (inklusive *MM_TEXT*), lassen sich die Größenverhältnisse nicht verändern; sie können hier nur die beiden *Org*-Funktionen verwenden.

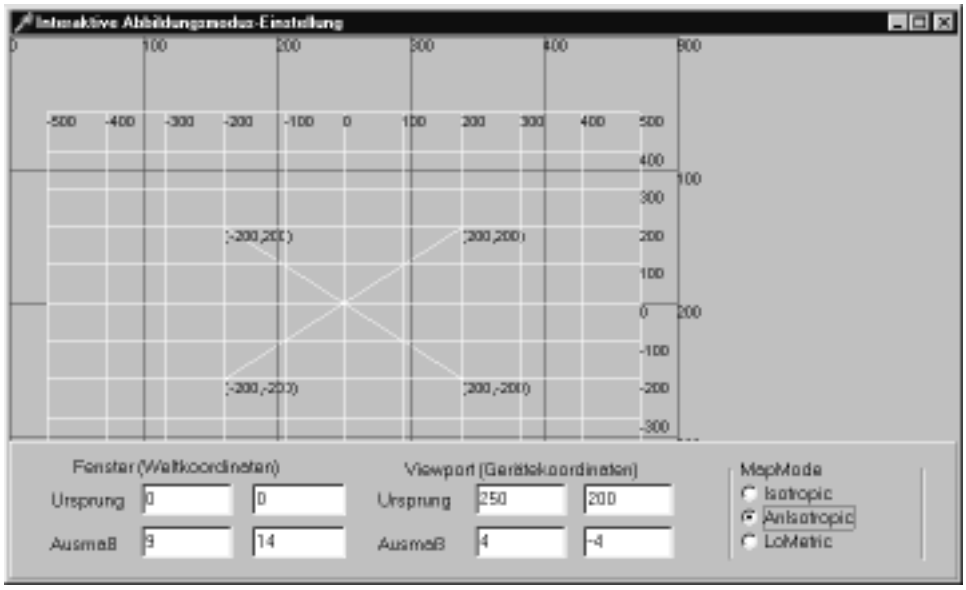


Abbildung 5.8: Ein selbst definiertes Koordinatensystem mit umgedrehter Y-Achse

TVirtualCanvas – Erweiterung für TCanvas

Da der TreeDesigner einige Grafikfunktionen benutzt, die nicht von *TCanvas* bereitgestellt werden, verwenden die meisten Zeichenmethoden des TreeDesigners eine andere Klasse: *TVirtualCanvas*, die bereits in Kapitel 5.5.2 kurz eingeführt wurde und die auf der CD in der Unit *VCanvas* zu finden ist. *TVirtualCanvas* hat drei wichtige Aufgaben:

- Alle von *TCanvas* angebotenen öffentlichen Methoden und Properties zur Verfügung zu stellen, damit ein *TVirtualCanvas*-Objekt anstelle eines *TCanvas*-Objekts verwendet werden kann.
- Methoden und Properties für alle zusätzlichen vom TreeDesigner benötigten Grafikfunktionen zu definieren (virtuelle Koordinatensysteme, Clipping, horizontale und vertikale Ausrichtungsoptionen für Textausgabe und Ungültigerklärung von Rechtecken).
- Alle zusätzlichen Methoden und Properties sowohl unter Windows als auch unter Linux zu implementieren. Die Unit *VCanvas* verwendet dafür zahlreiche Blöcke, die bedingt mit *$ifdef Windows* und *$ifdef Linux* kompiliert werden. Der Code, der *TVirtualCanvas* benutzt, sollte sowohl unter Linux mit der CLX als auch unter Windows mit der VCL (ohne Qt!) die erweiterten Grafikfunktionen wie Clipping, virtuelle Koordinatensysteme etc. verwenden können.

Verwendung von TVirtualCanvas *R86*

Da die VCL grundsätzlich nur ihre eigenen *TCanvas*-Objekte erzeugt, müssen wir das *TVirtualCanvas*-Objekt selbst erzeugen und es mit dem vorgegebenen *TCanvas*-Objekt verbinden, damit es mit diesem zusammenarbeitet[13]. Für die Verbindung mit dem *Canvas* muss das *VCLCanvas*-Property gesetzt werden (bei der Verwendung der CLX ist damit eigentlich *CLXCanvas* gemeint):

```
VCanvas := TVirtualCanvas.Create;     // Erzeugung des Objekts
VCanvas.VCLCanvas := Paintbox.Canvas;  // Verbindung mit Canvas-Objekt
VCanvas.Rectangle(...)                // Aufruf von TCanvas-Methoden
VCanvas.SetClippingEllipse(...)        // Aufruf von neuen Methoden
```

Wenn das *VCanvas*-Objekt nicht mehr benötigt wird, können Sie es mit *VCanvas.Free* freigegeben (optimal wäre es natürlich, alles, was dem *Create*-Aufruf folgt, in einen *try...finally*-Block einzuschließen und *Free* im *finally*-Teil aufzurufen). Die Freigabe des *TVirtualCanvas*-Objekts hat im Übrigen keinen Einfluss auf das damit verbundene *TCanvas*-Objekt der VCL).

13 Dies ist auch der Grund dafür, dass *TVirtualCanvas* nicht von *TCanvas* abgeleitet ist: Die Ableitung wäre nur sinnvoll, wenn *TVirtualCanvas* das bestehende *TCanvas*-Objekt *ersetzen* könnte. Da die CLX in jedem Fall ihr eigenes *TCanvas*-Objekt erzeugt, muss *TVirtualCanvas* mit diesem kooperieren und braucht selber nicht von *TCanvas* abgeleitet zu sein.

Das Dokumentformular des TreeDesigners geht auf eine besondere Weise mit *TVirtualCanvas* um: Da hier praktisch ununterbrochen (bei jeder Mausbewegung und jeder Zeichenoperation) ein Objekt dieser Klasse benötigt wird, wird eine *TVirtualCanvas*-Variable in der Formularklasse gespeichert, auf die über ein Property zugegriffen werden kann:

```
TDocumentForm = class(TForm)
private
  FPaintboxVCanvas: TVirtualCanvas;
  function GetPaintboxVCanvas: TVirtualCanvas;
public
  property PaintBoxVCanvas: TVirtualCanvas read GetPaintboxVCanvas;
```

Die Property-Lese-Methode erzeugt das Objekt, falls noch keines existiert:

```
function TDocumentForm.GetPaintboxVCanvas: TVirtualCanvas;
begin
  if FPaintBoxVCanvas = nil then
    FPaintboxVCanvas := TVirtualCanvas.Create;
  Result := FPaintboxVCanvas;
end;
```

Da das Objekt erst beim Schließen des Formulars freigegeben wird, findet diese Erzeugung für jedes Dokumentfenster nur einmal statt.

Hinweis: Ein solches Property, bei dessen erstem Lesezugriff ein dahinter liegendes Objekt erzeugt wird, eignet sich hervorragend zur Initialisierung von Objekten, für die nicht bekannt ist, wann genau sie zum ersten Mal benötigt werden. Muss *PaintBoxVCanvas* schon vor der *OnCreate*-Methode initialisiert sein oder erst danach? Wann treten die ersten *OnMouseMove* und *OnPaint*-Ereignisse auf, die das *TVirtualCanvas*-Objekt benötigen? Die Property-Lesemethode stellt sicher, dass das Objekt keine Sekunde zu früh und keine Sekunde zu spät erzeugt wird (was natürlich eher ein ideeler Gewinn ist, denn das Objekt verbraucht nur wenige Bytes an Speicher).

In Kapitel 5.5.2 wurde bereits gezeigt, wie die *OnPaint*-Methode ein *TVirtualCanvas*-Objekt konstruiert. Dies war jedoch eine Vereinfachung, die nicht der Realität im TreeDesigner entspricht. Der tatsächliche Code beginnt wie folgt:

```
procedure TDocumentForm.PaintBoxPaint(Sender: TObject);
var
  VCanvas: TVirtualCanvas;
  ClipRectLP: TRect;
begin
  VCanvas := SetMapMode(PaintBox.Canvas, PaintBox);
  Document.PaintBackground(VCanvas); // (weiter ohne Änderung)
  ...
```

Letztlich entscheidet also die Methode *SetMapMode*, die das Koordinatensystem einstellt, welches *TVirtualCanvas*-Objekt erzeugt wird. Sie verwendet lediglich das erläuterte *PaintBoxVCanvas*-Property:

```
function TDocumentForm.SetMapMode(DestCanvas: TCanvas;
  DestControl: TControl): TVirtualCanvas;
var
 R: TRect;
begin
  Result := PaintboxVCanvas;
  Result.VCLCanvas := DestCanvas;
```

Im Rest dieses Kapitels geht es nun um die eigentliche Aufgabe von *SetMapMode* – der Einstellung des virtuellen Koordinatensystems.

Die Koordinatensystem-Schnittstelle von TVirtualCanvas *R98*

Die Methodennamen von *TVirtualCanvas* sind an den Koordinatensystem-Funktionen des Windows-API orientiert, denn *TVirtualCanvas* sollte ja die Portierung des TreeDesigners nach Linux vereinfachen.

Im Folgenden sind die in diesem Kapitel benötigten *TVirtualCanvas*-Methoden aufgeführt:

- *SetViewportOrg* setzt den Ursprung des Viewport-Rechtecks,
- *SetViewportExt* setzt dessen Ausdehnung.
- *SetWindowOrg* und *SetWindowExt* definieren auf entsprechende Weise das Window-Rechteck.
- *SetMapMode* setzt einen von vier Abbildungsmodi, die den ähnlich benannten Mapping Modes des Windows-API entsprechen (siehe Kapitel 5.6.1).

Die folgenden Abschnitte geben zunächst einige allgemeine Anwendungsbeispiele für die Koordinatensystem-Methoden von *TVirtualCanvas*.

Der Ursprung

Mit den Methoden *SetViewportOrg* und *SetWindowOrg*, die Sie bei allen Abbildungsmodi verwenden können, verschieben Sie das virtuelle Koordinatensystem relativ zur Ausgabefläche. Per Voreinstellung befindet sich die logische Koordinate (0, 0) immer in der linken oberen Ecke der Ausgabefläche des Fensters oder Druckers, unabhängig davon, ob die Y-Koordinaten nach oben oder unten hin ansteigen.

- Das Haupteinsatzgebiet von *SetViewportOrg* ist eine feste Einstellung der Ausgabefläche, die sich zur Programmlaufzeit normalerweise nicht ändert: *SetViewportOrg* setzt den Ursprung des Viewports, an dem sich per Voreinstellung die logische

Koordinate (0, 0) befindet, an eine andere Position des Ausgabegeräts. Häufig soll beispielsweise nicht die linke obere, sondern die rechte untere Ecke der Ausgabefläche den Ursprung beinhalten. Angenommen, die Größe der Fensterfläche befindet sich im Property *TPaintBox.ClientHeight*. Um den Ursprung auf dem Bildschirm in die linke untere Ecke zu verschieben, müssten Sie *SetViewPortOrg* wie folgt aufrufen:

```
SetViewportOrg(0, PaintBox.ClientHeight);
```

- Während *SetViewportOrg* in Geräteeinheiten arbeitet, verwendet *SetWindowOrg* logische Einheiten. Mit dieser Funktion können Sie eine andere logische Koordinate angeben, die statt (0,0) am Ursprung des Viewports erscheinen soll. Hat der Benutzer beispielsweise die beiden Bildlaufleisten eines Fensters auf die Position (450, 560) eingestellt, so können Sie diesen Punkt mit

  ```
  SetWindowOrg(450, 560, nil)
  ```

  in die linke obere Ecke des Fensters bzw. dahin positionieren, wo Sie den Viewport mit *SetViewportOrgEx* gebracht haben. (Um Verwechslungen zu vermeiden: Der TreeDesigner geht beim Scrollen nicht auf diese Weise vor.)

Die Größenverhältnisse

Damit wir *Window* und *Viewport* auch als Rechtecke verstehen können, benötigen wir bildlich gesehen neben den im letzten Abschnitt besprochenen Ursprüngen noch die Seitenlängen. Diese lassen sich nur in den Abbildungsmodi *mmIsotropic* und *mmAnisotropic* sowie mit den Funktionen *SetViewportExt* und *SetWindowExt* verändern.

Wenn Grafik vergrößert oder verkleinert wird, so wird ein bestimmter Abschnitt aus dem Weltkoordinatensystem auf die Ausgabefläche projiziert (Abbildung 5.7). Vergrößern Sie diesen Ausschnitt, so verkleinert sich automatisch die Grafik, falls die Größe der Ausgabefläche gleich geblieben ist. Umgekehrt erreichen Sie eine Vergrößerung, indem Sie einen kleineren Ausschnitt aus dem Weltkoordinatensystem wählen.

Die folgenden Anweisungen dehnen einen Bereich von 100 mal 100 virtuellen Einheiten auf den gesamten Arbeitsbereich eines Fensters aus. Wenn Sie ein Fenster auf diese Weise mit einer Grafik füllen, können Sie diese zur Laufzeit vergrößern und verkleinern, indem Sie einfach die Größe des Fensters anpassen:

```
with VCanvas do begin
  MapMode := mmAnisotropic;
  SetWindowExt(100, 100);
  SetViewportExt(ClientHeight, ClientWidth);
  ... Grafikausgabe ...
end;
```

Hinter der anschaulichen Bezeichnung als »WindowExt/ViewportExt«, also den Ausmaßen von Window und Viewport, versteckt sich jedoch ein abstrakteres Konzept: Das Grafiksystem von Windows interessiert sich gar nicht für die absoluten Werte, die Sie diesen beiden Funktionen übergeben (wenn Sie das Ausmaß des Viewports auf 10*10 Einheiten festlegen, beschränkt dies die Ausgabe nicht auf 10*10 Geräteeinheiten, sondern füllt weiterhin die gesamte Ausgabefläche – zur Beschränkung der Ausgabe müssten Sie Clipping verwenden).

Vielmehr interessiert sich das System nur für das *Größenverhältnis* zwischen logischer Einheit und Geräteeinheit. Im obigen Beispiel bedeutet das z.B., dass in X-Richtung 100 logische Einheiten auf *PaintBox.Height* Geräteeinheiten abgebildet werden. Die beiden folgenden Anweisungen bewirken, dass eine logische Einheit sowohl in X- als auch in Y-Richtung in fünf Geräteeinheiten umgewandelt wird:

```
{ Diese Reihenfolge des Aufrufs ist unter Windows vorgeschrieben: }
  SetWindowExt(1, 1);
  SetViewportExt(5, 5);
```

Gegenüber dem Gerätekoordinatensystem ergibt sich so eine Vergrößerung um den Faktor 5.

Umdrehung der Achsenrichtungen

Auch in den Modi *mmIsotropic* und *mmAnisotropic* sind die Achsenrichtungen zunächst so definiert, dass die Y-Koordinaten nach unten hin ansteigen. Um das zu ändern, genügt es, wenn Sie wie in Abbildung 5.8 eine negative Höhe des Viewports angeben:

```
with VCanvas do begin
  SetWindowExt(100, 100);
  SetViewportExt(Width, -Height);
  ...
```

Größenverhältnisse im Modus MM_ISOTROPIC

Wenn Sie den Modus *MM_ISOTROPIC* (bzw. *mmIsotropic* von *TVirtualCanvas*) verwenden, passt Windows die von Ihnen gemachten Angaben so an, dass eine logische X-Einheit auf dem Ausgabegerät genauso groß ist wie eine logische Y-Einheit. Dabei wird die Grafik unter Umständen etwas kleiner, so dass sie entweder in der Breite oder in der Höhe nicht mehr das gesamte Fenster abdeckt. Inwieweit die von Ihnen gemachten Angaben automatisch angepasst wurden, können Sie genau erfahren, wenn Sie die Windows-API-Funktionen *GetWindowExtEx*, *GetWindowOrgEx*, *GetViewportExtEx* und *GetViewportOrgEx* aufrufen.

5.6.3 Das Koordinatensystem des TreeDesigners

Der TreeDesigner stellt sein Koordinatensystem ein, indem er die Methoden von *TVirtualCanvas* aufruft.

Wenn Sie die API-Funktionen von Windows direkt aufrufen wollen, so ist eine Übertragung der Listings aus diesem Kapitel leicht möglich, indem Sie an den Namen der *TVirtualCanvas*-Methoden ein »Ex« anhängen, als ersten Parameter das *Handle* des verwendeten *Canvas* anfügen und als letzten Parameter ein *nil*. Die Abbildungsmodi stellen Sie unter Windows mit der Funktion *SetMapMode* ein. Auch diese erwartet als ersten Parameter ein Handle, als zweiten Parameter die in der Tabelle von Kapitel 5.6.1 genannten Abbildungsmodus-Namen (z.B. *mm_LoMetric* statt *mmLoMetric*).

Wie am Anfang von Kapitel 5.5.4 genauer beschrieben, beinhaltet die vollständige TreeDesigner-Version auf der CD-ROM die Option, das virtuelle Koordinatensystem ein- und auszuschalten, und auch der Programmcode enthält entsprechende Fallunterscheidungen. Der in diesem Kapitel abgedruckte Code enthält nur die Anweisungen, die bei *eingeschaltetem* virtuellen Koordinatensystem gültig sind.

Anpassung der Zeichenfläche *R100*

Der *TreeDesigner* arbeitet am Bildschirm im Abbildungsmodus *mmIsoTropic* und druckt im Modus *mmLoMetric*. Über den ScrollBar in der Werkzeugleiste können Sie einen Vergrößerungsfaktor zwischen 0.1 und 10 einstellen.

Spätestens an dieser Stelle ist es sinnvoll, die Größe der Zeichenfläche in virtuellen Koordinaten festzulegen. Sie wird in der Unit *GrDoc* dadurch festgelegt, dass die Properties *Width* und *Height* des Grafikdokuments wie folgt initialisiert werden:

```
constructor TGraphicDoc.Create;
begin
  FWidth := 3000;
  FHeight := 3000;
  ...
```

Der Schritt vom Geräte- zum virtuellen Koordinatensystem soll sich nahtlos an die gewählte Methode des Scrollens anschließen. Das heißt, die Klasse *TScrollBox* soll auch dann, wenn die Grafik verkleinert oder vergrößert ist, die beiden folgenden Aufgaben automatisch erledigen:

- Der scrollbare Bereich soll die gesamte Größe der Grafik umfassen
- und beim seitenweisen Blättern mit den Bildlaufleisten soll die Grafik jeweils um den sichtbaren Bildbereich verschoben werden.

Das funktioniert nur, wenn die Paintbox-Zeichenfläche weiterhin den gesamten virtuellen Grafikbereich umfasst. Bei einer Vergrößerung muss also auch die Größe der

Paintbox weiterwachsen, so wie sie bei einer Verkleinerung schrumpfen soll. Genau genommen errechnet sich die Größe der Paintbox aus den virtuellen Größen *Document.Width* und *Document.Height*, jeweils multipliziert mit dem Vergrößerungsfaktor. Beim maximalem Vergrößerungsfaktor 10 wird die Paintbox also zu einem 30000 Einheiten großen Quadrat, womit die 16-Bit-Grenzen fast erreicht wären.

Als weitere Maßnahme ist es notwendig, die *TScrollBox*-Properties *VertScrollBar.Range* und *HorzScrollBar.Range* an die geschrumpfte bzw. gewachsene Zeichenfläche anzupassen, denn *TScrollBox* verändert den scrollbaren Bereich nicht automatisch, wenn sich eines der Kindfenster verändert.

Beide Maßnahmen sind in der Methode *ResizePaintbox* zusammengefasst, die auch noch die gesamte Paintbox mit *Invalidate* für ungültig erklärt, denn nach einer Änderung des Vergrößerungsfaktors muss die gesamte Grafik neu ausgegeben werden:

```
procedure TDocumentForm.ResizePaintbox;
begin
  PaintBox.Width := trunc(Document.Width * ZoomFactor);
  PaintBox.Height := trunc(Document.Height * ZoomFactor);
  PaintBox.Invalidate;
end;
```

ResizePaintBox wird im einfachsten Fall aufgerufen, wenn das im Folgenden besprochene Property *ZoomFactor* geändert wird.

Das ZoomFactor-Property

Der Vergrößerungsfaktor selbst befindet sich in der privaten Variablen *FZoomFactor*, die als Property *ZoomFactor* von der Methode *SetZoomFactor* geschrieben wird. *SetZoomFactor* ruft nur dann die Methode *ResizePaintbox* auf, wenn sich der Vergrößerungsfaktor auch geändert hat:

```
  TDocumentForm = class(TForm)
    ...
  private
    FZoomFactor: real;
    procedure SetZoomFactor(NewValue: real);
  public
    property ZoomFactor: real read FZoomFactor write SetZoomFactor;
    ...

procedure TDocumentForm.SetZoomFactor(NewValue: real);
begin // vereinfachte Version, siehe Hinweis unten
  // Der Faktor muss zwischen 0,1 und 1 liegen:
  if NewValue < 0.1 then NewValue := 0.1;
  if NewValue > 10 then NewValue := 10;
  if FZoomFactor <> NewValue then begin
    FZoomFactor := NewValue;
```

```
      ResizePaintbox; // siehe unten
    end;
    // Aktualisierung des Labels neben dem TrackBar:
    DecimalSeparator := '.'; // kein Komma verwenden
    ZoomLabel.Caption := FormatFloat('#0.####', ZoomFactor);
  end;
```

Die wohl häufigste Ursache für die Änderung des Properties *ZoomFactor* ist die *OnChange*-Methode des Vergrößerungsfaktor-Scrollbars (weitere Ursachen sind ein paar Menüpunkte aus dem Ansichtsmenü und die Initialisierung des Fensters beim *OnCreate*-Event). Da ein Scrollbar nur ganze Zahlen als Position liefert, wurde statt eines Bereiches von 0,1 bis 10 der Bereich 1 bis 100 gewählt, folglich muss die Scrollbar-Position wieder in einen Vergrößerungsfaktor umgerechnet werden. Für diese Umrechnung können Sie über das Popup-Menü des TrackBars zwischen zwei Modi wählen: einem linearen Modus, in dem der Bereich zwischen Faktor 0,1 und 1 im ersten Zehntel des TrackBars liegt, und einem exponentiellen Modus, in dem dieser Bereich über ungefähr drei Viertel des TrackBars ausgedehnt ist. Die *OnChange*-Methode berücksichtigt diese beiden Alternativen:

```
procedure TDocumentForm.ZoomFactorBarChange(Sender: TObject);
begin { vereinfachte Version }
  // LinearZooming wird über das Popupmenü des TrackBars eingestellt:
  if LinearZooming then ZoomFactor := ZoomFactorBar.Position / 10
    else ZoomFactor : =0.1 + exp(ZoomFactorBar.Position/10) / 2000;
end;
```

Hinweis: Damit beim Verändern des Vergrößerungsfaktors Schritt für Schritt der Eindruck entsteht, Sie würden auf das Zentrum des Bildes zoomen (oder von diesem wegzoomen), merkt sich die vollständige Methode *SetZoomFactor* vor der Änderung der Paintbox-Größe die Koordinaten des Punktes in der Mitte des Fensters und rückt diesen Punkt nach der Ausführung der oben abgedruckten Zeilen erneut in die Bildmitte.

Einstellung des Abbildungsmodus

Da der Abbildungsmodus und die zugehörigen Daten an mehreren Stellen eingestellt werden müssen, fasst das Dokumentformular des TreeDesigners alle dazugehörigen Aufrufe in einer Methode zusammen:

```
function TDocumentForm.SetMapMode(DestCanvas: TCanvas;
                  DestControl: TControl): TVirtualCanvas;
begin
  Result := PaintboxVCanvas;
  Result.VCLCanvas := DestCanvas;
  if not NoMapMode.Checked then begin
  { Der folgende Aufruf von SetViewPortOrg überträgt die Verschiebung
```

```
      der Paintbox innerhalb der Scrollbox auf das virtuelle
      Koordinatensystem. }
      Result.SetViewPortOrg(
               DestControl.Left + PaintBoxHMargin div 2,
               DestControl.Top + PaintBoxVMargin div 2)
      Result.MapMode := mmIsoTropic;
      // Ursprung des Koordinatensystems nach unten legen:
      Result.SetWindowOrg(0, Document.Height);
      // Größe des abgebildeten Dokuments einstellen
      Result.SetWindowExt(Document.Width, -Document.Height);
      // Größe der Abbildungsfläche einstellen (= Größe der Paintbox)
      Result.SetViewPortExt(DestControl.Width - PaintBoxHMargin,
                            DestControl.Height - PaintBoxVMargin);
    end;
  end;
```

Koordinatenumrechnung

Mit den bisher getroffenen Maßnahmen können Sie eine bestehende Grafik im *TreeDesigner* zwar ansehen und blättern; einige Dinge würden so jedoch ohne weitere Vorbereitung noch nicht funktionieren. Wenn Sie in einer von 1.0 verschiedenen Vergrößerungsstufe Objekte einzeichnen oder sonstige Mausaktionen durchführen wollen, geriete die Grafik aus den Fugen: Neu gezeichnete Elemente erscheinen nicht mehr unter dem Mauszeiger und sind vielleicht überhaupt nicht sichtbar, *TGraphicElement.Invalidate* lässt nicht mehr die richtigen Bereiche des Fensters neu zeichnen und auch das Abschneiden der Schrift durch manuelles Clipping ist nicht mehr treffsicher. Solche Probleme können entstehen, wenn Sie in einem Funktionsaufruf virtuelle Koordinaten verwenden, wo Gerätekoordinaten gefragt sind, oder umgekehrt.

DPtoLP *R77*

Die erste Richtung gilt bei den Methoden für die Mausereignisse: Sie erhalten Koordinaten, die in Geräteeinheiten angegeben sind, die aufgrund des virtuellen Koordinatensystems nicht mehr den Koordinaten der Grafik entsprechen. Eine einfache Windows-Funktion verspricht hier Abhilfe: *DPtoLP* ist eine Abkürzung für »Device Points to Logical Points« und transformiert dementsprechend Gerätekoordinaten in logische Koordinaten.

Die in Kapitel 5.4.1 abgedruckten Methoden für die Bearbeitung von *OnMouseDown*, *OnMouseMove* und *OnMouseUp* berücksichtigen die virtuellen Koordinatensysteme, indem sie die *x/y*-Werte der Maus durch die Methode *MapMousePos* umrechnen lassen. *MapMousePos* besteht im Wesentlichen nur aus dem Aufruf der *DPtoLP*-Funktion, sie muss die Mausposition, die relativ zur Ecke der Scrollbox übergeben wird, lediglich noch relativ zur Paintbox umrechnen, indem sie die Scrollposition der Scrollbox einbezieht:

```
procedure TDocumentForm.MapMousePos(var x, y : integer);
var
  P : TPoint;
begin
  P := Point(x-ScrollBox.HorzScrollbar.Position,
             y-ScrollBox.VertScrollbar.Position)
  PaintboxVCanvas.DPtoLP(P);
  x:=P.x; y:=P.y;
end;
```

Im TreeDesigner wurde *DPtoLP* wieder in einer gleichnamigen Methode von *TVirtualCanvas* gekapselt, die in obigem Listing statt der Windows-Funktion aufgerufen wird. Dabei wurde wie schon bei den *SetWindow/Viewport*-Methoden die Übergabe eines Gerätekontext-Handles eingespart.

LPtoDP

Entsprechend funktioniert die Umwandlung von logischen Koordinaten in Gerätekoordinaten mit der Funktion *LPtoDP*. Gerätekoordinaten werden im TreeDesigner benötigt, um den Clipping-Bereich einzustellen und rechteckige Bereiche des Fensters für ungültig zu erklären. Der Quelltext für die Ungültigerklärung der Fensterbereiche ist bereits in Kapitel 5.5.3 gezeigt worden und soll als Beispiel genügen.

5.6.4 Drucken

So wie die Grafiktreiber für die Bildschirmausgabe zuständig sind, gibt es für die Drucker die Druckertreiber, die es einer Windows-Applikation ersparen, direkt mit dem Drucker kommunizieren zu müssen. Durch diese geräteunabhängige Grundstruktur können Sie dieselben Grafikmethoden von *TCanvas*, mit denen Sie im Fenster zeichnen, auch für die Druckerausgabe verwenden. Allerdings gibt es beim Drucken einige Abweichungen zur Grafikausgabe im Fenster:

- Die Auflösung eines Druckers ist erheblich höher als die der heutigen Bildschirme.
- Die Seite, auf der der Drucker druckt, verhält sich insofern anders, als dass sie sich nur zeitweise im Drucker befindet. Beim Drucken auf mehreren Seiten müssen Sie steuern, wann die Seite gewechselt wird. Auch dann, wenn Sie nur eine Druckseite verwenden, müssen Sie festlegen, wann diese beginnt und wann sie ausgeworfen werden kann.
- Drucker weisen untereinander und im Vergleich zum Bildschirm unterschiedliche Fähigkeiten und Beschränkungen auf. So können beispielsweise Plotter keine Bitmaps zeichnen. Die Tatsache, dass Drucker die gedruckte Grafik nicht mit weißer Farbe übermalen können, wird bereits in den Grafiktreibern (oder Druckern) berücksichtigt, die eine Seite im Speicher aufbauen und erst dann drucken, wenn sie vollständig fertig ist.

- Die Zeichenfläche des Druckers ist geringfügig komplizierter zu erreichen als die Zeichenfläche eines Formulars: Sie müssen zuerst die Unit *Printers* einbinden und dann auf deren *Printer*-Objekt zugreifen, um an die Drucker-Zeichenfläche (*Printer.Canvas*) zu gelangen.

Textausgabe schnell und einfach *R101*

Sie können in Delphi auf zwei verschiedene einfache Weisen Ausgaben an den Drucker senden: Zur reinen Textausgabe kommen Sie mit den schon fast antiken Prozeduren *write* und *writeln* sowie mit der Prozedur *AssignPrn* aus. Für Letztere müssen Sie die Unit *Printers* in die Unit einbinden, in der Sie den Drucker verwenden wollen.

AssignPrn verbindet den Drucker mit einer Textdateivariablen, die Sie so verwenden können wie jede andere derartige Variable, die mit einer normalen Datei verbunden ist (siehe Kapitel 2.8.2). Das bedeutet, dass Sie die »Datei« (im folgenden Beispiel die Variable *DruckerAlsDatei*) mit *Rewrite* öffnen und am Schluss mit *Close* wieder schließen müssen. Zwischen diesen beiden Anweisungen geben Sie die zu druckenden Zeilen mit der Standardfunktion *writeln* aus, der Sie eine beliebig lange Liste von Parametern verschiedenen Typs (ohne die eckigen Klammern, die bei anderen Funktionen mit variabler Parameterzahl notwendig sind) übergeben können. Das folgende Beispiel ist dem Projekt *TextPrinter* von der CD-ROM entnommen:

```
var
  DruckerAlsDatei: System.Text;
begin
  AssignPrn(DruckerAlsDatei);
  Rewrite(DruckerAlsDatei);
  Writeln(DruckerAlsDatei, 'Dateiausgabe vom ', DateToStr(Now));
  for i := 0 to Memo1.Lines.Count - 1 do
    Writeln(DruckerAlsDatei, Memo1.Lines[i]);
  ...
  System.Close(DruckerAlsDatei);
  ...
```

Da in Formularen die Bezeichner *Close* und *Text* bereits für eine Methode bzw. für ein Property reserviert sind, müssen Sie in Formularmethoden den Bezeichner *System* vor diese Bezeichner stellen, damit der Compiler erkennt, dass Sie nicht die Methode bzw. das Property des Formulars, sondern die Funktion bzw. den Typ der Standardbibliothek *System* meinen.

AssignPrn verbindet die Textdatei nicht direkt mit dem Drucker, sondern mit dem Druckerobjekt *Printer* der Unit *Printers* (im Stichwortverzeichnis zu finden unter der Klasse *TPrinter*). Dieses leitet die Textdaten als Grafik an den Windows-Druckertreiber weiter und verwendet dabei die in seinem Property *Canvas.Font* angegebene Schriftart. Um also die Schriftart für den gedruckten Text zu setzen, schreiben Sie sie in das *TFont*-Objekt *Printer.Canvas.Font*. Das zuständige *Printer*-Objekt erhalten Sie als Rückgabe-

wert von der Funktion *AssignPrn*, so dass in obigem Listing die erste Anweisung etwa wie folgt erweitert werden könnte, um die Schrift auf die aktuelle Wahl in einem Schriftdialog zu setzen:

```
AssignPrn(DruckerAlsDatei).Canvas.Font := FontDialog1.Font;
```

Grafikausgabe schnell und einfach *R102*

Auch bei der Grafikausgabe kommt *Printer.Canvas* ins Spiel, und zwar verwenden Sie dieses Objekt wie die Zeichenfläche des Formulars. So können Sie mit *Draw* ganze Grafiken wie Bitmaps oder Metadateien ausgeben, sie mit *StretchDraw* vergrößern, so dass sie auch auf dem Drucker nicht zu klein erscheinen, und einfache Grafikbefehle wie *LineTo* und *TextOut* verwenden. Für die Seitensteuerung müssen Sie vor der ersten Ausgabe auf die Seite die Methode *BeginDoc*, für jede neue Seite die Methode *NewPage* und am Ende des Druckauftrags die Methode *EndDoc* aufrufen. Das Grundgerüst der Grafikausgabe sieht demnach so aus:

```
Printer.BeginDoc;
with Printer.Canvas do begin
  ... Grafikausgabe ...
end;
Printer.EndDoc;
```

Falls Sie einen mit *BeginDoc* gestarteten Druckauftrag abbrechen wollen, rufen Sie statt *EndDoc* die Methode *Abort* auf.

Druckerausgabe und Abbildungsmodi

Die Portierung der Ausgabe vom Fenster auf den Drucker hat den großen Unterschied in der Auflösung zu bedenken. Ein unverändert auf den Drucker übertragener Fensterinhalt schrumpft dort beträchtlich. Die einfachste Lösung wäre hier die Verwendung der Formularmethode *TForm.Print*, die jedoch eher dazu geeignet ist, den Aufbau eines Dialogformulars zu drucken, denn sie würde den Inhalt des Formulars nur als vergrößerte Pixelgrafik ausgeben.

Die bessere Möglichkeit für eine optimale Nutzung der Druckerseite wäre die Verwendung des Abbildungsmodus *MM_ISOTROPIC* im Bildschirmfenster und auf dem Drucker. Sie könnten dann die gesamte Zeichenfläche am Bildschirm in die gleiche Anzahl virtueller Koordinaten aufteilen wie die Druckerseite und könnten Bildschirmfenster und Druckerseite mit denselben Grafikausgabebefehlen bedienen. (Um die Ausmaße des Drucker-Viewports mit *SetViewportExt(Ex)* einstellen zu können, müssen Sie wissen, wie viele Geräteeinheiten diese groß ist; wir kommen hierauf am Schluss des Kapitels zurück.)

Wenn Sie die Größe des Ausdrucks jedoch mit festen Maßeinheiten steuern wollen, können Sie entweder

- die Auflösung des Druckers feststellen und die Grafik auf dieser Grundlage in Gerätekoordinaten berechnen und ausgeben
- oder die Abbildungsmodi des GDI verwenden, die mit festen Maßeinheiten arbeiten.

Obwohl Sie nur mit der ersten Methode jeden zur Verfügung stehenden Punkt des Druckers gezielt ansprechen können, dürften die Abbildungsmodi des GDI mit einer maximalen Genauigkeit von 0,018 mm (*MM_TWIPS*) für die meisten Zwecke ausreichen.

In einem seiner beiden Druckmodi steuert auch der TreeDesigner den Ausdruck in genauen Maßeinheiten, im anderen Modus nimmt er es nicht so genau, lässt dem Benutzer dafür aber im Drucken-Dialog mit Vorschau eine erheblich größere Flexibilität.

TreeDesigner-WYSIWYG 1/2 *R103*

Im ersten seiner beiden Druckmodi verwendet der TreeDesigner den Abbildungsmodus *MM_LOMETRIC* und legt den Ursprung des Drucker-Viewports auf die linke untere Ecke der Seite.

Eine Einheit auf dem Drucker ist im Modus *MM_LOMETRIC* 0,1 mm groß. Das heißt also, dass 100 virtuelle Einheiten auf dem Bildschirm zu einem Zentimeter Grafik auf dem Drucker werden. Damit der Druck genau mit der Darstellung auf dem Bildschirm übereinstimmt, muss der Vergrößerungsfaktor auf dem Bildschirm angepasst werden, und zwar so, dass 100 an den Linealen angezeigte Einheiten ein Zentimeter groß dargestellt werden (dies ist ungefähr bei Faktor 0,2 der Fall).

Beim Drucken werden die logischen Einheiten der Grafikobjekte unverändert an den Drucker gesendet. Bis auf die wahrscheinlich leicht unterschiedliche Größe ist die Bildschirmdarstellung damit eine WYSIWYG-Darstellung (what you see is what you get), die der Druckerausgabe entspricht, soweit es die Unterschiede zwischen Drucker und Bildschirm zulassen. Eine 29,7 cm lange DIN-A4-Seite passt mit ihren 2970 logischen Einheiten gerade in die 3000 Einheiten hohe Zeichenfläche hinein.

Der erste Druckmodus des TreeDesigners lässt Ihnen zwar beim Zeichnen einer Grafik die Kontrolle über die genaue Größe der Grafikelemente auf dem Ausdruck, dies ist jedoch nicht immer die Art der Kontrolle, die man bei einem Grafikprogramm dieser Art benötigt: So wie am Bildschirm sollte sich die Grafik auch für den Ausdruck skalieren lassen, damit sie z.B. genau auf eine Seite eingepasst werden kann. Um die Grafik

von Windows automatisch skalieren zu lassen, bietet sich hier wieder der Abbildungsmodus *MM_ISOTROPIC* an, den der TreeDesigner schon bei der Bildschirmausgabe verwendet.

TreeDesigner-WYSIWYG 2/2: Druckvorschaudialog *R104*

Dies führt uns zur zweiten Form der WYSIWYG-Darstellung im TreeDesigner, der Vorschaugrafik im Druckdialog (Abbildung 5.9). Anhand der Vorschaugrafik können Sie in diesem Dialog im Modus *Druckgröße anpassen* genau einstellen, wie groß und an welcher Position der Seite die Grafik gedruckt werden soll.

- Im Bereich *Zu druckender Ausschnitt* werden die Koordinaten des Ausschnittes angegeben, der gedruckt werden soll. Da dies normalerweise immer die gesamte Grafik ist, genügt hier ein Druck auf *Optimal*, um den Druckdialog selbst herausfinden zu lassen, wie groß die Grafik ist (der Dialog bedient sich dazu der in Kapitel 5.3.2 erwähnten Methode *TGraphicDoc.GetPaintRect*). Per Voreinstellung wird der gewählte Ausschnitt so groß gedruckt, dass er die gesamte Seite bedeckt (allerdings ohne das Seitenverhältnis zu verzerren, wie es im Modus *MM_ANISOTROPIC* möglich wäre).
- Der Bereich *Zielfläche in Prozent* dient dazu, Ränder einzustellen. Mit *X* und *Y* verschieben Sie die Grafik von der linken oberen Ecke nach rechts und nach unten, mit *Breite* und *Höhe* können Sie die Voreinstellung von 100% des Seitenmaßes verkleinern.

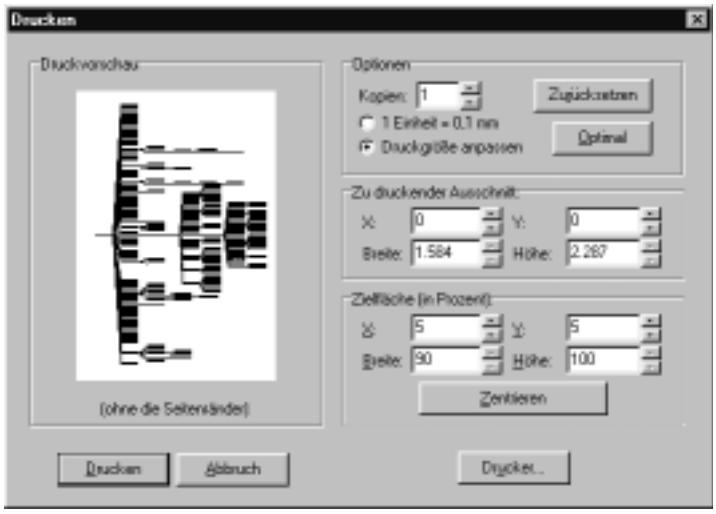


Abbildung 5.9: Nach Drücken des Schalters »Optimal«, Verkleinern der Zielfläche auf 90% und Verschieben der Zielfläche um jeweils 5% in X- und Y-Richtung wird diese Grafik zentriert gedruckt und füllt die Seite fast ganz aus.

Eine Beispieleinstellung dieser Werte finden Sie in Abbildung 5.9. Zur Implementierung dieses zweiten Druckmodus sind zwei neue Einstellungen von Koordinatensystemen erforderlich:

- Für die Vorschau muss die gesamte Grafik bzw. der gewählte Ausschnitt in einen vergleichsweise winzigen Bereich des Dialogs gezwängt werden, der in Pixeln angegeben wird (ca. 200*230 Pixel, der Dialog enthält dafür eine *TPaintBox*-Komponente).
- Für den Ausdruck muss der gewählte Ausschnitt der Grafik auf den im Dialog angegebenen Zielbereich abgebildet werden, der in Druckereinheiten angegeben wird. Bei einem 600-dpi-Drucker sind das z.B. ca. 4600*6800 Einheiten für die gesamte Seite.

Im Folgenden ist der Code abgedruckt, der im Falle des frei skalierbaren Modus die Druckvorschau zeichnet. Dafür wird das Koordinatensystem so eingestellt, dass der in den Eingabefeldern des Dialogs eingestellte Dokumentbereich auf die (ebenfalls im Dialog eingestellte) Zielfläche des Papiers (hier das Papier in der Vorschau-Grafik) projiziert wird:

PrevX, *PrevY*, *PrevW* und *PrevH* werden vorher in derselben Methode berechnet, sie geben die Position, Breite und Höhe des Teils im Vorschaubereich an, der der Einstellung der *Zielfläche* im Dialog entspricht (also einen Teilbereich innerhalb der weiß gezeichneten Seitenfläche; zur Berechnung dieser Werte siehe die CD-ROM). Auf genau diesen Bereich soll nämlich der gewählte Ausschnitt der Grafik abgebildet werden. Für die Einstellung dieses Ausschnitts stellt der Dialog, wie in Abbildung 5.9 dargestellt, vier Eingabefelder zur Verfügung, deren *TUpDown*-Elemente *WinX*, *WinY*, *WinW* und *WinH* in den obigen Zeilen abgefragt werden.

Dies ist jedoch nur ein kleiner Auszug aus dem vollständigen Code, der auch noch den weißen Seitenhintergrund zeichnen, Ränder vom Seitenhintergrund zu den Grenzen des Vorschaubereichs berechnen und die verschiedenen alternativen Dialogeinstellungen berücksichtigen muss.

Hinweis: Die Verkleinerung der Ausgabefläche im Dialog ist nur dann wirksam, wenn Sie die gesamte Grafik ausdrucken, denn der TreeDesigner verwendet beim Drucken kein zusätzliches Clipping. Es wird also auch außerhalb der Ränder gedruckt, wenn die Grafik noch über den Bereich hinausgeht, den Sie in *Zu druckender Ausschnitt* eingestellt haben.

Geräteunabhängigkeit in der Praxis

Das elementare Prinzip, das beim Drucken und beim Vorschaudialog verwendet wird, ist dasselbe wie beim Zoomen der Grafik auf dem Bildschirm: die völlige Geräteunabhängigkeit eines virtuellen Koordinatensystems. Überall wird die TreeDesigner-Grafik mit einem einzigen Aufruf von *TGraphicDoc.PaintAll* ausgegeben:

- am Bildschirm, wenn dort das Gerätekoordinatensystem verwendet wird,
- am Bildschirm, wenn dort ein virtuelles Koordinatensystem mit einer beliebigen Vergrößerungseinstellung verwendet wird,
- in der Druckvorschau, bei der die Grafik noch stärker verkleinert wird als beim kleinsten Zoomfaktor im Grafikfenster,
- beim Drucken im Modus *MM_LOMETRIC*,
- beim Drucken im frei skalierbaren Modus,
- beim Erstellen einer Metadatei für die Zwischenablage.

Druckereinrichtung

Wir kommen nun zur Implementierung der Menüpunkte für das Drucken. Der TreeDesigner stellt dafür in seinem Dateimenü die Punkte DRUCKEN... und DRUCKEREINRICHTUNG... zur Verfügung.

Für die Druckereinrichtung wird der von der VCL vordefinierte Einrichtungsdialog verwendet; er wirkt sich auf das im Folgenden verwendete *Printer*-Objekt der VCL aus und bedarf wohl keiner weiteren Erläuterung:

```
procedure TDocumentForm.CmPrinterSetupClick(Sender: TObject);
begin { den Standarddialog TPrinterSetupDialog aufrufen }
  PrinterSetupDialog1.Execute;
end;
```

Die Druckmethode

Nun zur Methode für das *OnClick*-Ereignis des Menüpunkts DATEI | DRUCKEN, durch den der besprochene und in Abbildung 5.9 gezeigte Druckdialog aufgerufen werden soll. Wenn dieser mit *OK* beendet wird, werden die folgenden Properties des Dialogs ausgewertet:

- *Copies.Position* ist der Wert des *TUpDown*-Feldes neben dem Eingabefeld für die Kopienanzahl. Dieser Wert muss nur an *Printer.Copies* weitergegeben werden, damit automatisch die entsprechende Anzahl an Kopien gedruckt wird. (Ein Eingabefeld für die Zahl der Kopien steht Ihnen auch zur Verfügung, wenn Sie den Standard-Druckdialog der VCL, *TPrintDialog*, verwenden).

- *LoMetric.Checked* ist der Zustand des Radioschalters *Druckgröße anpassen*.
- WinX / WinY / WinW / WinH.Position sind die schon erwähnten TUpDown-Werte im Dialogbereich Zu druckender Ausschnitt.
- *VPX/VPY/VPW/VPH.Position* sind die entsprechenden Werte im Dialogbereich *Zielfläche*.

Das Drucken selbst läuft zwischen den schon erwähnten Aufrufen von *BeginDoc* und *EndDoc* des *Printer*-Objekts ab. Das folgende Listing zeigt den Code für die beiden beschriebenen Druckmodi des TreeDesigners:

```
procedure TDocumentForm.PrintClick(Sender: TObject);
var
  VCanvas: TVirtualCanvas;
begin
  Document.DeselectAll; // Markierungen würden sonst mitgedruckt werden
  PrintDialog.Doc := Document; // Dem Dialog das Dokument übergeben
  if PrintDialog.ShowModal = mrOK then begin
    Screen.Cursor := crHourGlass; // Sanduhrsymbol anzeigen
    Printer.Title := 'TreeDesigner Druckauftrag';
    Printer.Copies := PrintDialog.Copies.Position;
    Printer.BeginDoc;
    VCanvas := TVirtualCanvas.Create;
    VCanvas.VCLCanvas := Printer.Canvas;
    with PrintDialog do begin
      if LoMetric.Checked then begin
        // siehe Abschnitt "TreeDesigner WYSIWYG 1/2"
        VCanvas.MapMode := mmLoMetric; { 0.1 mm pro Einheit }
        VCanvas.SetWindowOrg(0, Document.Height);
        // Die im Dialog eingestellte Verschiebung berücksichtigen:
        VCanvas.SetViewportOrg(Printer.PageWidth * VPX.Position div 100,
                               Printer.PageHeight * VPY.Position div 100);
      end else begin
        // siehe Abschnitt "TreeDesigner WYSIWYG 2/2"
        VCanvas.MapMode := mmIsoTropic;
        VCanvas.SetWindowOrg(WinX.Position, WinY.Position);
        VCanvas.SetWindowExt(WinW.Position, -WinH.Position);

        VCanvas.SetViewportExt(Printer.PageWidth * VPW.Position div 100,
                               Printer.PageHeight * VPH.Position div 100);
        VCanvas.SetViewportOrg(Printer.PageWidth * VPX.Position div 100,
                Printer.PageHeight * (VPW.Position+VPY.Position) div 100);
      end;
    end;
    Document.PaintAll(VCanvas);
    Printer.EndDoc;
    Screen.Cursor := crDefault; // normale Mauszeigerform wiederherstellen
  end;
end;
```

Zu einer Zusammenfassung der Properties von *Printer*, die in dieser Methode verwendet werden, und weiterer Properties siehe die nächste Tabelle.

TPrinter-Properties und die Druckerkonfiguration

Die wichtigen Properties der Klasse *TPrinter*, darunter die oben verwendeten Properties *Title*, *Copies*, *PageWidth* und *PageHeight*, sind in der folgenden Tabelle zusammengefasst:

TPrinter-Property	**Beschreibung**
Aborted: Boolean	*True*, wenn der aktuelle Druckauftrag durch den Benutzer abgebrochen wurde; könnte oben in *CmPrintClick* abgefragt werden, um den Druck vorzeitig zu beenden.
Canvas	Zeichenfläche des Druckers, gibt im Property *PixelsPerInch* die Auflösung an, in *Font* können Sie eine Schrift für die Ausgabe von Textdaten (siehe obiger Abschnitt *Textausgabe schnell und einfach*) wählen, während eines mit *BeginDoc* gestarteten Druckauftrags können Sie die gewohnten *TCanvas*-Grafikmethoden aufrufen.
Fonts: TStrings	Liste der vom Drucker unterstützten Schriftarten, inklusive True-Type-Schriften.
Handle: THandle	Handle auf den Drucker, siehe nächster Abschnitt.
Orientation: TPrinterOrientation	*poPortrait* oder *poLandscape* geben an, ob das Papier längs oder quer bedruckt wird.
PageHeight: Integer	Höhe der Seite in Gerätekoordinaten (Pixeln).
PageNumber: Integer	zählt beginnend bei 1 die Seitenzahlen, wird durch *NewPage* erhöht.
PageWidth: Integer	Breite der Seite in Gerätekoordinaten (Pixeln).
PrinterIndex: Integer	Index in das *Printers*-Array, der den gerade gewählten Drucker angibt.
Printers: TString	Liste aller im System installierten Druckertreiber.
Printing: Boolean	*True*, wenn gerade ein Druckauftrag erstellt wird (wenn weder *EndDoc*, noch *Abort* aufgerufen wurde).
Title: String	Titel, den der Druckertreiber für dieses Dokument anzeigen soll.

Zusätzliche Informationen über den Drucker

Neben diesen Informationen benötigt der TreeDesigner für die Implementierung des Menüpunktes ANSICHT | SEITENANZEIGE auch noch die Ausmaße der Seite in Zentimetern. Ohne dieses Maß aus anderen Daten wie Pixelzahl und Auflösung selbst zu berechnen, können wir solche Daten von der Windows-Funktion *GetDeviceCaps* erhalten (mit der *TPrinter* intern ebenfalls arbeitet).

GetDeviceCaps verlangt wieder ein Handle, und zwar das des Druckerobjekts. Und auch dieses Mal befindet sich das Handle in einem Property des entsprechenden VCL-Objekts: *Printer.Handle*. Im zweiten Parameter von *GetDeviceCaps* wird die Art der gewünschten Information angegeben; *GetDeviceCaps* gibt diese in Form eines Integerwerts zurück. Beispielsweise erhalten Sie mit *HorzSize* die Breite der bedruckbaren Fläche in Millimetern:

```
Width := GetDeviceCaps(Printer.Handle, HORZSIZE);
```

Ebenso können Sie mit *VertSize* die Höhe der Fläche erhalten, und mit *Technology* erhalten Sie *DT_RASPRINTER*, falls es sich um einen mit Punkten arbeitenden Drucker handelt. Die zahlreichen weiteren, für Anwendungen teilweise völlig unwichtigen Informationen über den Drucker und den Treiber können Sie abfragen, wie in der Online-Referenz zu *GetDeviceCaps* beschrieben.

Hinweis: Dieselben Informationen wie von *Printer.Handle* können Sie auch von *Printer.Canvas.Handle* erhalten, jedoch können Sie letzteres Handle nur abfragen, solange ein mit *BeginDoc* gestarteter Druckauftrag läuft. Der interne Unterschied zwischen den beiden Handles ist, dass *Printer.Handle* kein vollständiger Gerätekontext ist, sondern nur ein Informationskontext (*Information Context*), der von der API-Funktion *CreateIC* erstellt wird. Seine Zweck ist es beispielsweise, mit *GetDeviceCaps* Informationen abfragen zu können.

Seitenformate

Wenn Sie die Druckertreiber-Informationen über Breite und Höhe einer Seite berücksichtigen, kommt Ihre Anwendung mit jedem Papierformat, das der Benutzer wählt, zurecht. Es spielt noch nicht einmal eine Rolle, ob der Benutzer es längs oder quer bedrucken will (*Portrait* oder *Landscape*).

Im TreeDesigner wird diese Information wie folgt berücksichtigt: Wenn Sie den Menüpunkt ANSICHT | SEITENANZEIGE wählen, werden die derzeitigen Maße der Seite abgefragt und die Abmessungen der Zeichenfläche (*Document.Height* und *Document.Width*) an diese Werte angepasst (ein Millimeter erhält 10 logische Einheiten, da ja im Modus *MM_LOMETRIC* gedruckt wird).

Dadurch wird rechts und unten so viel von den ursprünglichen 3000x3000 Einheiten abgeschnitten, dass die restliche Fläche genau dem bedruckbaren Bereich auf dem Drucker entspricht (die nicht bedruckbaren Ränder werden im TreeDesigner nicht angezeigt, was wieder zu einer Einbuße an WYSIWYG-Qualität führt).

5.7 MDI-Anwendungen

Anwendungen, in denen der Benutzer mehrere Dokumente gleichzeitig bearbeiten kann, halten sich unter Windows häufig an den MDI-Standard oder bilden ihn zumindest nach. Der MDI-Standard sorgt für eine Vereinheitlichung in der Handhabung mehrerer Dokumente und erspart es dem Benutzer auf diese Weise, diese Handhabung in jeder Anwendung aufs Neue erlernen zu müssen.

5.7.1 MDI im Überblick

Das wichtigste und eigentlich schon selbstverständliche Merkmal einer MDI-Anwendung ist, dass jedes Dokument in einem eigenen Fenster, dem *Dokumentfenster* oder *MDI-Kindfenster*, dargestellt wird. Diese Dokumentfenster befinden sich innerhalb des Hauptfensters, können sich beliebig überlappen und sind auch ansonsten in ähnlicher Weise wie die Hauptfenster auf dem Windows-Desktop zu bedienen.

Aufbau einer MDI-Anwendung *R6 I*

Um eine Delphi-Anwendung zu einer funktionsfähigen MDI-Anwendung zu machen, sind die folgenden drei Schritte erforderlich:

- Entwerfen Sie ein MDI-Hauptformular, dessen *FormStyle*-Property Sie auf *fsMDIForm* einstellen. Normalerweise wird dieses Formular automatisch als Hauptfenster der Anwendung erzeugt.
- Entwerfen Sie ein Formular für MDI-Kindfenster mit dem *FormStyle* von *fsMDIChild*. Sie können auch mehrere Typen von Kindfenstern in mehreren Formularen definieren. Normalerweise wird die automatische Erzeugung der Kindformulare in den Projektoptionen abgeschaltet.
- Statt dieser automatischen Erzeugung sollten Sie einen Menüpunkt wie DATEI | NEU zur Verfügung stellen, über den der Anwender zur Laufzeit nicht nur ein Kindfenster, sondern beliebig viele erstellen kann. In der zu diesem Menüpunkt gehörenden Methode erzeugen Sie dynamisch ein neues dieser MDI-Kindfenster (Kapitel 5.7.4).

Die folgenden Schritte sind ebenfalls empfehlenswert, aber optional – teilweise sind sie besonders einfach durchzuführen:

- Geben Sie sowohl dem Hauptformular als auch dem Kindformular ein eigenes Menü (im Property *Menu*). Die VCL verbindet diese Menüs zur Laufzeit automatisch zu einem einzigen Menü (Kapitel 5.7.3).

- Um die gewohnten Funktionen zum Anordnen der Fenster zur Verfügung zu stellen, legen Sie ein *Fenster*-Menü an, dessen *OnClick*-Methoden aus ein oder zwei Zeilen bestehen, oder Sie verwenden eine Aktionsliste mit Standardaktionen und müssen keine einzige Zeile schreiben. Und um eine Fensterliste an dieses Menü anzuhängen, setzen Sie das Formular-Property *WindowMenu*.
- Damit der Benutzer die MDI-Kindfenster auch wieder schließen kann, bearbeiten Sie das *OnClose*-Ereignis (per Voreinstellung wird das Fenster bei einem solchen Ereignis nur zum Symbol verkleinert).
- Nachdem der Benutzer die Anwendung gestartet hat, sollte er die Möglichkeit haben, sofort mit der Arbeit zu beginnen. Die Anwendung sollte also automatisch ein neues Dokumentfenster öffnen. Die letztgenannten drei Punkte werden in Kapitel 5.7.4 behandelt.
- Die Werkzeugleiste oder Mauspalette können Sie lokal für die Dokumentfenster oder global für die gesamte Anwendung anlegen (Kapitel 5.7.5 und 5.7.6).

Delphis MDI-Schablone

Unter Delphis Projektschablonen befindet sich auch eine Schablone für eine MDI-Anwendung mit globaler Symbolleiste, die bereits die wichtigen Schalter zum Öffnen und Schließen von Dokumenten, für Zwischenablageoperationen und zum Schließen der Anwendung enthält (inklusive einer leicht wiederverwendbaren Bilderliste). In dieser Schablone sind bereits die meisten der oben genannten Punkte berücksichtigt, allerdings wird beim Programmstart noch kein Kindfenster automatisch geöffnet und es sind keine von den MDI-Kindfenstern gemeinsam benutzten Mauspaletten oder Hilfsfenster vorhanden. Das MDI-Kindformular der Schablone ist noch vollständig leer, hat also auch noch kein eigenes Menü.

Zwei gegensätzliche Ziele

Bei der Entwicklung einer MDI-Anwendung gibt es zwei gegensätzliche Ziele: Einerseits soll das Dokumentformular weitgehend unabhängig vom Hauptformular sein, weil die Programmstruktur sonst sehr unübersichtlich werden könnte. Andererseits würde der Bildschirmaufbau unübersichtlich werden, wenn jedes MDI-Kindfenster über ein eigenes Menü und eigene Schalter- und Statusleisten verfügen würde. Wenn Sie viele derartige MDI-Kindfenster nebeneinander anordnen würden, würde die kleine, jedem Fenster noch verbleibende Fläche zum größten Teil durch Menü, Mauspalette und Statuszeile gefüllt werden.

Die VCL bietet für die Menüs eine Lösung an, die beiden Zielen entspricht: Sie entwerfen das Dokumentformular als abgeschlossene, unabhängige Einheit mitsamt einem eigenen Menü, und zur Laufzeit blendet die VCL dieses Menü in das Menü des Hauptfensters ein.

Ähnliches geschieht mit Mauspaletten und Statuszeilen nicht automatisch, jedoch finden Sie in Kapitel 5.7.5 eine Beschreibung, wie Sie auch einen solchen Fensterbestandteil des Dokumentfensters im Hauptfenster einblenden können.

5.7.2 MDI- und SDI-Versionen des TreeDesigners

Die SDI-Version des TreeDesigners (Projektname: *TreeDesignerSDI*) unterscheidet sich von der MDI-Version dadurch, dass alle Leisten, die in der MDI-Version im Hauptfenster erscheinen, im Dokumentfenster enthalten sind. In der SDI-Version *ist* das Dokumentformular das Hauptfenster.

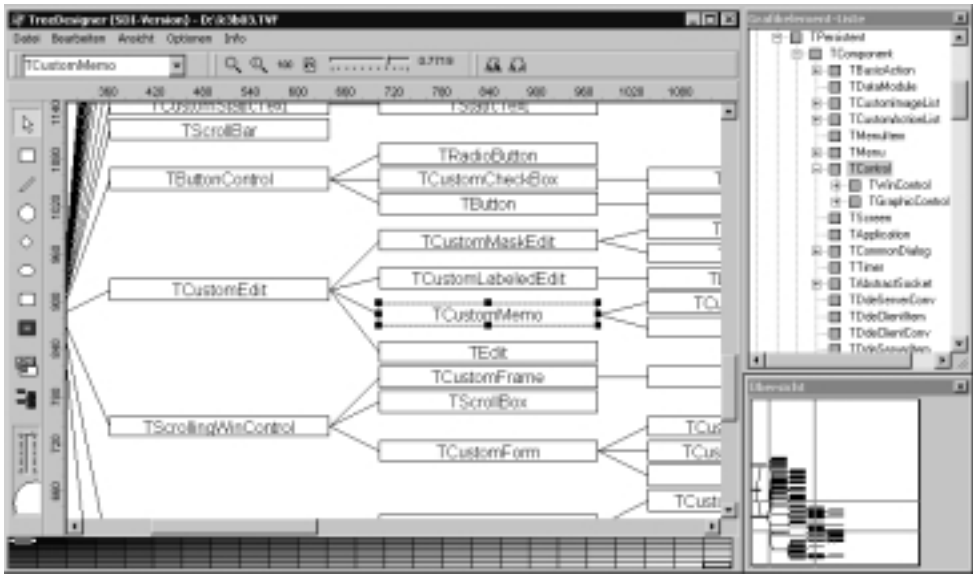


Abbildung 5.10: In der SDI-Version behält das Dokumentfenster seine Toolbars (oben) und erhält eine eigene Werkzeugleiste (links) und eine Farbpalette (unten). Die vom MDI-Hauptformular bekannte Andockmöglichkeit für die Hilfsfenster (rechts) wurde in der SDI-Version nicht übernommen.

In Absehung davon, dass Sie die MDI-Version des TreeDesigners schon in den bisherigen Kapiteln verwenden konnten, nehmen wir hier an, dass wir bisher nur die SDI-Version vorliegen haben (Abbildung 5.10), die wir hier zur MDI-Anwendung ausbauen. Wir wandeln also das Dokumentformular in ein MDI-Kindfenster um, indem

wir das Property *TDocumentForm.FormStyle* auf *fsMDIChild* ändern. Dadurch entsteht auch gleich der Bedarf für ein »neues« Hauptformular.

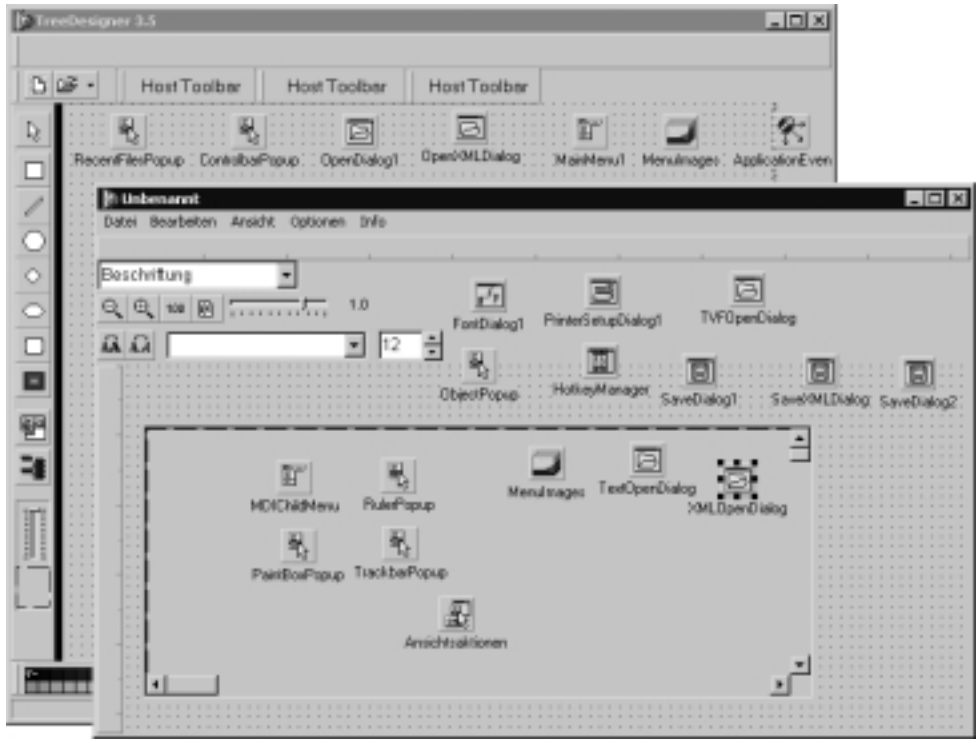


Abbildung 5.11: Die Formulare für Haupt- und Dokumentfenster der MDI-Version zur Entwurfszeit

Abbildung 5.11 zeigt Haupt- und Dokumentformulare im Entwurfsmodus. Das Hauptformular des TreeDesigners heißt *MainForm* und ist in der Unit *MdiForm* definiert. Die wichtigste Property-Einstellung für das neue Hauptfenster findet wieder in *FormStyle* statt. Hier bewirken Sie durch Auswahl von *fsMdiForm*, dass beispielsweise neu erzeugte MDI-Kindfenster automatisch in dieses Formular eingefügt werden, dass die Menüs verschmolzen werden, wie im nächsten Kapitel beschrieben, und dass die dreidimensionale Gestaltung des Rahmens so geändert wird, dass der Hintergrund des Hauptfensters abgesenkt erscheint. Im TreeDesigner wird außerdem das Property *Caption* geändert, damit das Hauptfenster einen angemessenen Titel erhält. In Kapitel 5.7.4 wird als Drittes noch das Property *WindowMenu* angepasst.

Alle weiteren Änderungen am Hauptfenster finden nicht in den Properties statt, sondern über neu eingefügte Komponenten:

- eine *TOpenDialog*-Komponente, die den *Datei-öffnen*-Dialog zum Öffnen eines Dokuments bereitstellt,
- eine *TMainMenu*-Komponente, in der die globalen Menüpunkte DATEI, FENSTER und HILFE untergebracht werden. Zur Laufzeit wird dieses Hauptmenü nur angezeigt, solange keine MDI-Kindfenster geöffnet sind. Wichtig ist vor allem, dass in dieser Situation ein Menüpunkt zum Öffnen einer neuen Datei zugänglich ist. Dies ist durch das globale DATEI-Menü gewährleistet, welches einzig aus den Punkten ÖFFNEN... und NEU besteht und nach Öffnen eines Dokumentfensters durch ein ausführlicheres Menü ersetzt wird.
- eine *TImageList* für die im Hauptmenü anzuzeigenden Symbole. Das Hauptformular kann in diesem Fall *nicht* die Bilderliste des Dokumentformulars mitbenutzen, weil das Dokumentformular dynamisch erzeugt wird, also gar nicht immer ein Exemplar davon existiert. Daher wurde die Bilderliste des Dokumentformulars einfach über die Zwischenablage in das Hauptformular kopiert. Sollte die Bilderliste des Dokumentformulars einmal geändert werden, ist es unter Umständen erforderlich, sie erneut in das Hauptformular zu kopieren. (Möglicherweise werden Sie in ähnlichen Situationen die Benutzung einer gemeinsamen Bilderliste vorziehen. Dies wäre natürlich auch im TreeDesigner möglich, wenn die Liste z.B. in einem Datenmodul untergebracht wäre oder das Dokumentformular die Bilderliste des Hauptformulars mitverwenden würde.)
- eine neue *TControlBar*-Komponente, die mit *alTop* am oberen Rand ausgerichtet wird.
- Aus dem SDI-Formular werden übernommen: die Werkzeugleiste (das mit *alLeft* ausgerichtete Panel) und die *TControlBar*-Komponente vom unteren Fensterrand mitsamt der darin enthaltenen Farbpalette.
- Aus der oberen *TControlBar*-Komponente des SDI-Formulars werden die *MenuToolbar* und die kleine Toolbar mit den Schaltern zum Öffnen von Dateien sowie das zugehörige *RecentFilesPopup* für das Menü der zurückliegenden Dateien übernommen. *Die vier anderen Toolbars bleiben im Dokumentformular*, werden aber mit *Visible:=False* unsichtbar gemacht. Die ControlBar, in der sich diese vier Toolbars befinden, kann entweder auch unsichtbar gemacht oder ganz entfernt werden.
- Als Vorbereitung für die Einblendung der Kind-Toolbars werden in die obere ControlBar des Hauptformulars vier Platzhalter-Panels eingebaut, sie zeigen sich zur Entwurfszeit wie in Abbildung 5.11 durch vier schmale ControlBar-Bänder.

Weitere Umstellungsmaßnahmen

Das folgende Kapitel wird sich zunächst um das Verschmelzen des globalen Hauptmenüs mit den Hauptmenüs der Dokumentfenster kümmern. In Kapitel 5.7.4 wird es um die Verwaltung der Kindfenster gehen, zum Schluss bleibt noch das Symbolleisten-Management. Die obige Aufzählung hat hier schon einen wichtigen Unterschied deutlich gemacht:

- Vier Toolbars des SDI-Formulars werden im Hauptfenster eingeblendet. Für dieses Einblenden entwickelt Kapitel 5.7.5 eine Schnittstelle, über die Sie ähnliche Einblendungsvorgänge leicht in eigene Anwendungen übernehmen können.
- Die Farbpalette, die Toolbar zum Öffnen von Dateien und die Werkzeugleiste zur Auswahl der Grafikform werden schon zur Entwurfszeit in das Hauptformular verlagert. Dies hat zur Folge, dass die darin enthaltenen Komponenten nicht mehr so leicht in der Unit des Dokumentformulars angesprochen werden können. Um dieses Problem wird sich Kapitel 5.7.6 kümmern.

5.7.3 Verschmelzen von Menüs

Die VCL bietet eine einfache Möglichkeit an, sowohl für das MDI-Hauptformular als auch für das Dokumentformular jeweils ein eigenes Hauptmenü im Property *Menu* anzugeben. Wenn ein Dokumentfenster aktiviert wird, blendet die VCL dessen Menü in das Menü des Hauptfensters ein. Jedes Mal, wenn ein anderes MDI-Kindfenster aktiviert wird, konstruiert die VCL das Hauptmenü neu, so dass darin immer die Menüpunkte des gerade aktiven Kindes enthalten sind. Es können also auch Dokumentfenster von verschiedenen Typen gleichzeitig im Hauptfenster enthalten sein. Ist kein MDI-Kindfenster vorhanden, enthält die Menüzeile des Hauptfensters nur das Original-Menü des Hauptformulars, das Sie in dessen *Menu*-Property angegeben haben.

Beim Entwurf der Dokumentformulare erweist es sich als sehr günstig, dass Sie das Menü lokal innerhalb derselben entwerfen können, denn dadurch werden die einzelnen Menübefehle zu Teilen des Dokumentfensters und Sie können die zugehörigen Ereignismethoden im Objektinspektor direkt mit Methoden dieses Fensters in Verbindung bringen. Dies wäre nicht möglich, wenn die Menüpunkte dem Hauptformular gehören würden, denn dann könnten Sie sie auch nur mit Methoden des Hauptformulars verknüpfen.

Sollten Sie das Dokumentfenster einmal außerhalb der MDI-Anwendung als eigenständiges Fenster verwenden, wird das Menü automatisch im Dokumentfenster angezeigt. Sie brauchen also oft kaum mehr als das Property *FormStyle* zu ändern, um ein MDI-Dokumentfenster zu einem eigenständigen Fenster, beispielsweise zum Hauptfenster einer SDI-Anwendung, zu machen.

Der GroupIndex bestimmt den Menüinhalt

Eine wichtige Regel für das (kombinierte) Menü, das zur Laufzeit im Hauptfenster angezeigt wird, ist, dass sich in diesem Menü immer alle Menüpunkte des MDI-Kindes befinden. Wenn Punkte ersetzt werden, dann sind das immer die Punkte des Hauptformular-Menüs. Die kleinste Einheit bei der Verschmelzung von Hauptfenster- und Dokumentmenüs sind die Punkte der obersten Menüebene (also der Menüzeile). Innerhalb der Popup-Menüs findet kein Austausch statt, ein Popup-Menü gehört also entweder ganz zum Haupt- oder ganz zum Dokumentfenster.

Maßgeblich bei der Kombination der Menüs ist hauptsächlich das Property *GroupIndex* der Menüpunkte der obersten Ebene. Per Voreinstellung ist dieses bei jedem Menüpunkt mit dem Wert 0 belegt, was zur Folge hat, dass das Menü des Hauptfensters komplett durch das Menü des Kindfensters ersetzt wird.

Allgemein wird ein Menü des Hauptmenüs immer dann ersetzt, wenn es im Kindmenü einen oder mehrere Punkte gibt, die denselben Gruppenindex besitzen. Hat ein Kindmenüpunkt einen Index, der im Menü des Hauptfensters nicht enthalten ist, so wird dieser Punkt zusätzlich in das Hauptmenü eingefügt, und zwar so, dass die *GroupIndex*-Werte im kombinierten Menü von links nach rechts in aufsteigender Reihenfolge angeordnet sind. Dabei setzt die VCL voraus, dass sich diese Werte in den einzelnen, noch nicht kombinierten Menüs bereits in aufsteigender Reihenfolge befinden.

Verschmelzungs-Beispiel

Im TreeDesigner enthält das Menü des Hauptformulars beispielsweise die Punkte *Datei* (*GroupIndex=1*), *Fenster* (*GroupIndex=9*) und *Hilfe* (*GroupIndex=10*). Die Menüs des Dokumentformulars haben alle einen Gruppenindex von 1. Zur Laufzeit bedeutet das, dass das Dateimenü des Hauptfensters ersetzt wird, da das Dokumentformular Menüpunkte mit demselben Gruppenindex besitzt. Die Menüs *Fenster* und *Hilfe* werden nicht ersetzt, sondern nach den Menüpunkten mit dem Gruppenindex 1 eingefügt. Letzter Menüpunkt bleibt das Hilfemenü mit dem größten Gruppenindex.

Da die Punkte aus dem Dateimenü des Hauptfensters durch das neue Dateimenü ersetzt werden, muss dieses die Punkte des Hauptfenstermenüs wiederholen, wenn diese weiter anwählbar sein sollen. (In diesem Beispiel handelt es sich um die Punkte ÖFFNEN... und NEU.) Das hat jedoch zur Folge, dass das Dokumentformular auch die zugehörigen *OnClick*-Events bearbeiten muss, die schon vom Hauptformular bearbeitet wurden. Normalerweise ruft es dazu einfach die entsprechende Event-Bearbeitungsmethoden des Hauptfensters auf oder verknüpft die *OnClick*-Ereignisse direkt mit diesen (siehe Kapitel 5.7.4).

AutoMerge

Mit dem *TMainMenu*-Property *AutoMerge* können Sie auch bei Nicht-MDI-Anwendungen erreichen, dass das Menü eines untergeordneten Fensters in das Menü des Hauptfensters eingeblendet wird. Standardmäßig ist dieses Property *False*, was bedeutet, dass die Verschmelzung unterbleibt. Die Verschmelzung der Menüs in MDI-Anwendungen findet jedoch auch dann statt, wenn Sie das *AutoMerge*-Property der Dokumentmenüs bei seinem voreingestellten Wert *False* belassen.

Hinweis: Hat *AutoMerge* den Wert *True*, dann wird das Menü nicht in dem Fenster eingeblendet, dem es eigentlich gehört, selbst dann nicht, wenn es gar kein übergeordnetes Fenster gibt, in dem es eine Ersatzunterkunft finden könnte. Das bedeutet, dass Sie zur Laufzeit gar kein Menü erhalten, wenn Sie *AutoMerge* auch beim Hauptformular auf *True* setzen.

5.7.4 Verwaltung der MDI-Kindfenster

Der Entwurf eines Haupt- und eines Kindformulars führt alleine noch nicht zu einer funktionierenden MDI-Anwendung. Das Mindeste, was noch hinzukommen muss, ist eine Möglichkeit, zur Laufzeit neue Kindfenster zu öffnen. Aber auch dann bleiben noch einige kleinere Aufgaben zu erledigen, bis der gewohnte MDI-Komfort erreicht wird. Bevor wir uns diesen Einzelheiten zuwenden, sehen wir uns die Art an, in der die Kindfenster angesprochen werden. (Die MDI-Dokumentfenster werden hier so lange allgemein als Kindfenster bezeichnet, wie es sich nicht um spezielle Fenster handelt, die ein Dokument darstellen.)

Die Kindfenster ansprechen

Drei Properties des Hauptformulars erleichtern Ihnen das direkte Ansprechen der MDI-Kindfenster zur Programmlaufzeit. Zunächst speichert *ActiveMDIChild* das gerade aktive Kindfenster. Wenn Sie auch auf die anderen Kindfenster zugreifen wollen, so finden Sie in *MDIChildCount* die Zahl dieser Fenster und können über das Property *MDIChildren* auf jedes einzelne davon zugreifen (siehe auch *TForm*-Übersicht in Kapitel 3.4.3).

Die globale Formularvariable, die Delphi für jede Formularklasse anlegt, wird für die Kindfenster nicht benötigt. Normalerweise dient sie dazu, eine einzelne Fensterinstanz zu speichern. Wenn Delphi ein Formular automatisch erzeugt (durch den *CreateForm*-Aufruf, siehe Kapitel 1.5.3), so verwendet es dazu diese globale Variable. Da der Benutzer jedoch zur Programmlaufzeit beliebig viele MDI-Kindfenster öffnen kann, ist diese Variable völlig unzureichend. Selbst wenn sie von Delphi dazu verwendet wird, beim Programmstart automatisch ein Kindfenster zu öffnen, sollten Sie sie nach dem Pro-

grammstart nicht mehr verwenden, denn der Benutzer kann üblicherweise jedes Dokumentfenster jederzeit schließen und dadurch die globale Variable ungültig machen.

Kindfenster erzeugen *R33*

Die Erzeugung eines Kindfensters läuft nach dem gleichen Muster ab wie die dynamische Erzeugung beliebiger anderer Objekte, und zwar über einen Konstruktoraufruf. Die Erzeugung eines Fensters sollte in einer eigenen, von Ereignissen unabhängigen Methode gekapselt werden. Im TreeDesigner heißt sie *NewWindow* und erwartet als Parameter ein bereits erzeugtes Dokument-Objekt (zur Unabhängigkeit zwischen Dokument und Fenster siehe Kapitel 5.3.1):

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
  Window: TDocumentForm;
begin
  { vereinfachter Create-Aufruf (Erweiterung in Kapitel 5.7.5): }
  Window := TDocumentForm.Create(self, Document);
  Window.Visible := True;
  Window.Show;
  Result := Window;
end;
```

Diese Methode kann dann zu mehreren Anlässen aufgerufen werden:

- als Antwort auf den Menübefehl DATEI | NEU,
- wenn die Anwendung gestartet wird und
- beim Menübefehl DATEI | ÖFFNEN: Der TreeDesigner erzeugt bei Anwahl dieses Menüpunkts mit *NewWindow* zuerst ein neues leeres Kindfenster und lädt dann eine Datei in dieses hinein.

Die Methoden für die beiden Punkte des Datei-Menüs sind schnell programmiert, da sie nur aus Aufrufen der oben gezeigten Methode *NewWindow* bzw. aus Methoden, die in anderen Kapiteln besprochen werden, bestehen:

```
{ OnClick-Ereignis von Datei|Öffnen }
procedure TMainForm.MDIOpenClick(Sender: TObject);
var
  NewChild: TDocumentForm;
  Document: TGraphicDoc;
begin
  if OpenDialog1.Execute then begin
    Document := AddDoc(OpenDialog1.FileName); { sucht das Dokument unter
       den bereits geöffneten bzw. öffnet ein neues, siehe Kapitel 5.3.1 }
    NewChild := NewWindow(Document);
  end;
```

```
end;

{ OnClick-Ereignis von Datei|Neu }
procedure TMainForm.MDINewClick(Sender: TObject);
begin
  NewWindow(AddDoc(''));
end;
```

Hinweis: Wichtig ist, dass Sie in einer Methode wie *NewWindow* nicht das *Parent*-Property des neuen Kindfensters setzen. Dies ist nur dann erforderlich, wenn Sie ein Nicht-MDI-Kind als Unterfenster in ein Hauptfenster einfügen wollen. Für MDI-Kindfenster bleibt das *Parent*-Property *nil*, sonst kann das Hauptfenster bei der Darstellung des Kindfensters Probleme bekommen und es erscheinen beispielsweise unnötige Bildlaufleisten.

Kindfenster bei Programmstart öffnen *R38*

Etwas schwieriger ist es, schon beim Programmstart ein neues Dokumentfenster anzulegen. Im Zusammenhang mit dem *OnCreate*-Ereignis des Hauptfensters ist dies beispielsweise nicht möglich, weil das Hauptfenster zu diesem Zeitpunkt noch nicht vollständig initialisiert ist. Zumindest im TreeDesigner darf dieses erste Kindfenster auch nicht automatisch von der VCL erzeugt werden, da diese nicht den neuen *Create*-Konstruktor von *TDocumentForm*, der in Kapitel 5.7.5 eingeführt wird, aufrufen würde.

Die wohl einfachste noch bleibende Möglichkeit ist, die Erzeugung eines Fensters durch *NewWindow* per Hand in die Projektdatei einzufügen, und zwar zwischen den letzten von Delphi automatisch erzeugten *CreateForm*-Aufruf und den ebenfalls automatisch erzeugten Aufruf von *Application.Run* (diese Projektdatei erhalten Sie am schnellsten über den Menüpunkt PROJEKT | QUELLTEXT ANZEIGEN):

```
{ Hauptprogramm der Projektdatei }
begin
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  ... Erzeugung der anderen Formulare ...
  Application.CreateForm(TOverviewForm, OverviewForm);
  MainForm.BeforeRun; { <<< Manuell eingefügt }
  Application.Run;
end.
```

Da der TreeDesigner neben dem Öffnen eines ersten Fensters auch noch andere Dinge zu diesem Zeitpunkt zu erledigen hat, wurde all dies in der Methode *BeforeRun* verpackt. Bei weiteren Änderungen braucht so statt des Projektdatei-Quelltextes nur noch diese Methode angepasst zu werden:

```
procedure TMainForm.BeforeRun;
begin
```

```
  { automatisch eine Datei laden, falls das Programm
    mit Befehlszeilenparametern aufgerufen wurde: }
  if (ParamStr(1)<>'') and (ParamStr(1)[1]<>'-')
  {$ifdef Windows} and (ParamStr(1)[1]<>'/')  {$endif}
  then
    NewWindow(AddDoc(ParamStr(1)))
  else
    NewWindow(AddDoc('')); // neues leeres Dokument
  // hier folgt noch die Wiederherstellung des Docking-Layouts,
  // siehe Kapitel 5.8.2
end;
```

Die *if*-Anweisung untersucht, ob der Anwendung ein Startparameter übergeben wurde (die Startparameter geben Sie in der Delphi-IDE unter START | PARAMETER... an) und ob dieser nicht durch ein vorangehendes »-« oder »/« als Option gekennzeichnet ist. Alles, was keine Option ist, wird als Dateiname interpretiert. Die obige Methode versucht dann, diese Datei mit *LoadFromFile* in das neue Fenster zu laden.

Hinweis: In der Linux-Version darf das Zeichen »/« nicht als Beginn einer Option gedeutet werden, da unter Linux Pfadangaben mit diesem Zeichen beginnen können. Der TreeDesigner hat eigentlich nicht die bewusste Absicht, überhaupt Befehlszeilenoptionen zu beachten, erbt aber als COM-Objekt-Server die Eigenschaft der VCL, auf die Standardparameter */RegServer* und */UnregServer* zu reagieren (siehe *Registrierungs-Formalitäten* in Kapitel 8.7.2). Werden diese tatsächlich angegeben, sorgt die obige Abfrage dafür, dass sie nicht als Dateiname mißinterpretiert werden.

Kindfenster schließen

Das Schließen der Kindfenster läuft so ab wie bei normalen Formularen. So fragt die VCL vor dem Schließen des Fensters mit dem *OnCloseQuery*-Event um Erlaubnis. Die hierfür in Kapitel 5.2.6 geschriebene Methode bleibt daher weiterhin gültig. Wenn die gesamte MDI-Anwendung geschlossen werden soll, ruft die VCL für jedes Kindfenster die *OnQueryClose*-Methode auf und schließt die Anwendung nur dann, wenn jede dieser Methoden die Schließung erlaubt.

Damit der Benutzer die Dokumentfenster auch schließen kann, muss darüber hinaus das *OnClose*-Event wie folgt bearbeitet werden:

```
procedure TDocumentForm.
   FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;
```

Ohne Zuweisung von *caFree* würde das Formular lediglich minimiert werden (zu *caFree* siehe auch in Kapitel 3.4.4 den Abschnitt *Ressourcen sparen durch geändertes Schließverhalten*).

Das Fenstermenü

Die einfachste aller Aufgaben ist das Implementieren des Fenstermenüs, das Sie bei Verwendung einer Aktionsliste und den mit Delphi mitgelieferten Standardaktionen theoretisch sogar ohne Zuhilfenahme der Tastatur funktionsfähig machen können. Um nicht den Bezug zur Realität zu verlieren, sei zur Einleitung trotzdem die Implementierung des Menüpunktes FENSTER | HORIZONTAL AUFGETEILT auf die herkömmliche Art gezeigt:

```
procedure TMainForm.HorizontalTile1Click(Sender: TObject);
begin
  TileMode := tbHorizontal;
  Tile;
end;
```

Entsprechend erreichen Sie ein NEBENEINANDER (VERTIKAL AUFGETEILT), wenn Sie vor dem Aufruf von *Tile TileMode* auf *tbVertical* setzen. Die Unterscheidung zwischen zwei verschiedenen *Tile*-Aufrufen stellt übrigens schon die Luxusversion der Anordnen-Funktion dar, denn manche MDI-Anwendungen bieten nur die Befehle *Nebeneinander*, *Übereinander* und *Symbole anordnen* an.

Für die Implementierung der Punkte ÜBEREINANDER und ICONS ANORDNEN müssten Sie nun maximal die Prozedur *Cascade* bzw. *ArrangeIcons* aufrufen. Wenn Sie jedoch Aktionslisten (*TActionList*-Komponenten, siehe Kapitel 4.6.4) verwenden, brauchen Sie für alle vier Menüpunkte keinen Code zu schreiben:

- Öffnen Sie den Aktionslisteneditor zu Ihrer *TActionList*-Komponente, rufen Sie aus dem lokalen Menü NEUE STANDARD-AKTION... auf und fügen Sie die vier Standardaktionen für MDI-Fenster ein. Danach genügt es bereits, die *Action*-Properties der vier Menüpunkte mit diesen Aktionen zu verknüpfen.

Was für ein komplettes Fenstermenü noch fehlt, ist die Liste der offenen Dokumente (bzw. Dokumentfenster), die üblicherweise hinter einer Trennungslinie an das bestehende Fenstermenü angehängt wird. Auch die Implementierung dieser Funktion besteht lediglich im Setzen eines Properties:

- Geben Sie im Property *WindowMenu* die Menüpunktkomponente an, an die die VCL die Liste der vorhandenen Kindfenster anhängen soll, bzw. wählen Sie einen solchen aus der vom Objektinspektor angezeigten Liste aus (im TreeDesigner hat diese Komponente beispielsweise den Namen *Fenster1*).

Hauptmenüpunkte im Kindfenster *R34*

Wenn ein Menüpunkt des Dokumentfensters einem globalen Befehl aus dem Menü des Hauptfensters entspricht, so sollte dieser auch vom Hauptfenster bearbeitet werden. So wurden in diesem Kapitel bereits die Methoden *MDIOpenClick* und *MDINewClick* für die Hauptformular-Menüpunkte DATEI | NEU und DATEI | ÖFFNEN des TreeDesigners gezeigt, die im Menü des Dokumentfensters wiederholt werden müssen – aus den erwähnten Gründen des Menüverschmelzens.

Eine nahe liegende Möglichkeit besteht darin, im Dokumentformular für den doppelten Menüpunkt einfach eine neue Bearbeitungsmethode zu erzeugen, in der Sie dann die entsprechende Methode des Hauptformulars aufrufen:

```
procedure TDocumentForm.CmGlobalOpenClick(Sender: TObject);
begin
  MainForm.MDIOpenClick(Sender);
end;
```

Man kann sich eine solche Methode jedoch auch sparen, indem man das *OnClick*-Ereignis des Dokumentfenster-Menüpunktes zur Entwurfszeit unverknüpft lässt und es statt dessen zur Laufzeit direkt mit der entsprechenden Methode des Hauptformulars verknüpft (z.B. beim *OnCreate*-Event). Im TreeDesigner wird der zweite der doppelten Menüpunkte (mit dem Namen *CmGlobalNew*) zur Abwechslung auf diese Weise bearbeitet, wobei die Methode *TMainForm.MDINewClick* bereits am Anfang des Kapitels gezeigt wurde:

```
procedure TDocumentForm.FormCreate(Sender: TObject);
begin
  CmGlobalNew.OnClick := MainForm.MDINewClick;
  ... weitere Initialisierung ...
```

Für beide Lösungsmöglichkeiten müssen Sie die Unit des Hauptformulars in die Unit des MDI-Kindfensters einbinden, um auf die globale Variable des MDI-Hauptfensters zugreifen zu können (im Beispiel die Variable *MainForm*). Da die Unit des Hauptformulars bereits die Unit des Dokumentformulars einbindet, können Sie die umgekehrte Einbindung nur innerhalb des Implementationsteils der Dokumentformular-Unit vornehmen (siehe Kapitel 2.1.7):

```
implementation
uses
  MdiForm;
```

Eine dritte Möglichkeit kommt ohne diese Unit-Einbindung aus: Setzen Sie den *OnClick*-Methodenzeiger des Dokumentformulars vom MDI-Hauptfenster aus, beispielsweise direkt nach der Erzeugung des Dokumentfensters:

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
```

```
  Window: TDocumentForm;
begin
  Window := TDocumentForm.Create(self, Document);
  Window.CmGlobalNew.OnClick := MDINewClick;
  ...
```

Hinweis: Die visuelle Formularverknüpfung (benötigt bei Datenmodulen, siehe Kapitel 7.2.1) ist leider nicht dazu geeignet, Ereignisse mit Methoden eines anderen Formulars zu verknüpfen, so dass Sie hier eine der beschriebenen nicht-visuellen Vorgehensweisen wählen müssen.

5.7.5 Dynamisches Werkzeugleisten-Management

Wie schon in Kapitel 5.7.1 unter *Zwei gegensätzliche Ziele* festgestellt, sollten Symbol- und Werkzeugleisten sich in einer MDI-Anwendung aus Gründen der Übersichtlichkeit auch dann im Hauptfenster befinden, wenn sie zur Manipulation des Dokuments dienen. Da die VCL die Symbolleisten des Dokumentfensters nicht automatisch im Hauptfenster einblendet wie die Menüs, werden wir diese Aufgabe hier selbst übernehmen.

Vorteile lokaler Werkzeugleisten

Ziel ist es also, die Leiste im Dokumentfenster entwerfen zu können und erst zur Laufzeit in das Hauptfenster zu verschieben. Für diese Vorgehensweise sprechen wichtige Vorteile:

- Sie erhält dem Dokumentformular die weitgehende Unabhängigkeit vom Hauptfenster, die es durch sein eigenes Hauptmenü gewonnen hat.
- Die Programmierung der Werkzeugleiste vereinfacht sich erheblich, denn die aufwändigeren, in Kapitel 5.7.6 beschriebenen Vorgänge entfallen, wenn die Leiste lokal ist.
- Da schließlich jedes Dokumentfenster seine eigene Kopie der Werkzeugleiste besitzt, bleiben alle darin vorgenommenen Einstellungen erhalten. Haben Sie beispielsweise ein Fenster mit dem Vergrößerungsfaktor 3 und eines mit dem Faktor 0,5, so können Sie beliebig zwischen beiden wechseln, und die Werkzeugleiste zeigt für jedes den richtigen Vergrößerungsfaktor an. Es scheint zwar so, als passe die Leiste die angezeigten Werte dem jeweils aktuellen Fenster an, tatsächlich handelt es sich jedoch um zwei verschiedene Leisten, die sich jeweils nicht verändern.

Wie der Abschnitt *Interner Ablauf der Fenster-Adoption* zeigen wird, verbrauchen mehrere Kopien einer Werkzeugleiste in mehreren Dokumentfenstern unter gewissen Umständen noch nicht einmal mehr Windows-Ressourcen als eine einzige globale Werkzeugleiste für alle Dokumentfenster.

Man könnte es höchstens als kleinen Nachteil bezeichnen, dass die Werkzeugleiste bei jedem Wechsel des Dokumentfensters neu aufgebaut werden muss. Solange die Grafikkarte dies jedoch schnell erledigt und das Programm ein Flackern größerer Teile des Bildschirms unterbindet, bekommt der Anwender eher den angenehmen Eindruck, dass das Programm besonders fleißig und kontextsensitiv ist.

Elternwechsel für die lokale Werkzeugleiste

Durch die Unterscheidung zwischen dem Besitzer einer Komponente und deren Elternfenster erzielt die VCL eine bemerkenswerte Flexibilität. Ausnahmen von der Regel »Besitzer=Elternfenster« wurden bereits in Kapitel 3 genannt (ein Formular besitzt auch seine »Enkel-Komponenten«, die einem seiner Gruppenfenster oder Panels als Kindfenster untergeordnet sind), und beim Docking von Fenstern gibt es weitere Ausnahmen (eine abgetrennte Symbolleiste bleibt im Besitz des Formulars, hat aber ein anderes Fenster als Elternkomponente erhalten).

Auch der folgende Fall ist denkbar: Komponenten, die sich im Besitz eines MDI-Kindfensters befinden, können zur Laufzeit das MDI-Hauptfenster als Elternfenster zugeordnet bekommen, ohne dadurch den Besitzer zu wechseln.

Die grundsätzliche Vorgehensweise sieht also wie folgt aus:

- Wenn ein Dokumentfenster aktiviert wird, benachrichtigt es das Hauptfenster darüber, dass es gerne eine oder mehrere Werkzeugleisten abgeben würde, und gibt dem Hauptfenster damit die Möglichkeit, diese Werkzeugleisten zu übernehmen. Dazu muss das Hauptfenster lediglich das *Parent*-Property der Werkzeugleisten auf sich selbst setzen und die Leisten eventuell noch sichtbar machen (damit dieser Vorgang vor dem Benutzer verborgen bleibt, sollten die Leisten nämlich im Dokumentfenster unsichtbar sein).
- Wenn das Fenster wieder deaktiviert wird, gibt es dem Hauptfenster erneut Bescheid, so dass dieses die Werkzeugleisten wieder an das Dokumentfenster zurückgeben kann.

Hinweis: Das Dokumentfenster könnte die Werkzeugleisten natürlich auch selbst in das Hauptfenster einblenden, die hier beschriebene Vorgehensweise entspricht aber bereits der noch zu zeigenden Vorgehensweise im TreeDesigner, bei der das Ziel verfolgt wurde, das Dokumentfenster möglichst unabhängig vom Hauptfenster zu halten.

Maßnahme für ein ruhiges Bild

Das alleinige Einblenden der Werkzeugleiste ist noch nicht als endgültige Lösung geeignet. Für den Anwender läuft das Einblenden nämlich nicht ganz so unsichtbar ab

wie für den Programmierer. Wenn der Anwender zwischen zwei Dokumentfenstern wechselt, wird zuerst die Werkzeugleiste des bisher aktiven Fensters entfernt. Angenommen, die Werkzeugleiste war zu diesem Zeitpunkt gerade unter der Menüzeile per *Align = alTop* befestigt. Durch das Entfernen der Leiste könnte sich dann der gesamte Arbeitsbereich des MDI-Hauptfensters mitsamt allen Dokumentfenstern nach oben in die von der Werkzeugleiste gelassene Lücke verschieben. Daraufhin wird die neue Werkzeugleiste eingefügt, was eine sofortige Zurückversetzung des verschobenen Bereichs an die bisherige Position zur Folge hätte. Je nach Systemgeschwindigkeit entstünde so ein störendes Flackern oder ein umständlich erscheinender Bildumbau.

Um das zu vermeiden, können Sie ein leeres Panel als Platzhalter in das Hauptfenster einfügen und die lokale Werkzeugleiste als Kindfenster in dieses Panel einblenden. Das Platzhalter-Panel braucht niemals entfernt zu werden und dient quasi als Stütze gegen ein kurzfristiges Einbrechen des Arbeitsbereichs in die von der Werkzeugleiste gelassene Lücke.

Hinweis: Im TreeDesigner werden die Symbolleisten zwar nicht in ein Panel, sondern in eine *TControlBar*-Komponente des Hauptfensters eingefügt, trotzdem verwendet auch das TreeDesigner-Hauptfenster vier »Stütz-Panel«, die nicht nur einen möglichst ruhigen Bildaufbau garantieren, sondern auch noch sicherstellen, dass die Symbolleisten der Dokumentfenster beliebig abgetrennt und angedockt werden können, ohne dass die Dokumentfenster etwas davon merken und ohne dass das Hauptfenster sich um die Symbolleisten des Dokumentfensters kümmern muss.

Abstimmung zwischen Dokument- und Hauptformular *R40*

Die schnelle Lösung wäre nun, das Platzhalter-Panel des Hauptformulars direkt aus dem Dokumentformular heraus anzusprechen, um die Werkzeugleiste einzufügen. Auf diese Weise würde jedoch wieder eine Abhängigkeit des Dokumentformulars vom Hauptformular entstehen. Diese kann aber leicht vermieden werden, zum Beispiel,

- indem das Hauptformular dem Dokumentformular sagt, welches Panel zur Leisten-Einblendung vorgesehen ist (es könnte z.B. eine Variable des Dokumentfensters auf dieses Panel setzen);
- oder indem das Hauptfenster dem Dokumentfenster eine Schnittstelle zur Verfügung stellt, über die das Dokumentfenster dem Hauptfenster mitteilen kann, welche Werkzeugleisten ein- oder auszublenden sind. Das Einblenden und Ausblenden könnte dann im Hauptfenster stattfinden.

Der TreeDesigner definiert die genannte Schnittstelle über ein Interface:

```
type
  IToolbarHost = interface
    function TBHInsertPart(W: TWinControl; Width: Integer;
                           Visible: Boolean): Boolean;
    procedure TBHRemovePart(W: TWinControl);
    procedure TBHBeginUpdate;
    procedure TBHEndUpdate;
    procedure SetVisible(Part: TWinControl; Value: Boolean);
    function GetVisible(Part: TWinControl): Boolean;
    property PartsVisible[W: TWinControl]: Boolean
               read GetVisible write SetVisible;
  end;
```

Ein Hauptfenster, das zur Aufnahme von Toolbars des TreeDesigner-Dokumentfensters bereit ist, muss eine solche Schnittstelle bereitstellen, dem Dokumentformular also eine Möglichkeit geben, die in diesem Interface deklarierten Methoden aufzurufen (zu Details über die Interfaces siehe Kapitel 2.7). Die Aufgabe der einzelnen Methoden wird später am praktischen Aufruf erläutert.

Übergabe der Schnittstelle

Die Klasse *TDocumentForm* des TreeDesigners verlangt, dass eine solche Schnittstelle als Parameter für den *Create*-Konstruktor angegeben wird. Dieser speichert die Schnittstelle in der *TDocumentForm*-Variablen *ToolbarHost*, über die *TDocumentForm* später die oben genannten vier Methoden aufrufen wird:

```
constructor TDocumentForm.Create(AOwner: TComponent;
                                 TBH: IToolbarHost);
begin
  ToolbarHost := TBH;
  inherited Create(AOwner);
end;
```

Für die Übergabe dieses zweiten *Create*-Parameters muss die Formularerzeugungs-Methode *TMainForm.NewWindow* aus Kapitel 5.7.4 angepasst werden:

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
  Window: TDocumentForm;
begin
  Window := TDocumentForm.Create(self, MainFormUtil, Document);
  ...
```

Dabei ist *MainFormUtil* die geforderte Schnittstelle, deren Herstellung wir uns später genauer ansehen werden; zunächst geht es nur darum, wie die Methoden dieser Schnittstelle vom Dokumentformular aufgerufen werden. Wichtig ist dabei, dass der Aufbau des Hauptfensters für das Dokumentformular überhaupt keine Rolle spielt.

Das Dokumentformular weiß noch nicht einmal, ob das Hauptfenster seine Werkzeugleisten in eine ControlBar oder in ein normales Panel einordnen wird.

Praktische Umsetzung

Unser Ziel ist es also, bei jeder Aktivierung oder Deaktivierung eines Dokumentfensters die oben gezeigte Schnittstelle zu verwenden, um die Symbolleisten ein- bzw. auszublenden. Dazu brauchen wir als Erstes ein passendes Ereignis. Die Ereignisse *OnActivate* und *OnDeactivate* scheinen vom ihrem Namen her bereits gut geeignet zu sein, bei näherer Betrachtung stellt sich jedoch heraus, dass sie nicht nur bei einem Wechsel zwischen den MDI-Fenstern auftreten, sondern auch, wenn der Eingabefokus an ein Eingabeelement einer Leiste wechselt, angenommen, dass sich diese Leiste bereits im Hauptfenster, also *außerhalb* des MDI-Kindfensters befindet. Würden wir die Leiste beim *OnDeactivate*-Ereignis entfernen, so würde sie paradoxerweise auch dann sofort verschwinden, wenn der Anwender mit der Maus auf eines ihrer Eingabeelemente klickt.

MdiActivate und MdiDeactivate R37

Wir benötigen also ein Ereignis, das nur auftritt, wenn der Benutzer von einem MDI-Kindfenster zu einem anderen MDI-Kindfenster wechselt. Während frühere TreeDesigner-Versionen noch ein spezielles Windows-Ereignis mit dem Namen *WM_MdiActivate* abgefangen haben, geht die aktuelle Version auf eine plattformunbhängige Weise vor: Das Hauptfenster erhält dabei die Aufgabe, zu überwachen, wann ein anderes MDI-Kind / Dokumentfenster aktiviert wird. Dafür definieren die Dokumentfenster ein neues Ereignis namens *OnActivateDocWin*. Dieses soll eintreten, wenn das Dokumentfenster ein *OnActivate*-Event empfängt. Im Prinzip ist also *OnActivateDocWin* lediglich die Weitergabe des *OnActivate*-Ereignisses vom Dokumentfenster an ein anderes Fenster (in diesem Fall das Hauptfenster):

```
procedure TDocumentForm.FormActivate(Sender: TObject);
begin
  if Assigned(FOnActivateDocWin) then
    FOnActivateDocWin(self);
end;
```

Das Hauptfenster bearbeitet das Ereignis mit der folgenden Methode:

```
procedure TMainForm.ChildWindowActivate(Sender: TObject);
var
  NewActiveDoc: TDocumentForm;
begin
  if (Sender is TDocumentForm) then begin
    NewActiveDoc := Sender as TDocumentForm;
    if NewActiveDoc <> LastActiveDoc then begin
      if LastActiveDoc <> nil then
```

```
          LastActiveDoc.MdiDeactivate;
        NewActiveDoc.MdiActivate;
        LastActiveDoc := NewActiveDoc;
      end;
    end;
  end;
```

Wenn das neu aktivierte Fenster ein anderes ist als das bisher aktive Fenster, wird die *MdiDeactivate*-Methode des bisher aktiven Fensters und die *MdiActivate*-Methode des neu aktivierten Fensters aufgerufen. Außerdem merkt sich die Methode in der Formularvariablen *LastActiveDoc*, welches Fenster zuletzt in *ChildWindowActivate* bearbeitet wurde.

Als Ergebnis der bisherigen Vorgänge können wir festhalten: Wenn der Benutzer zu einem anderen MDI-Dokumentfenster wechselt, können wir in der Methode *MdiDeactivate* die Leisten des alten Dokumentfensters aus dem Hauptfenster entfernen und in *MdiActivate* die des neuen in das Hauptfenster einsetzen. Die Verantwortlichkeit für den Aufruf von *MdiActivate/MdiDeactivate* liegt beim Hauptfenster, das allerdings umgekehrt von seinen Dokumentfenstern erwarten darf, über deren Aktivierung informiert zu werden.

Die Verknüpfung des Ereignisses *TDocumentForm.FOnActivateDocWin* mit *TMainForm.ChildWindowActivate* geschieht übrigens ausnahmsweise nicht über ein Property. Da die Bearbeitung diese Ereignisses so wichtig ist, wurde es als weiterer Parameter in den Konstruktor *TDocumentForm.Create* hinzugefügt. Dadurch wird dem oben bereits erweiterten Aufruf ein weiteres Element hinzugefügt:

```
function TMainForm.NewWindow(Document: TGraphicDoc): TDocumentForm;
var
  Window: TDocumentForm;
begin
  Window := TDocumentForm.Create(self, MainFormUtil,
                                 Document, ChildWindowActivate);
```

Die Symbolleisten des TreeDesigners

Der Aufruf der Methoden *MdiActivate* und *MdiDeactivate* ist also vorbereitet, was noch fehlt, ist deren Implementierung. Zunächst zu *MdiActivate*, die dafür zuständig ist, die Symbolleisten des Fensters in das Hauptformular einzublenden. Im Falle des TreeDesigners haben wir es gleich mit vier *TToolBar*-Symbolleisten zu tun, die in das Hauptfenster eingeblendet werden sollen. Mit Hilfe der bereits genannten *ToolbarHost*-Schnittstellenmethoden fügt das Dokumentformular diese wie folgt in das Hauptfenster ein:

```
procedure TDocumentForm.MdiActivate(var Param: TWMMdiActivate);
begin
  if Assigned(ToolbarHost) then
```

```
      if Param.ActiveWnd = Handle then begin
        ToolbarHost.TBHBeginUpdate;
        ToolbarHost.TBHInsertPart(ShapeTextToolbar, 180, True);
        ToolbarHost.TBHInsertPart(ZoomToolbar, 240, True);
        ToolbarHost.TBHInsertPart(FontToolbar, 283, True);
        ToolbarHost.TBHInsertPart(NTToolbar, 60, Assigned(Document)
                                  and Document.DrawEnhanced);
        ToolbarHost.TBHEndUpdate;
      end
    end;
  end;
```

Jede Symbolleiste wird über einen Aufruf von *TBHInsertPart* in das Hauptfenster eingefügt.

- Erster Parameter der Methode ist die einzufügende Symbolleiste, in diesem Fall immer eine *TToolbar*-Komponente.
- Im zweiten Parameter wird die notwendige Mindestbreite der Toolbar angegeben. Werden die Toolbars nämlich zur Entwurfszeit im Dokumentformular mit *alTop* ausgerichtet, so stellt das ihre Breite auf die Breite des Dokumentformulars ein. Um nicht alle Begrenzungen der ControlBar des Hauptfensters zu sprengen, muss die Breite daher auf das notwendige Minimum verringert werden, bevor die Leisten im Hauptfenster eingeblendet werden.
- Der dritte Parameter besagt, ob die Symbolleiste sichtbar sein soll. Dies ist bei den Leisten des TreeDesigners per Voreinstellung immer der Fall.

Alle Aufrufe von *TBHInsertPart* werden schließlich durch Aufrufe von *TBHBeginUpdate* und *TBHEndUpdate* eingeklammert, so dass sich das Hauptfenster (bzw. das Objekt, das sich hinter der Schnittstelle *ToolbarHost* verbirgt) nicht nach jedem einzelnen Einfügen aktualisieren muss, sondern erst nach dem *TBHEndUpdate*-»Signal« . Dies ist allerdings nur eine Idee für zukünftige Erweiterungen, denn in der derzeitigen Version tun *TBHBegin/EndUpdate* nichts, die Bildschirmdarstellung wird also bei jedem *TBHInsertPart*-Aufruf sofort aktualisiert.

Das Entfernen der Werkzeugleiste findet auf ähnliche Weise über die Methode *TBHRemovePart* statt:

```
procedure TDocumentForm.MdiDeactivate;
begin
  if Assigned(ToolbarHost) then begin
    ToolbarHost.TBHBeginUpdate;
    ToolbarHost.TBHRemovePart(ShapeTextToolbar);
    ToolbarHost.TBHRemovePart(ZoomToolbar);
    ToolbarHost.TBHRemovePart(FontToolbar);
    ToolbarHost.TBHEndUpdate;
  end;
end;
```

Hinweis: Wenn Sie Symbolleisten ein- und ausblenden wollen (entweder auf Veranlassung des Benutzers oder im Falle kontextsensitiver Symbolleisten automatisch), genügt es nicht, einfach die Toolbar selbst unsichtbar zu machen (*MeineToolbar.Visible:=False*), denn auch die Elternkomponente der Toolbar, das Platzhalter-Panel aus dem Hauptformular, sollte ja verborgen werden. Auch *MeineToolbar.Parent.Visible:=False* wäre noch keine ausreichende Maßnahme, denn wenn das Platzhalter-Panel von der Toolbar abgetrennt wird, bekommt es von der VCL ein Hilfsfenster übergeordnet, in dem es vom Benutzer frei über den Bildschirm bewegt werden kann. Um dem Dokumentformular die Verantwortlichkeit für diese Umstände abzunehmen, stellt die *IToolbarHost*-Schnittstelle die Methoden *SetVisible* und *GetVisible* bzw. das *PartsVisible*-Property zur Verfügung. Sie können eine Toolbar also etwa mit *ToolbarHost.PartsVisible[MeineToolbar] := False* verstecken.

Implementierung der IToolbarHost-Schnittstelle

Offen geblieben ist bisher noch die *IToolbarHost*-Implementierung, die im TreeDesigner in einer eigenen Klasse zu finden ist:

```
TMainFormUtil = class(TInterfacedObject, IToolbarHost)
public
  constructor Create;
  ... hier folgen die vier Methoden...
```

Da es auch nur ein globales Hauptfenster gibt, schadet es nicht, ein globales Objekt dieser Hilfsklasse im Initialisierungsteil der Hauptformular-Unit zu erzeugen:

```
initialization
  MainFormUtil := TMainFormUtil.Create;
```

MainFormUtil ist das Objekt, das oben bereits an den *Create*-Konstruktor der Dokumentformulare übergeben wurde (innerhalb dieser Dokumentformulare ist *MainFormUtil* allerdings nicht als Objekt, sondern nur als Schnittstellenvariable ansprechbar).

Aus Platzgründen können hier nur zwei der vier implementierten Methoden abgedruckt werden. Beim Einfügen einer Symbolleiste verwendet der TreeDesigner die vier zur Entwurfszeit erzeugten Platzhalter-Panels innerhalb der ControlBar. *TBHInsertPart* sucht den ersten freien Platzhalter und fügt die als Parameter erhaltene Komponente dort ein:

```
function TMainFormUtil.TBHInsertPart(W: TWinControl;
              PartWidth: Integer; Visible: Boolean): Boolean;
var
  Dest: TPanel;
begin
  Result := False;
  Dest := nil;
  with MainForm do // Freien Platzhalter suchen und in Dest speichern:
```

```
      if Panel1.ControlCount = 0 then Dest := Panel1
      else if Panel2.ControlCount = 0 then Dest := Panel2
      else if Panel3.ControlCount = 0 then Dest := Panel3;
    if Assigned(Dest) then begin // Wenn ein Platzhalter frei ist...
      // Wenn W unsichtbar sein soll, werden die Elternkomponenten
      // gleich zu Beginn unsichtbar gemacht, damit keine unnötigen
      // Aktualisierungen am Bildschirm erscheinen:
      if not Visible then begin
        if Dest.Floating then Dest.HostDockSite.Visible := False
          else Dest.Visible := False;
      end;
      Dest.Width := PartWidth;
      W.Parent := Dest;
      W.Visible := True; // W ist nun sichtbar, wenn auch die
                         // Elternkomponenten sichtbar sind.
      if W is TToolbar then
        Dest.Caption := (W as TToolBar).Caption;
      // Symbolleiste erst am Schluss sichtbar machen, falls erforderlich:
      if Visible then PartsVisible[W] := True;
      Result := True;
    end;
  end;
```

Für das Abtrennen der Symbolleisten ist es auch wichtig, dass das *Caption*-Property der Platzhalter-Panel auf den *Caption*-Text der Symbolleisten gesetzt wird, damit die VCL die frei verschiebbaren Symbolleisten mit einem aussagekräftigen Fenstertitel versieht.

Das Entfernen der Symbolleiste verläuft etwas einfacher:

```
procedure TMainFormUtil.TBHRemovePart(W: TWinControl);
var
  Dest: TPanel;
begin
  Dest := W.Parent as TPanel;
  with MainForm do
    if (Dest = Panel1) or (Dest = Panel2) or
       (Dest = Panel3) then
    begin
      Dest.Caption := '';
      W.Visible := False;
      W.Parent := W.Owner as TWinControl;
    end;
end;
```

Wichtig ist dabei, dass *W* unsichtbar gemacht wird, *bevor* es wieder ein Kindelement seines Besitzers (also des Dokumentfensters) wird. Und da das Platzhalter-Panel dann für einen Moment leer am Bildschirm erscheinen wird, sollte als Allererstes sein *Caption*-Property gelöscht werden, das dem Benutzer sonst als eine Art »Bildschirmmüll« in der Mitte der leeren *Panel*-Komponente erscheinen könnte.

Auf das Abdrucken der Implementierung der Methoden *SetVisible* und *GetVisible* soll hier verzichtet werden, da das *Verstecken und Anzeigen von Symbolleisten* bereits in Kapitel 5.2.2 behandelt wurde.

Interner Ablauf der Fensteradoption

Unter Windows selbst ist es nicht möglich, einem schon existierenden Fenster ein anderes Elternfenster zuzuweisen. Wenn Sie auf API-Ebene eine derart fundamentale Fenstereigenschaft wie das Elternfenster ändern wollen, müssen Sie das Fenster zuerst zerstören und dann komplett neu erzeugen. Umso erfreulicher ist es, dass die VCL diesen Vorgang für den Programmierer absolut transparent und automatisch erledigt und dass dabei die Daten der Formularkomponente nicht verändert werden.

Der interne Ablauf der *Parent*-Anpassung sieht kurz zusammengefasst wie folgt aus: Die Property-Zugriffsmethode *TControl.SetParent* entfernt die Komponente mit *Parent.RemoveControl(self)* aus ihrer bisherigen Elternkomponente und fügt sie mit *NewParent.InsertControl(self)* in die neue Elternkomponente ein. *RemoveControl* löscht das Windows-Fenster mit der Methode *DestroyHandle*. Diese zerstört das Windows-Fenster quasi heimlich, also ohne dass Ihr Programmcode das merkt. Wenn die Werkzeugleiste schließlich mit *Visible := True* sichtbar gemacht wird, erzeugt die VCL das zugehörige Windows-Fenster wieder.

Eine besondere Bedeutung beim Ausblenden der Werkzeugleiste hat die Reihenfolge, in der die Properties gesetzt werden:

```
Toolbar.Visible := False;
Toolbar.Parent := Dokumentfenster;
```

Die VCL gibt auch bei diesem *Parent*-Wechsel mit *DestroyHandle* die Werkzeugleiste zuerst einmal frei. Da die Leiste jedoch nicht mehr sichtbar ist, braucht sie das zugehörige Windows-Fenster nicht gleich wieder neu zu erzeugen, die Werkzeugleiste verbraucht dann also überhaupt keine Windows-Ressourcen. Wäre *Visible* erst nach der *Parent*-Änderung abgeschaltet worden, würde die VCL das Windows-Fenster zuerst neu erzeugen und dann beim Unsichtbarmachen des Fensters nur verstecken. Dieses Verstecken würde aber die belegten Windows-Ressourcen nicht freigeben. Mit der gewählten Reihenfolge ist also sichergestellt, dass nur die gerade sichtbare Werkzeugleiste Windows-Ressourcen belegt.

Andere Aktionen, die die Neuerstellung des Windows-Fensters erfordern, sind in dem Abschnitt *Veränderungen zur Laufzeit* in Kapitel 3.4.2 erwähnt, zum Sparen von Ressourcen mit *DestroyHandle* siehe auch Kapitel 3.4.4.

Zusammenfassung R39

Zum schnellen Nachbau der dynamischen Toolbars können Sie die folgenden Schritte ausführen, wobei der einzufügende Code teilweise bereits an einem Beispiel gezeigt wurde:

- Erstellen Sie ein beliebiges MDI-Hauptformular und ein MDI-Dokumentformular, und übernehmen Sie die Unit *TBHost* mit der Schnittstellendefinition *IToollbarHost* aus dem TreeDesigner, oder passen Sie diese Schnittstelle für Ihre eigenen Bedürfnisse an.
- Fügen Sie im Hauptformular eine *TControlBar*-Komponente als Fundament für die Symbolleisten ein. Bei Bedarf fügen Sie globale Toolbars, die immer in der ControlBar erscheinen sollen, direkt in diese ein; für jede einzublendende Toolbar eines MDI-Kindfensters fügen Sie nur ein Platzhalter-Panel ein.
- Entwerfen Sie in Ihrem Dokumentformular wie im Dokumentformular des TreeDesigners eine oder mehrere Toolbars, und schreiben Sie *MdiActivate*- und *MdiDeactivate*-Methoden, in denen Sie die Werkzeugleisten wie oben gezeigt über die *IToolbarHost*- oder Ihre eigene Schnittstelle einfügen und ausblenden (je nach dem Parameter *Param.ActiveWnd*).
- Um den Aufruf von *MdiActivate* und *MdiDeactivate* sicherzustellen, können Sie im MDI-Kindfenster ein Ereignis *OnActivateDocWindow* bereitstellen, das Sie im Hauptfenster bearbeiten, um *MdiActivate* und *MdiDeactivate* aufzurufen (nach dem Muster der gezeigten Methode *ChildWindowActivate*).
- Implementieren Sie die *IToolbarHost*-Schnittstelle in Ihrer Hauptformular-Unit und übergeben Sie die Schnittstellenvariable an jedes Ihrer Dokumentfenster (z.B. wie in der oben gezeigten Methode *TMainForm.NewWindow*). Zur Implementation der Schnittstelle müssen Sie den oben gezeigten Code wahrscheinlich kaum abändern.

Hinweis: Das Einblenden der Symbolleisten im TreeDesigner ist völlig unabhängig von der Drag&Dock-Funktionalität (siehe Kapitel 5.2.2). So war das dynamische Werkzeugleistenmanagement auch schon Bestandteil der TreeDesigner-Version 2.0, während die abtrennbaren Werkzeugleisten durch die Neuerungen der VCL in Delphi 4 erst in der TreeDesigner-Version 2.5 Einzug gefunden haben. Die Unabhängigkeit vom Docking wird dadurch erreicht, dass das Docking ganz auf der Basis der Platzhalter-Panels für die einzublendenden Leisten stattfindet. Ob und mit welcher Symbolleiste diese Panels ausgefüllt sind, spielt für die Docking-Mechanismen der VCL keine Rolle.

5.7.6 Globale Werkzeugleisten und Mauspaletten

Dieses Kapitel beschäftigt sich mit der Programmierung von zwei der globalen Leisten des TreeDesigner-Hauptfensters: der Farbpalette am unteren und der Werkzeugleiste am linken Rand. Anders als die globale Toolbar mit den Schaltern für DATEI | NEU und DATEI | ÖFFNEN enthalten diese beiden Leisten nämlich zahlreiche Elemente, die vom gerade geöffneten Dokumentfenster abhängig sind. Sie müssen in zwei Richtungen mit den Dokumentfenstern verknüpft werden:

- Das Dokumentformular (bzw. das aktive Dokumentfenster) muss die Komponenten der beiden Leisten ansprechen können, z.B. muss es beim Einzeichnen eines neuen Grafikelements wissen, welche Grafikform und welche Farben gerade eingestellt sind. Also muss es auf Elemente zugreifen, die dem Hauptformular gehören.
- Die Ereignisse der Komponenten in den Leisten müssen vom Hauptformular an das Dokumentformular weitergeleitet werden. Hier findet also der umgekehrte Zugriff vom Haupt- auf das Dokumentformular statt.

Auslagern von Werkzeugleiste und Farbpalette *R4 I*

Wir nehmen dazu wie bereits in Kapitel 5.7.2 an, dass wir Werkzeugleiste und Farbpalette bereits im Dokumentformular definiert haben und diese nun in das Hauptformular auslagern wollen. Das eigentliche Auslagern einer Werkzeugleiste in das Hauptformular ist einfach über die Zwischenablage möglich: Markieren Sie die Leiste, wählen Sie BEARBEITEN | AUSSCHNEIDEN und fügen Sie die Leiste dann im Hauptformular entsprechend wieder ein.

Hinweis: Bei diesem Kopiervorgang werden grundsätzlich alle Property-Einstellungen kopiert, jedoch mit den folgenden Ausnahmen: Zur Vermeidung doppelter Namen erhalten einige Komponenten unter Umständen Vorgabenamen wie *Panel1*; leere *Caption*-Properties von *TPanel*-Komponenten werden mit dem Namen des Panels gefüllt; und wenn Sie die Komponenten in ein anderes Formular kopieren, werden die Verknüpfungen der Ereignisse grundsätzlich nicht übertragen (wenn Sie auch die verknüpften Methoden kopieren wollen, können Sie dies dadurch erreichen, dass Sie die Komponenten nicht über die Zwischenablage kopieren, sondern mit DER OBJEKTABLAGE HINZUFÜGEN... eine Schablone davon anfertigen und diese im Zielformular einfügen.)

Nun muss wie erwähnt der Programmcode des Dokumentformulars auf einige Elemente der Leisten zugreifen. Ein Beispiel dafür ist die Methode *TDocumentForm.SetAttributes*, in der die Einstellungen der Leisten-Steuerelemente auf die Properties eines Grafikelements übertragen werden:

```
procedure TDocumentForm.SetAttributes(Obj: TGraphicElement);
{ Alle Einstellungen in den Symbolleisten auf ein Grafikobjekt anwenden. }
begin
  Obj.Pen.Color := ColorControl.PenColor;
  Obj.Brush.Color := ColorControl.BrushColor;
  Obj.Font.Color := ColorControl.FontColor;
  Obj.Font.Height := FontSize.Position;
  Obj.Font.Name := FontList.Text;
  // Der folgende Aufruf wird unten noch einmal geändert, wenn
  // der Trackbar PenWidthControl in ein Frame ausgelagert wird:
  Obj.Pen.Width := PenWidthControl.Position;
  ...
```

Hier werden in den ersten drei Zeilen die Properties der Farbpalette, in den folgenden beiden Zeilen zwei Einstellungen aus der Schriftattribut-Toolbar und in der letzten Zeile ein Property des TrackBars zur Einstellung der Linienbreite ausgelesen. Um die Einstellungen der Schriftattribute-Toolbar brauchen wir uns hier nicht zu kümmern, denn wie in Kapitel 5.7.5 beschrieben, wird diese Toolbar im TreeDesigner zwar in das Hauptformular eingeblendet, bleibt aber trotzdem im Besitz des Dokumentformulars. Der Zugriff auf die Komponenten dieser Toolbar – hier *FontSize* und *FontList* – ist also kein Problem.

Die Farbpalette und die Werkzeugleiste mit dem TrackBar zur Einstellung der Linienbreite jedoch sollen ganz in das Hauptformular verlagert werden. Dies würde normalerweise bedeuten, dass *ColorControl* und *PenWidthControl* nicht mehr wie oben aus dem Dokumentfenster angesprochen werden können, sie müssten in den obigen Zeilen mit *MainForm.ColorControl* und *MainForm.PenWidthControl* angesprochen werden. Es ist jedoch möglich, dass die Zugriffe auf *ColorControl* und *PenWidthControl nicht* geändert zu werden brauchen. Dazu werden *ColorControl* und *PenWidthControl* nach der Verlagerung erneut als Variablen des Dokumentformulars deklariert, diesmal aber manuell im *public*-Bereich und nicht in dem von Delphi verwalteten *published*-Bereich am Anfang der Deklaration:

```
TDocumentForm = class(TForm)
begin
   // Von Delphi verwalteter published-Bereich.
   ...
public
   // Eigene Deklarationen
  FDocument: TGraphicDoc;
  ColorControl: TColorPalette;
  PenWidthControl: TTrackBar;
  ...
```

Nun müssen diese Variablen lediglich noch auf die entsprechenden Komponenten des Hauptformulars gesetzt werden. Im TreeDesigner geschieht das sofort bei der Konstruktion eines neuen Formulars, also im Formularkonstruktor. Der schon in Kapitel 5.7.5 um zwei Parameter erweiterte Konstruktor erhält nun abermals zwei neue Parameter:

```
constructor TDocumentForm.Create(AOwner: TComponent; TBH: IToolbarHost;
   APalette: TColorPalette; ATrackBar: TTrackBar; ADocument: TGraphicDoc;
   AOnActivateDoc: TNotifyEvent);
begin
  // hier finden zunächst die Dokument-View-bezogenen
  ...         // Initialisierungen statt (siehe Kapitel 5.3.1)
  ToolbarHost := TBH;
  ColorControl := APalette;
  PenWidthControl := ATrackBar;
  FOnActivateDocWin := AOnActivateDoc;
  inherited Create(AOwner);
end;
```

Entsprechend muss auch der Aufruf des Konstruktors im Hauptformular angepasst werden. Die Initialisierung der Variablen *ColorControl* und *PenWidthControl* gleich zu Anfang und die Tatsache, dass das Hauptfenster immer länger existiert als seine Kindfenster, garantieren, dass das Dokumentformular jederzeit auf die beiden Variablen zugreifen kann, als handle es sich dabei um seine eigenen Steuerelemente.

Hinweis: In der kompletten TreeDesigner-Version ist *TDocumentForm.Create* um noch zwei weitere Parameter reicher. Der erste gibt dem Dokumentformular Zugriff auf eine *TMenuToolBar*-Komponente des Hauptformulars und ist nur notwendig, da der TreeDesigner den Standard-Menüverschmelzungsdienst der VCL zugunsten der Menüs im Toolbar-Stil verschmäht, wie in Kapitel 5.2.3 beschrieben. Der letzte Zusatz-Parameter ist für die Liste der zurückliegenden Dateien gedacht, die vom Dokumentformular erweitert wird, wenn der Benutzer eine Datei mit DATEI | SPEICHERN UNTER speichert.

Weiterleiten der Ereignisse zum Dokumentformular *R35*

Da der Objektinspektor die Ereignisse der Komponenten nur mit Methoden desselben Formulars verknüpfen kann, müssen im TreeDesigner einige Ereignisse der globalen Werkzeugleisten manuell an das Dokumentformular weitergegeben werden. Die einzige kleine Schwierigkeit dabei ist die Tatsache, dass es zur Laufzeit sehr viele Dokumentfenster geben kann und normalerweise nur eines davon das Ereignis bearbeiten soll.

Doch lässt sich das aktive Dokumentfenster leicht über das Property *TForm.ActiveChild* herausfinden. Da wir im Gegensatz zum Compiler ja wissen, dass alle MDI-Kinder des TreeDesigners immer den Typ *TDocumentForm* haben, können wir *ActiveChild* außerdem noch ohne vorherige Sicherheitsabfrage mit dem *as*-Operator in ein *TDocumentForm* umwandeln, so dass sich anschließend alle Methoden des aktiven Dokumentformulars direkt aufrufen lassen. Das folgende Beispiel zeigt die Weitergabe des *OnBrushColorChange*-Ereignisses der Farbpalette:

```
procedure TMainForm.PaletteBrushColorChange(Sender: TObject;
  NewColor: TColor);
begin
  StylePreview.Invalidate; // Vorschau (Werkzeugleiste unten) neu zeichnen
  if ActiveMdiChild <> nil then
    with ActiveMdiChild as TDocumentForm do
      BrushColorChange(Sender, NewColor);
end;
```

Hinweis: Wenn Ihre Anwendung über mehrere Typen von MDI-Kindfenstern verfügt, können Sie die Klasse des aktiven Fensters über den *is*-Operator überprüfen.

Wenn mehrere ähnliche Ereignisse an das Dokumentformular weitergeleitet werden müssen, brauchen Sie nicht für jedes eine eigene Methode zu schreiben. Mehrere Schalter können beispielsweise durch den Wert des *Tag*-Properties unterschieden und von einer einzigen Methode bearbeitet werden:

```
procedure TMainForm.ChildCommandClick(Sender: TObject);
begin
  if ActiveMdiChild <> nil then
    with ActiveMdiChild as TDocumentForm do
      case (Sender as TSpeedButton).Tag of
        1: cmArrangeTreeClick(Sender);
      { 2: ...  (in der TreeDesigner-Werkzeugleiste befindet
                sich nur ein Aktionsschalter) }
      end;
end;
```

Verwendung der Frame-Technik zur Mehrfachnutzung einer Leiste *R27*

Nehmen wir nun einmal an, dass auf diese Weise sehr viele Komponenten von einem Dokumentformular in ein Hauptformular verschoben werden sollen. Bei der oben beschriebenen Vorgehensweise müsste jede verschobene Komponente einzeln wieder mit ihrem Ursprungsformular verknüpft werden.

Oder nehmen wir an, dass zwei Versionen der Anwendung parallel gepflegt werden sollen – z.B. beim TreeDesigner sowohl die MDI- als auch die SDI-Version. Wenn jetzt etwa die Werkzeugleiste im Hauptformular der MDI-Version geändert wird, muss

auch das Dokumentformular der SDI-Version, in dem ja eine Kopie der Leiste enthalten ist, entsprechend mitverändert werden und umgekehrt.

Um diese sehr umständlichen Wartungsarbeiten zu vermeiden, bietet sich die in Kapitel 3.7.2 detailliert erläuterte Frame-Technik an. Im Falle des TreeDesigners wird sie wie folgt verwendet: Die Werkzeugleiste wird weder im Dokument- noch im Hauptformular definiert, sondern in einem separaten Frame namens *GlobalToolbar*. Zu diesem Frame gehören auch einige Methoden, die in der zugehörigen Unit *FrameGlobalToolbar* definiert sind, z.B. die Methode, die im unteren Bereich der Toolbar einen kleinen Kreis als Vorschau für die gerade eingestellten Farben und die Stiftbreite zeichnet (siehe z.B. Abbildung 5.1).

Das *GlobalToolbar*-Frame wurde so gestaltet, dass es sich wie eine Komponente verwenden lässt. Dazu gehört auch, dass es Properties und Events zur Verfügung stellt:

```
type
  TGlobalToolbar = class(TFrame)
    ...
  public
    property CurShapeType: TShapeType read FCurShapeType
                                      write SetCurShapeType;
    property OnShapeTypeChange: TNotifyEvent read ...;
    property OnPenWidthChange: TNotifyEvent read ...;
    property OnArrangeTreeClick: TNotifyEvent read ...
    property ColorPalette: TColorPalette read FColorPalette
  end;
```

In *CurShapeType* speichert das Frame, welche Grafikform gerade gewählt ist. Das Formular, das diesen Frame verwendet, braucht so nicht die einzelnen Schalter des Frames abzufragen. Mit *OnShapeTypeChange* und *OnPenWidthChange* sowie *OnArrangeTreeClick* kann es seine Methoden verknüpfen, die auf das Ändern der Grafikform oder der Stiftbreite und auf den Schalter zum Anordnen des Baumes reagieren. Schließlich erwartet *GlobalToolbar*, dass das *ColorPalette*-Property auf die Farbpalette gesetzt wird, deren Farben im Vorschaubereich berücksichtigt werden sollen.

All diese Properties richtig zu setzen, sei dem vollständigen Code auf der CD überlassen. An dieser Stelle soll nur kurz skizziert werden, wie der Frame im TreeDesigner »eingebaut« wird.

Zunächst zur MDI-Version: In dieser wird ein Exemplar des Frames in das Hauptformular eingezeichnet und jedes Dokumentformular erhält eine Referenz darauf. Wenn Sie das Kapitel 5 bisher vollständig gelesen haben, wird Ihnen der Konstruktor von *TDocumentForm* ja schon als ständige Baustelle bekannt sein. Auch das Werkzeugleisten-Frame wird ihm als Parameter übergeben. Da die Werkzeugleiste jedoch bereits den TrackBar für die Stiftbreite enthält, kann der in *R41* (Seite 744) dafür vorgesehene Parameter *ATrackBar* wieder weggelassen werden. Es ergibt sich dann der folgende Konstruktor:

```
constructor TDocumentForm.Create(AOwner: TComponent;
   GTB: TGlobalToolbar; TBH: IToolbarHost; APalette: TColorPalette;
   ADocument: TGraphicDoc; AOnActivateDoc: TNotifyEvent);
begin
  ToolbarHost := TBH;
  GlobalToolbar := GTB;
  ...
```

Im restlichen Code des Dokumentformulars kann nun über die *GlobalToolbar*-Variable auf die einzelnen Komponenten und Properties des Frames zugegriffen werden. Z.B. ändert sich die Einstellung der Stiftbreite in der oben schon einmal gezeigten Methode *SetAttributes* wie folgt:

```
procedure TDocumentForm.SetAttributes(Obj: TGraphicElement);
begin
  if Assigned(GlobalToolbar) then
    Obj.Pen.Width := GlobalToolbar.PenWidth.Position
```

Die linksseitige Werkzeugleiste des TreeDesigners war in früheren Versionen einmal fester Bestandteil des Hauptformulars und wurde erst in der Version 3.5 als Frame ausgelagert. Eine solche Umstellung bringt folgende Vorteile:

- Für alle im Dokumentformular genutzten Elemente der Toolbar muss am Anfang nur eine Referenz auf den Frame an das Formular übergeben werden – im Falle des TreeDesigner an dessen Konstruktor.
- Da der Frame selbst einige Methoden enthält, verringert sich dadurch die Größe der Hauptformular-Unit (in diesem Fall immerhin um 100 von ca. 1500 Zeilen). Die vorher über die Hauptformular-Unit verstreute Funktionalität des Frames ist nun erheblich besser gekapselt.
- Die SDI-Version der Anwendung kann mit minimalem Aufwand aus der MDI-Version erzeugt werden.

5.7.7 Verwenden des MDI-Kindfensters als SDI-Hauptformular

Das Dokumentformular des TreeDesigners erreicht eine hohe Unabhängigkeit vom Hauptfenster, indem es keine Variablen des Hauptformulars direkt anspricht, sondern nur indirekt über eigene Variablen, die im Konstruktor gesetzt werden. Die an zwei Stellen zu findende Abhängigkeit vom MDI-Hauptformular wurde nur zur Demonstration der verschiedenen Möglichkeiten im Abschnitt *Hauptmenüpunkte im Kindfenster* auf Seite 732 eingegangen. Beide Abhängigkeiten ließen sich (wie ebenfalls im genannten Abschnitt beschrieben) leicht auflösen.

Die Unabhängigkeit geht sogar so weit, dass das Dokumentformular sich selbstständig machen und als Hauptfenster in einer eigenen SDI-Anwendung arbeiten kann. Auf der CD finden Sie im Projekt *TreeDesignerSDI* eine SDI-Version des TreeDesigners, die

sich von der MDI-Version hauptsächlich dadurch unterscheidet, dass kein Exemplar des MDI-Hauptformulars konstruiert wird. Als Hauptformular dient statt dessen ein Exemplar der Klasse *TDocumentForm* aus der unveränderten Dokumentformular-Unit der MDI-Version. Wie jedes Hauptformular wird auch das Dokumentformular nun in der Projektdatei erzeugt, und zwar durch folgenden Aufruf:

```
Application.CreateForm(TDocumentForm, DocumentForm);
```

Dabei wird der normale *Create*-Konstruktor, den *TDocumentForm* von *TForm* erbt, aufgerufen, nicht jedoch der oben beschriebene, ständig um neue Parameter erweiterte Konstruktor (was ja auch kein Wunder ist, denn wie sollte die VCL diese ihr unbekannten Parameter setzen?).

TDocumentForm muss also auch funktionieren, wenn es mit dem normalen *Create*-Konstruktor aufgerufen wird. Dazu müssen alle Variablen, die in der MDI-Version von außen an den erweiterten *Create*-Konstruktor übergeben werden, nun vom Dokumentformular selbst initialisiert werden, indem die fehlenden Komponenten etwa dynamisch selbst konstruiert werden. *TDocumentForm* erledigt dies bei seinem *OnCreate*-Ereignis.

Da ja dieselbe Formularversion in MDI- und SDI-Version verwendet werden soll, müssen nun in *FormCreate* beide Fälle berücksichtigt werden. Sie werden zur Laufzeit durch die boolesche Variable *FIsMDIForm* unterschieden, die nur im *Create*-Konstruktor der MDI-Version auf *True* gesetzt wird (dazu muss der zuletzt in Kapitel 5.7.6 erweiterte Code des Konstruktors erneut um eine Zuweisung ergänzt werden, was Sie sich aber sicher auch ohne ein erneutes Listing vorstellen können). In der SDI-Version ist *FIsMDIForm* immer *False*, da alle Objektvariablen von der VCL automatisch mit 0 bzw. *False* initialisiert werden.

Aus Platzgründen kann nur ein Ausschnitt aus *FormCreate* gedruckt werden, was jedoch zum Verständnis des Prinzips ausreichen sollte. Außer dem unbedingt notwendigen *GlobalToolbar*-Frame und der ebenfalls wichtigen Farbpalette wird auch eine *TControlBar* für die flexible Aufbewahrung der Toolbars erzeugt (diese können ja nun nicht mehr in die ControlBar eines MDI-Hauptfensters verlagert werden):

```
procedure TDocumentForm.FormCreate(Sender: TObject);
// Methode für das Ereignis TDocumentForm.OnCreate
var
  FontIndex: Integer;
  CB: TControlBar;
begin
  if FIsMDIVersion then begin
    CmGlobalNew.OnClick := MainForm.MDINewClick; // (bekannt aus K.5.7.4)
  end else begin // SDI-Version!
    // Einrichten des Formulars für die SDI-Version:
    Menu := MDIChildMenu; // Menü als Hauptmenü anzeigen
    GlobalToolbar := TGlobalToolbar.Create(self);
```

```
      GlobalToolbar.Parent := self;
      CB := TControlBar.Create(self);
      with CB do begin
        Parent := self;
        Align := alTop;
        AutoSize := True;
        Height := ShapeTextToolbar.Height * 2;
      end;
      ShapeTextToolbar.Parent := CB;
      ... (Einfügen der Toolbars in die Controlbar)
      ColorControl := TColorPalette.Create(self);
      ... (Erzeugen einer Farbpalette)
      // Verknüpfen der Frame-Events mit eigenen Methoden:
      with GlobalToolbar do begin
        OnShapeTypeChange := self.SpeedBtnRectangleDblClick;
        OnPenWidthChange := self.PenWidthChange; // Hier ist "self." wichtig
        OnArrangeTreeClick := self.cmArrangeTreeClick;
      end;
    end;
    ... (weitere Initialisierungen für beide Fälle)
  end;
```

Hinweise: Auch das *FormStyle*-Property muss natürlich in SDI- und MDI-Version verschieden gesetzt werden. Im TreeDesigner-Dokumentformular wurde dieses Property zur Entwurfszeit bereits mit *fsNormal* initialisiert. Der bereits mehrfach erwähnte spezielle *Create*-Konstruktor, der nur in der MDI-Version aufgerufen wird, setzt *FormStyle* dann auf *fsMdiChild*.

Der Spezial-Code zur Initialisierung des SDI-Formulars ist noch erweiterungsfähig. Die aktuelle SDI-Version des TreeDesigners enthält beispielsweise noch keine Docking-Fähigkeiten für Baumansichts- und Übersichtsfenster und keine Statuszeile. Beides ist im MDI-Hauptfenster definiert und müsste noch gesondert in das Dokumentformular übertragen werden.

5.8 Erweiterung der Benutzerschnittstelle

In diesem Kapitel geht es um einige weitere Themen, die mit der Benutzerschnittstelle einer Anwendung zusammenhängen:

- Tastatureingaben in eine Komponente, die von sich aus noch keine Tastatureingabe bearbeitet, um die automatische Weiterleitung des Tastaturfokus und speziell um die Handhabung des Tastaturfokus im TreeDesigner (Kapitel 5.8.1);
- Andocken von Fenstern (Kapitel 5.8.2);

- Positionsabhängige Popup-Menüs für die TreeDesigner-Zeichenfläche (Kapitel 5.8.3).
- Drag&Drop (Kapitel 5.8.3).

5.8.1 Tastatursteuerung

Da die meisten Tastatureingaben bereits von Menüs und Steuerelementen verarbeitet werden, ist es in den meisten Formularen nicht notwendig, die Tastaturereignisse selbst zu bearbeiten. Auch der TreeDesigner bearbeitet nur zwei verschiedene Tasten zur Veränderung der Grafikobjekte: Entf und PgDn. Dieses Kapitel zeigt Besonderheiten, die bei dieser Bearbeitung auftreten können, und gibt ein Beispiel für die gezielte Kontrolle des Tastaturfokus. Grundlegende Informationen zu den drei Tastatureingabe-Ereignissen, zu den beiden Fokusereignissen *OnEnter* und *OnExit* und zur Tastatursteuerung im Allgemeinen finden Sie in Kapitel 3.3.2.

Tastatureingaben

Ein Grafikprogramm kommt normalerweise nicht mit den Mausziehoperationen aus Kapitel 5.4 aus, sondern macht auch einige Funktionen über die Tastatur zugänglich, ohne dass dabei Menüs und Schalter eingesetzt werden. Im TreeDesigner sind das die folgenden Funktionen:

- Mit Entf löschen Sie alle markierten Grafikobjekte.
- Mit PgDn verschieben Sie das oberste Grafikelement hinter alle anderen Elemente und können so Elemente sichtbar machen, die vorher verdeckt waren.

Da beide Tasten Sondertasten sind und keinem Zeichen des ANSI-Zeichensatzes entsprechen, können sie nicht im Ereignis *OnKeyPress* bearbeitet werden, sondern sind nur über das Ereignis *OnKeyDown* erreichbar. Die abzufragenden Tastencodes sind *VK_DELETE* für Entf und *VK_NEXT* für PgDn.

Herkunft der Tastaturereignisse im TreeDesigner

Es stellt sich nun die Frage, wo sich der Tastaturfokus befindet, wenn der Benutzer im TreeDesigner beispielsweise Entf drückt, um ein Grafikelement zu löschen. Wir nehmen dazu den schwierigsten Fall für die Tastaturfokus-Verwaltung an: wenn die Werkzeugleiste im Dokumentfenster integriert ist (wie in der SDI-Version des TreeDesigners) und deren Eingabeelemente den Fokus an sich ziehen. Wenn der Benutzer nun gerade Text im Beschriftungsfeld der Leiste eingegeben hat, so hat dieses Eingabefeld noch den Fokus.

Will der Benutzer nun das aktuelle Grafikelement mit Entf löschen, wird er versuchen – wenn er mit der Bedienung grafischer Benutzeroberflächen vertraut ist –, die Zeichenfläche zu fokussieren, indem er beispielsweise mit der Maus auf sie klickt. Da die Paintbox als *TGraphicControl*-Komponente den Fokus jedoch nicht erhalten kann, kann der Fokus nur an die übergeordnete Scrollbox gehen (siehe auch *Der Fokus für die Scrollbox* auf Seite 753).

Für die beiden oben beschriebenen Tastatureingaben müssten also die Tastaturereignisse von *TScrollBox* behandelt werden. Bedauerlicherweise sind diese jedoch noch als *protected* deklariert und demnach weder im Objektinspektor noch zur Laufzeit ansprechbar.

Die Bergung verlorener Ereignisse

Das heißt also, dass die Tastaturereignisse gewissermaßen in den Tiefen von *TScrollBox* verloren gehen. Der TreeDesigner ist nur eines von vielen vorstellbaren Beispielen, in denen Tastatureingaben auch in einer Scrollbox auftreten können.

Um an solche verloren gegangene Ereignisse zu gelangen, können Sie zwischen den folgenden beiden Möglichkeiten wählen:

- Sie können das *KeyPreview*-Property des Formulars auf *True* einstellen, wodurch die Tastaturereignisse *aller* im Formular befindlichen Komponenten an dieses gesendet werden (in Form der üblichen *OnKey...*-Ereignisse). Sie müssten dann auch abfragen, welche Komponente gerade den Tastaturfokus hat (Property *ActiveControl*).
- Sie packen das Hindernis bei der Wurzel und bringen die Tastaturereignisse von *TScrollBox* an die Öffentlichkeit, und zwar durch Ableitung einer neuen Komponentenklasse von *TScrollBox*, bzw. Sie verwenden die Klasse *TScrollBoxEx* aus Kapitel 6.5.2.

Der TreeDesigner verwendet daher statt *TScrollBox* die erweiterte Klasse *TScrollBoxEx* und bearbeitet deren *OnKey...*-Ereignisse. Für die beiden erwähnten Tasten genügt bereits die Bearbeitung von *OnKeyDown*. Die mit den Tasten aufzurufenden Funktionen werden von eigenen Methoden bereitgestellt: *DeleteCommand* (Entf) und *ShiftOrderCommand* (PgDn), so dass beide Befehle leicht auch über *OnClick*-Methoden, beispielsweise des Hauptmenüs, aufgerufen werden können:

```
procedure TDocumentForm.ScrollboxKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if key = VK_DELETE then
    DeleteCommand
  else if key = VK_NEXT then
    ShiftOrderCommand
end;
```

```
procedure TDocumentForm.DeleteCommand;
begin
  Document.DeleteSelected; { alle gewählten Objekte löschen }
  MouseObject := nil;  { kein Objekt ist mehr markiert }
end;

procedure TDocumentForm.ShiftOrderCommand;
begin
  if Document.Count > 1 then begin
    Document.DeselectAll; { die Markierung aller Objekte aufheben }
    Document.LastVisibleObjectToFront;
    MouseObject := Document.Items[Document.Count-1];
    MouseObject.Marked := True;
    // Markiertes Objekt in den sichtbaren Bereich rücken
    (* Im vollständigen Code wird außerdem sichergestellt, dass das
       markierte Objekt sichtbar ist (u.a. mit ScrollPointToCenter. *)
  end;
end;
```

Der Fokus für die Scrollbox *R71*

Das Bearbeiten der *OnKeyDown*-Ereignisse alleine genügt hier jedoch noch nicht, denn die Scrollbox gehört zu den Komponenten, die nicht automatisch den Tastaturfokus erhalten, wenn sie angeklickt werden. Der Benutzer könnte den Fokus also nur im Rahmen der üblichen Dialogsteuerung mit den Pfeiltasten oder mit Tab (falls das Property *TabStop* der Scrollbox *True* ist) zur Scrollbox bringen.

Normalerweise ist es angemessen, dass eine Scrollbox sich bei Mausklicks nicht den Fokus reserviert, denn oft kann sie gar nichts mit den Tastatureingaben anfangen. So ist es auch zu erklären, dass *TScrollBox* die Tastaturereignisse nicht veröffentlicht.

Um eine von *TWinControl* abgeleitete Komponente wie *TScrollBox* zu einem vollwertigen Eingabeelement zu machen, das den Tastaturfokus auch durch einen Mausklick erhalten kann, müssen Sie den Fokus beim Ereignis *OnMouseDown* selbst auf dieses Element setzen. Im TreeDesigner wird der Mausklick auf die Fläche der Scrollbox jedoch schon durch die darüber liegende Paintbox abgefangen. Daher besteht die erste Aktion der vollständigen Methode *PaintBoxMouseDown* im Setzen des Tastaturfokus auf die ScrollBox (die PaintBox selbst kann den Fokus ja nicht bekommen, da sie kein *TWinControl* ist):

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  { funktioniert in diesem Fall nicht verlässlich: Scrollbox.SetFocus }
  Windows.SetFocus(Scrollbox.Handle);
  ...
```

Die VCL-Methode *SetFocus* greift intern auf die Windows-API-Funktion *SetFocus* zurück; jedoch geht sie etwas knausrig mit dem Aufruf der API-Funktion um: Sie versucht, den Aufruf einzusparen, wenn sie meint, dass er nicht nötig sei. Sollte dies einmal nicht richtig funktionieren, können Sie die API-Funktion wie im obigen Beispiel auch direkt aufrufen.

Tastatursteuerung für die Scrollbox *R76*

Wenn eine ScrollBox bereits den Tastaturfokus hat, bietet es sich an, die Ereignisse der Cursortasten zu bearbeiten, damit der Benutzer auch mit der Tastatur scrollen kann. Auch hierfür bedarf es einer Ergänzung der Klasse *TScrollBox*, denn die VCL reserviert einige Pfeiltasten für die Tastaturfokus-Steuerung, so dass diese in den *OnKeyDown*-Methoden standardmäßig unsichtbar sind. Wenn eine Komponente derartige Tasten selbst bearbeiten oder in ihren *OnKey...*-Ereignissen zur Verfügung stellen will, muss sie die VCL-Nachricht *CM_WantSpecialKey* abfangen. *TScrollBoxEx* tut dies und macht diese Nachricht zum Event *OnWantSpecialKey*, das Sie bei Benutzung der Komponente mit einer Methode verknüpfen können. Der hierfür notwendige zusätzliche *TScrollBoxEx*-Quelltext befindet sich auf der CD-ROM. Im Testprogramm für *TScrollBoxEx*, zu finden im Verzeichnis COMPLIB unter dem Namen SBTEST, wird auch die Verwendung von *OnWantSpecialKey* demonstriert. Das Programm verwendet eine ScrollBoxEx, die Sie mit der Tabulatortaste ansteuern und mit den Cursortasten scrollen können.

Hinweis: Auch der TreeDesigner verwendet das *OnWantSpecialKey*-Ereignis von *TScrollBoxEx*, und zwar, um die Taste Pfeil runter abzufangen (*VK_DOWN*) und diese dem Benutzer als Alternative zu PgDn zur Verfügung zu stellen.

Automatisches Weiterleiten von Fokus und Tastatureingabe *R75*

Normalerweise steuert der Anwender den Tastaturfokus selbst durch die verschiedenen Steuerelemente, im einfachsten und möglicherweise langsamsten Fall einfach durch Anklicken der Elemente.

Manchmal weiß er es jedoch auch zu schätzen, wenn der Fokus automatisch zu der passenden Stelle springt. Im TreeDesigner ergibt sich eine gute Gelegenheit zu einer solchen Automatikfunktion: Der Benutzer kann nach dem Einzeichnen eines Grafikelements sofort den Beschriftungstext eingeben, obwohl der Tastaturfokus durch das Einzeichnen des Elements zur Scrollbox wandert.

Die Eingabe der Beschriftung ist sehr leicht von anderen Eingaben zu unterscheiden: Da in der Scrollbox nur die Tasten Entf und PgDn verwendet werden, kann sich der TreeDesigner denken, dass jede andere Taste einem anderen Zweck dienen soll. Bei

jeder Eingabe eines einfachen Zeichens in die Scrollbox, also beim Ereignis *OnKeyPress*, leitet der TreeDesigner den Fokus und das schon eingegebene Zeichen daher an das Eingabefeld für die Beschriftung (*ShapeText*) weiter:

```
procedure TDocumentForm.
  ScrollBoxKeyPress(Sender: TObject; var Key: Char);
begin
  ShapeText.SetFocus;
  ShapeText.Perform(WM_CHAR, Byte(key), 0); { Eingabe weiterleiten }
end;
```

Der Aufruf von *Perform* simuliert die Eingabe eines Zeichens in *ShapeText* durch die Original-Windows-Botschaft *WM_CHAR*, die auch bei jeder normalen Eingabe eines Zeichens auftritt. (Durch diese *WM_CHAR*-Nachricht wird also auch ein *OnKeyPress*-Ereignis von *ShapeText* erzeugt.)

Hinweis: Hinweis zu *Perform* in der Nachrichtenverarbeitung (siehe Kapitel 3.1.4 und Abbildung 3.2 auf Seite 324): *Perform* ruft direkt die *WndProc*-Methode von *ShapeText* auf, macht also keinen Umweg über eine Nachrichtenversendungsfunktion von Windows (wie etwa *SendMessage* oder *PostMessage*). Windows erfährt dennoch früh genug von der Nachricht: Wie jede andere Nachricht auch gelangt die selbst erzeugte *WM_CHAR*-Botschaft am Ende der Nachrichtenleitung zur Windows-Funktion *DefWindowProc*.

AutoSelect

Ein wichtiges Detail bei der Tastatursteuerung ist auch, dass der Inhalt des Editierfelds vollständig markiert wird, sobald das Feld den Fokus erhält. So ist es möglich, den gesamten Inhalt durch Eingabe eines neuen Zeichens zu überschreiben. In *TEdit*-Komponenten können Sie dieses Verhalten durch das Property *AutoSelect* abschalten, in *TCombobox* gibt es dieses Property nicht, der Feldinhalt wird dort immer automatisch selektiert.

KeyPreview *R79*

Abschließend sei noch auf ein Beispiel zur Verwendung des am Anfang erwähnten *KeyPreview*-Events verwiesen. Das Beispielprogramm *NBDemo1* aus den Kapiteln 3.5.4 und 3.5.6 verwendet ein TabbedNotebook zur Darstellung mehrerer Dialogseiten. Im Gegensatz zur Komponente *TPageControl* blättert diese nicht automatisch um, wenn der Benutzer Strg+Tab oder Shift+Strg+Tab drückt. Diese Umblätterfunktion wurde in *NBDemo1* dadurch realisiert, dass das *KeyPreview*-Property des Formulars auf *True* gesetzt wurde, wodurch das Formular für alle Tastatureingaben in seine Komponenten Tastaturereignisse erhält. Die Methode für *OnKeyDown* braucht so nur die entsprechenden Tasten abzufangen und die Seiten umzublättern:

```
if Key = VK_TAB then begin
  if [ssCtrl, ssShift] <= Shift then
    NewPageIndex := Notebook.PageIndex-1
  else if ssCtrl in Shift then
    NewPageIndex := Notebook.PageIndex+1;
```

5.8.2 Docking von Fenstern

Nachdem das Docking von Symbolleisten bereits Thema von Kapitel 5.2.2 war, befasst sich dieses Kapitel mit dem Docking von Fenstern. Die im Zusammenhang mit den Symbolleisten dargestellten Grundlagen der Docking-Properties und -Ereignisse gelten zunächst auch für das Fenster-Docking. Abbildung 5.12 verdeutlicht noch einmal, in welcher Reihenfolge und für welche Komponenten die Ereignisse auftreten.

Das Beispiel des TreeDesigners wird uns nun auch die Gelegenheit geben, die Docking-Ereignisse *OnDockDrop* und *OnUndock* zu verwenden sowie die Botschaft *CM_DockClient* zu bearbeiten. Der TreeDesigner besitzt mit dem TreeExplorer (Formularklasse *TTreeExplorerForm*) und dem Übersichtsfenster (*TOverviewForm*) zwei andockbare Fenster, die jeweils von der (ebenfalls selbst definierten) Formularklasse *TDockableForm* abgeleitet sind. Aufgrund dieser Formularvererbung brauchen die für das Docking relevanten Methoden und Properties nur einmal gesetzt bzw. implementiert zu werden.

Das TreeDesigner-Hauptfenster hält auf der rechten Seite eine *TPanel*-Komponente als Andockstelle in Bereitschaft (*TPanel.DockSite* und *TPanel. AutoSize* wurden auf *True* gesetzt). Beim Formular des TreeExplorers wurden die gleichen Einstellungen vorgenommen wie bei den Symbolleisten in Kapitel 5.2.2: *DragKind* ist *dkDock* und *DragMode = dmAutomatic*.

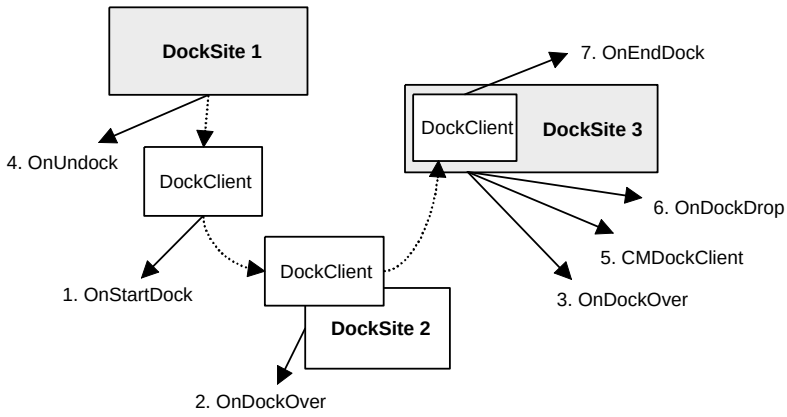


Abbildung 5.12: Die Ereignisse beim Drag&Dock eines andockbaren Fensters (DockClient) von einer ersten Andockstelle (DockSite1) über eine zweite hinweg bis zu einer dritten, über der die Maustaste losgelassen wird.

Wichtig ist außerdem, dass der Benutzer die Größe des TreeExplorers auch nach dem Andocken noch ändern kann. Hierzu befindet sich direkt neben der *TPanel*-Komponente eine mit *alRight* ausgerichtete *TSplitter*-Komponente, die sich dem Benutzer zur Laufzeit als nach rechts und links verschiebbarer vertikaler Balken darstellt.

Bemerkungen: Das Andocken freier Fenster würde auch im *DragMode dmManual* noch automatisch funktionieren. Selbst das Loslösen eines angedockten Fensters wäre dann noch mit einem Doppelklick möglich. Nur das Herausziehen eines angedockten Fensters aus seiner Andockstelle müssten Sie durch einen Aufruf von *BeginDock* selbst veranlassen.

Ein weiteres mit dem Docking zusammenhängendes Element der TreeDesigner-Benutzerschnittstelle ist der Popup-Menüpunkt *Andockbar*, den Sie aus den Hilfsfenstern der Delphi-IDE kennen. Über Ihn können Sie das Docking-Verhalten der beiden TreeDesigner-Hilfsfenster unterbinden. Im Programmcode funktioniert dies ganz einfach durch Zurücksetzen von *DragKind* auf *dkDrag*.

Wenn Sie in Ihrer Anwendung den Docking-Modus wie in der Delphi-IDE in den Umgebungsoptionen umschalten wollen, so dass das automatische Docking nur noch bei gedrückter Strg-Taste stattfindet, können Sie das Property *Application.AutoDragDocking* verwenden (ab Delphi 6).

Fenstermanagement

Der wesentliche Unterschied zum Docking von Symbolleisten besteht darin, dass sich die Symbolleisten zur Entwurfszeit im angedockten Zustand befinden, während TreeExplorer und Übersichtsfenster freie Formulare sind. Die Symbolleisten bekommen, falls der Benutzer es will, zur Laufzeit ein Hilfs-Elternfenster spendiert, über das der Benutzer sie frei über den Bildschirm bewegen kann. Ein solches Hilfsfenster ist für eine andockbare Komponente, die schon selbst ein Formular ist, natürlich nicht notwendig.

Was geschieht nun, wenn beispielsweise der TreeExplorer an das Hauptfenster angedockt wird? Vergleicht man den Vorgang mit dem Andocken von Symbolleisten, bei dem ja das dynamisch erzeugte Hilfs-Elternfenster für die Symbolleiste wieder entfernt wird, könnte man auf die Idee kommen, dass die VCL den Inhalt aus dem Formular heraustrennt und dann in die Andockstelle einfügt. Dem ist aber nicht so, wie Abbildung 5.13 zeigt. Das Fenster wird als Ganzes in das Hauptfenster eingefügt. Zwar wird VCL-intern das Fenster mit Rahmen erst gelöscht und dann ein neues Fenster ohne Rahmen als Kindfenster in das Hauptformular eingefügt, das *TTreeExplorerForm*-Objekt und alle darin enthaltenen Komponenten bleiben davon jedoch völlig unbeeinträchtigt.

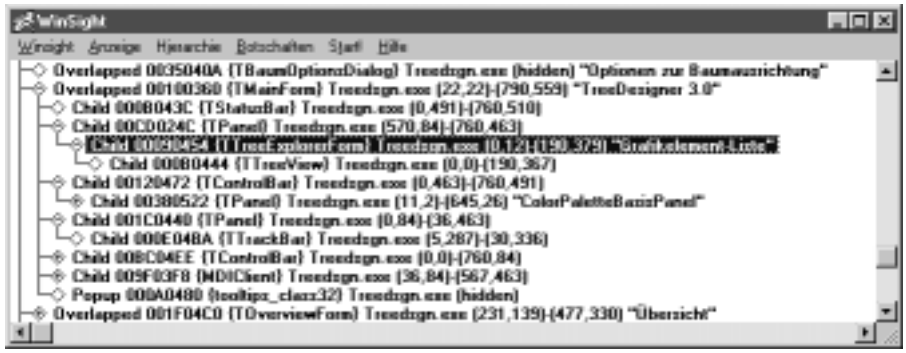


Abbildung 5.13: WinSight32 zeigt die Fensterhierarchie des TreeDesigners bei angedockter Grafikelemente-Liste (alias TreeExplorer) und frei beweglichem Übersichtsfenster.

Der Docking-Manager

Die Aussage, dass das angedockte Fenster keinen Rahmen hat, ist vielleicht etwas verwunderlich, sieht man doch nach dem Andocken sehr wohl eine Umrandung um die angedockten Fenster – sogar eine Kopfleiste ist vorhanden, wenn auch ohne Beschriftung. *Dieser* Rahmen wird jedoch automatisch von der Andockstelle gezeichnet und ist nicht dem angedockten Fenster zuzurechnen. (Dies kann man ab und zu in Fehlersituationen daran erkennen, dass der Rahmen gegenüber dem angedockten Fensterbereich verschoben dargestellt wird.)

Bevor es gleich um den im TreeDesigner für das Docking zuständigen Code geht, widmet sich dieser Abschnitt den vom Docking-Manager der VCL automatisch bereitgestellten Funktionen. Standardmäßig ist das Property *UseDockManager* einer *TPanel*-Komponente eingeschaltet, weshalb diese Funktionen aktiviert sind.

Der Docking-Manager erlaubt es, mehrere Fenster an einer Andockstelle anzudocken. Dabei teilt er die zur Verfügung stehende Fläche immer vollständig unter allen angedockten Fenstern auf. Das erste angedockte Fenster bekommt also die gesamte Fläche zugeteilt; kommt ein zweites hinzu, wird es auf die Hälfte verkleinert. In welche Hälfte das zweite Fenster eingelagert wird, hängt von der Bildschirmposition ab, an der der Benutzer die Maustaste beim Drag&Dock-Vorgang losgelassen hat. Während des Drag-Vorgangs werden hierfür die von der Delphi-IDE bekannten Vorschau-Rechtecke gezeichnet.

Eine Ausnahme bildet das Vorschau-Rechteck, das die Delphi-IDE im Zentrum der Andock-Bereiche anzeigt, um die mögliche Erzeugung eines Registers (*TPageControl*) zu symbolisieren, auf dessen Seiten die andockbaren Fenster verteilt werden. Um diese Funktionalität nachzubilden, müssen Sie einigen zusätzlichen Code selbst schreiben, ebenso, wenn Sie andockbare Fenster aneinander koppeln wollen (die andockbaren

Fenster sind dann selbst Andockstellen und die Vereinigung der beiden ist möglicherweise wieder ein andockbares Fenster).

Der TreeDesigner beschränkt sich auf die vordefinierten Andock-Variationen, denn erstens sollten diese für die Verwaltung nur zweier andockbarer Fenster ausreichen, und zweitens erfordert eine benutzerfreundliche Darbietung der Docking-Technik schon bei diesen einfachen Variationen einige zusätzliche Maßnahmen, die in den folgenden Abschnitten beschrieben werden.

Hinweis: Wenn Sie die *TPageControl*-Komponente nicht dynamisch zur Laufzeit erzeugen wollen, sondern schon zur Entwurfszeit fest in das Formular einzeichnen, brauchen Sie *keinen* zusätzlichen Code für das Docking, denn *TPageControl* ist auf das automatische Andocken von Fenstern ausgelegt und erzeugt sogar von selbst eine neue Seite, falls ein Fenster angedockt wird.

Unsichtbare und vergrößerbare Andockstellen *R67*

Wir kommen nun zu zwei einfachen Eingriffsmöglichkeiten in das Docking. Wie schon erwähnt, besteht die Andockstelle im Hauptfenster des TreeDesigners aus einer *TPanel*- und einer *TSplitter*-Komponente. Solange jedoch kein Fenster angedockt ist, ist die *TSplitter*-Komponente ziemlich unnötig, daher wurde sie zur Entwurfszeit unsichtbar gemacht. Auch die *TPanel*-Komponente ist bei Programmstart zunächst unsichtbar, da ihr *AutoSize*-Property eingeschaltet ist und sie dank der fehlenden Kindelemente eine automatische Breite von 0 zugeordnet bekommt (die Höhe wird nicht auf 0 gesetzt, da das Panel mit *alRight* ausgerichtet ist). Die nicht vorhandene Breite bewirkt zwar, dass die Andockstelle nicht als solche erkennbar ist, aber das ist für den Benutzer kein Problem, wie die Delphi-IDE ja zeigt. Auch für die Funktion des Dockings macht es nichts aus, wenn die Andockstelle gar keine Fläche hat, die VCL bezieht diese Fälle in ihre Berechnungen ein.

Hinweis: Die VCL realisiert dies, indem sie intern eine Liste aller Andockstellen verwaltet und bei jeder Mausbewegung alle diese Stellen nach verschiedenen Daten, wie etwa einem *InfluenceRect*, fragt. Dieses »Einflussbereichs-Rechteck« ist der Bereich, in dem die Andockstelle auf den Mauszeiger reagieren soll, wobei er größer sein kann als die Andockstelle selbst. Sie können in diese Vorgänge auch von Ihrem Programm aus eingreifen, indem Sie das Ereignis *OnGetSiteInfo* der Andockstelle bearbeiten.

Aufgabe des Programms ist es jetzt also, die Splitter-Komponente sichtbar zu machen, sobald eines der andockbaren Fenster an das Hauptfenster angedockt wird. Dazu muss es einfach das *OnDockDrop*-Ereignis des Andock-Panels bearbeiten. Dabei wird

gleichzeitig auch noch das *AutoSize*-Property des Panels abgeschaltet, da die Splitter-Komponente sonst wirkungslos bleiben würde. Schließlich muss noch der Fall beachtet werden, dass die Andockstelle eine Breite von Null hat:

```
procedure TMainForm.ToolwinDockSiteDockDrop(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer);
begin
  RightSplitter.Visible := True;
  ToolwinDockSite.AutoSize := False;
  if ToolwinDockSite.Width = 0 then begin
    // Einstellung der Breite auf maximal ein Viertel des Client-Bereichs
    // (siehe auch unten unter Bestimmung des Andock-Rechtecks).
    DockSiteWidth := Self.ClientWidth div 4; // Maximalbreite
    if Source.Control.Undockwidth < DockSiteWidth
       then DockSiteWidth := Source.Control.UndockWidth;
    ToolwinDockSite.Width := DockSiteWidth;
  end;
end;
```

Wenn der Benutzer ein Fenster aus der Andockstelle herauszieht, sendet diese eine *OnUnDock*-Nachricht, bei der überprüft werden muss, ob sich kein weiteres Fenster mehr in der Andockstelle befindet (die Zahl der Fenster lässt sich aus dem Property *DockClientCount* ablesen). Ist das der Fall, so werden die beiden eben gezeigten Schritte wieder rückgängig gemacht:

```
procedure TMainForm.ToolwinDockSiteUnDock(Sender: TObject;
  Client: TControl; NewTarget: TWinControl; var Allow: Boolean);
begin
  if ToolwinDockSite.DockClientCount = 1  // Client-Zahl vor dem Abdocken
  then begin
    RightSplitter.Visible := False;
    ToolwinDockSite.AutoSize := True;
  end;
end;
```

Bestimmung des Andock-Rechtecks *R91*

Das verbleibende Problem können Sie testen, indem Sie den Inhalt der folgenden Methode ausklammern und den TreeDesigner dann starten. Beim Andocken des Hilfsfensters wird dieses nämlich in seiner Originalgröße in das Hauptfenster eingebunden. Und das kann leicht so groß sein, dass kaum mehr Platz für die Dokumentfenster bleibt. Ein umsichtig agierendes Programm kümmert sich also auch um eine vernünftige Vorbelegung des *Docking-Rechtecks*, das ist eine *TRect*-Struktur, die die Koordinaten des angedockten Elements nach dem Andocken angibt.

Die beste Gelegenheit, dieses Rechteck festzulegen, ist das *OnDockOver*-Ereignis, denn die VCL verwendet das hier berechnete Rechteck auch zur Darstellung des Vorschau-Rahmens während des Mauszieh-Vorgangs. Dieser Vorschau-Rahmen ist auch wichtig

dafür, dass der Benutzer überhaupt erkennt, wo das Fenster andockbar ist. Sicher werden Sie in der Delphi-IDE bereits einige Erfahrung mit solchen umspringenden Vorschau-Rechtecken gemacht haben.

Die Methode für das *OnDockOver*-Ereignis führt zunächst den schon aus Kapitel 5.2.2 bekannten Test durch, ob das mit der Maus verschobene Element überhaupt akzeptiert werden kann. Ist das der Fall, berechnet sie ein Rechteck, das maximal ein Viertel der Breite des Hauptfensters überdeckt, und weist es dem Feld *Source.DockRect* zu. Die VCL erwartet darin ein Rechteck in Bildschirmkoordinaten, zur Errechnung davon dient die Methode *ClientToScreen*:

```
procedure TMainForm.ToolwinDockSiteDockOver(Sender: TObject;
  Source: TDragDockObject; X, Y: Integer; State: TDragState;
  var Accept: Boolean);
var
  ARect: TRect;
  Breite: Integer;
begin
  Accept := (Source.Control is TTreeExplorerForm)
            or (Source.Control is TOverviewForm);
  if Accept and (ToolwinDockSite.DockClientCount = 0) then begin
    Breite := Self.ClientWidth div 4; // Maximalbreite
    if Source.Control.Width < Breite then Breite := Source.Control.Width;
    ARect.TopLeft := ToolwinDockSite.ClientToScreen(
      Point(-Breite, 0));
    ARect.BottomRight := ToolwinDockSite.ClientToScreen(
      Point(0, ToolwinDockSite.Height));
    Source.DockRect := ARect;
  end;
end;
```

Wie der Name *TDragDockObject* schon andeutet, kapselt *Source* die wichtigen Daten des Docking-Vorgangs in einem Objekt. Dieses Objekt überdauert den gesamten Mauszieh-Vorgang – wenn Sie also in *OnDockOver* das *DockRect* setzen, bleibt dieses bis zum Loslassen der angedockten Komponente so gesetzt (falls der Benutzer die Komponente nicht wieder an eine andere Stelle zieht). Die VCL verwendet es dann am Schluss auch zur Einstellung der endgültigen Größe der Komponente in ihrer neuen Heimat.

Auch die obige Methode überprüft wieder anhand von *DockClientCount*, ob bereits ein anderes Fenster angedockt ist. Ist das der Fall, ist das Andock-Panel bereits sichtbar und der Docking-Manager der VCL berechnet von sich aus ein von der Mausposition abhängiges Docking-Rechteck, das in den bereits gegebenen Umfang des Panels hineinpasst. *ARect* ist dann schon gesetzt und wird von der Methode unverändert wieder zurückgegeben.

Speichern der Docking-Einstellungen *R28*

Für das Fenster-Docking gilt wie für das Docking von Symbolleisten: Wenn die Einstellungen des Benutzers zwischen den Programmläufen nicht gespeichert werden, ist das Docking nur die Hälfte wert. Denn was nutzen die flexibelsten Einstellmöglichkeiten, wenn der Benutzer nach jedem Programmstart zuerst eine halbe Stunde damit verbringen muss, sich an seine optimalen Einstellungen zu erinnern und diese wiederherzustellen. Es könnte sein, dass er dann lieber auf diese Einstellmöglichkeiten verzichtet.

Ziel ist also, die genaue Anordnung der angedockten Fenster zu speichern und beim nächsten Programmstart exakt wiederherzustellen. Wer einmal den für das Docking zuständigen Quelltext der VCL untersucht, wird feststellen, dass die Anordnung der Docking-Clients (das *Docking-Layout*) in einer Baumstruktur (vom Typ *TDockTree*) gespeichert wird, die sich aber von außen nicht abfragen lässt, da nicht alle privaten Klassenbereiche nach außen über Properties zugänglich sind. Zwar könnte man einfach die Koordinaten der angedockten Fenster speichern und beim nächsten Programmstart wiederherstellen, dabei würde dann aber der *DockTree* und mit ihm die übliche Funktionsweise der Andockstelle (wie das Verschieben der Trennbalken zwischen den angedockten Fenstern) verloren gehen.

Glücklicherweise hat Borland in der VCL Vorsorge für das anstehende Vorhaben getroffen und eine Methode *SaveToStream* (nebst zugehöriger *LoadFromStream*) geschrieben, die die gesamte Baumstruktur des Dockings speichert. *SaveToStream* speichert sowohl die Namen der angedockten Fenster als auch ihr Layout innerhalb der DockSite. *SaveToStream* ist im Interface-Typ *IDockManager* deklariert, und das *DockManager*-Property einer *TWinControl*-Komponente hat genau diesen Interface-Typ. Intern wird dieser Docking-Manager bzw. das Interface *IDockManager* durch die schon erwähnte Klasse *TDockTree* implementiert, jedoch spielt dies für die Anwendung keine Rolle. Wir müssen nur dafür Sorge tragen, dass das *UseDockManager*-Property der *TWinControl*-Komponente eingeschaltet ist, so dass wir über einen Docking-Manager verfügen können, der das Docking-Layout speichern kann.

Vor der Implementierung der Speicherfunktion gibt es allerdings noch drei Dinge zu bedenken:

- Der Docking-Manager kann die andockbaren Fenster nur dann im Stream speichern, wenn sie denselben Besitzer (Owner) wie die Andockstelle haben. Per Voreinstellung ist das Property *Owner* eines einzelnen Formulars wie des Tree-Explorers oder des Übersichtsfensters aber das *Application*-Objekt.
- Angenommen, das gespeicherte Docking-Layout soll wieder geladen werden. Nach dem Programmstart ist das Andock-Panel des Hauptfensters noch unsichtbar, was eine korrekte Wiederherstellung des Layouts verhindert. Die VCL speichert nur die Aufteilung der Fenster *innerhalb* einer Andockstelle. Wir müssen also

die Breite des Panels manuell speichern und wiederherstellen, bevor wir *DockManager.LoadFromStream* aufrufen. Was ebenfalls manuell gespeichert werden muss, ist die Größe, die ein angedocktes Fenster im nicht-angedockten Zustand hatte (*UndockHeight* und *UndockWidth*). Falls ein Fenster gar nicht angedockt ist, sollte sowohl seine Größe als auch seine Position gespeichert werden (die üblichen Properties *Left*, *Right*, *Width* und *Height* also).

- Schließlich benötigen wir auch noch einen Speicherort bzw. einen Stream, den wir an *SaveToStream* übergeben können. Wenn die Einstellungen wie beim TreeDesigner in der Windows-Registry gespeichert werden sollen, sollte der Stream kein Datei-Stream sein, sondern ein Memory-Stream, dessen Inhalt dann mit *TRegistry.WriteBinaryData* in die Registry übertragen werden kann (zu Memory-Streams siehe Kapitel 4.3.5).

Um das erste Problem zu lösen, müssen die andockbaren Fenster in den Besitz des Hauptfensters eingefügt werden. Die Basisklasse der beiden andockbaren Fenster des TreeDesigners, *TDockableForm*, definiert zu diesem Zweck ein Property *DockSiteOwner*. Wenn dies gesetzt wird, trennt sich das Fenster von seinem bisherigen Besitzer und fügt sich in den Besitz von *DockSiteOwner* ein:

```
type
  TDockableForm = class(TForm)
    ...
  private
    FDockSiteOwner: TWinControl;
    procedure SetDockSiteOwner(const Value: TWinControl);
  public
    property DockSiteOwner: TWinControl read FDockSiteOwner
                                write SetDockSiteOwner;
    ...

procedure TDockableForm.SetDockSiteOwner(const Value: TWinControl);
begin
  if FDockSiteOwner <> Value then begin
    FDockSiteOwner := Value;
    if Assigned(FDockSiteOwner) then begin
      Owner.RemoveComponent(self);
      DockSiteOwner.InsertComponent(self);
    end;
  end;
end;
```

Das Property wird vom Hauptformular wie folgt gesetzt:

```
procedure TMainForm.InitToolWindows;
// wird von "BeforeRun" aufgerufen - unmittelbar vor dem Start
// der Anwendung mit Application.Run
```

```
begin
  OverviewForm.DockSiteOwner := self;
  TreeExplorerForm.DockSiteOwner := self;
```

Die Speicherung der Layout-Daten findet beim Schließen des Hauptfensters in der Methode für das *OnClose*-Ereignis statt. Wie erwähnt verwendet diese einen Memory-Stream, um die Daten für die Registry vorzubereiten. Die Daten setzen sich aus der Breite des Andock-Panels (*ToolwinDocksite.Width*) und den vom Docking-Manager mit *SaveToStream* gespeicherten Daten zusammen. Mit *TRegistry.WriteBinaryData* werden sie schließlich gemeinsam unter dem Schlüssel *RightDockPanel* in der Registry abgelegt.

Nach der Speicherung des Layouts müssen noch die Positionsdaten der Fenster gesichert werden (dazu gehören *Left*, *Right*, *Top*, *Bottom*, *UndockWidth*, *UndockHeight* und das *Floating*-Flag). Hierfür verwendet *FormClose* wieder eine Hilfsfunktion aus der Unit *TDUtil*:

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
var
  S: TMemoryStream;
  W: Integer;
  R: TRegistry;
begin
  // Speichern des Hauptfenster-Zustandes:
  TDUtil.SaveWindowPos(IniPath+'MainWindow', self);
  // Speichern der Toolbar-Einstellungen (siehe Kapitel 5.2.2)
  TDUtil.SaveToolbarDockStatus(IniPath+'Palette', ColorPaletteBasisPanel);
    ... (wiederholt für alle anderen Toolbars)
  // Speichern des Zustandes der angedockten Hilfsfenster:
  S := TMemoryStream.Create;
  R := TRegistry.Create;
  try
    R.OpenKey(RegPath+'RightDockPanel', True);
    W := ToolwinDocksite.Width;
    S.Write(W, sizeof(W));
    MainForm.ToolwinDockSite.DockManager.SaveToStream(S);
    R.WriteBinaryData('DataStream', S.Memory^, S.Size);
  finally
    S.Free;
    R.Free;
  end;
  TDUtil.SaveControlDockStatus(IniPath, OverviewForm);
  TDUtil.SaveControlDockStatus(IniPath, TreeExplorerForm);
end;
```

Hinweis: Der tatsächliche Code im TreeDesigner ist noch etwas komplizierter, da hier auch das manuelle Docking berücksichtigt wird, das speziell für die Linux-Version entwickelt wurde (unter Kylix steht kein VCL-Docking zur Verfügung). Die Klasse *TDockableForm* definiert zu diesem Zweck eine Methode *SaveDockStatus*, die unter anderem die oben genannte Funktion *TDUtil.SaveControlDockStatus* aufruft. Die oben gezeigten letzten beiden Anweisungen in *FormClose* ändern sich dadurch wie folgt:

```
OverviewForm.SaveDockStatus(IniPath);
   TreeExplorerForm.SaveDockStatus(IniPath);
```

Zum Wiederherstellen der Docking-Konfiguration läuft der gesamte Vorgang umgekehrt ab, und zwar in der oben schon zur Initialisierung der *DockSite*-Properties verwendeten Methode *InitToolWindows*:

```
procedure TMainForm.InitToolWindows;
var
  S: TMemoryStream;
  W: Integer;
  R: TRegistry;
begin
  OverviewForm.DockSiteOwner := self; // (siehe oben)
  TreeExplorerForm.DockSiteOwner := self; // (siehe oben)
  S := TMemoryStream.Create;
  R := TRegistry.Create;
  try
    R.OpenKey(IniPath+'RightDockPanel', False); // Exception möglich
    S.SetSize(R.GetDataSize('DataStream'));
    R.ReadBinaryData('DataStream', S.Memory^, S.Size);
    S.Read(W, sizeof(W));
    if W <> 0 then begin
      RightSplitter.Visible := True;
      ToolwinDockSite.AutoSize := False;
      ToolwinDocksite.Width := w;
    end;
    ToolwinDockSite.DockManager.LoadFromStream(S);
  finally
    S.Free; R.Free;
  end;
end;
```

Was besonders wichtig ist, ist der Zeitpunkt der Wiederherstellung. Diese darf nämlich erst dann stattfinden, wenn die beiden andockbaren Fenster bereits konstruiert worden sind und wenn ihr *DockSiteOwner*-Property gesetzt wurde. Die Initialisierung der Fenster ist spätestens dann abgeschlossen, wenn alle *CreateForm*-Aufrufe der Projektdatei (PROJEKT | QUELLTEXT ANZEIGEN) ausgeführt wurden und unmittelbar, bevor *Application.Run* gestartet wird. Im TreeDesigner wurde die Projektdatei daher von Hand entsprechend modifiziert:

```
Application.Initialize;
Application.CreateForm(TMainForm, MainForm);
...
// hier zwischen werden noch 10 weitere Formulare initialisiert
Application.CreateForm(TOverviewForm, OverviewForm);
MainForm.BeforeRun; // <- ruft die obige Methode InitToolWindows auf
Application.Run;
```

5.8.3 Interaktion mit Grafikobjekten

Die einzelnen Grafikobjekte im TreeDesigner sollen noch zwei weitere Fähigkeiten erhalten, die bisher nur ganzen Komponenten zuteil wurden: Jedes Grafikobjekt soll ein Popup-Menü erhalten und die Farben der Objekte sollen per Drag&Drop geändert werden können.

Popup-Menüs für ganze Komponenten sowie Drag&Drop zwischen ganzen Komponenten sind durch die automatische Verhaltensweise der VCL sehr einfach zu implementieren. Schwieriger ist es, innerhalb einer Komponente zwischen verschiedenen Objekten zu unterscheiden. Hier kommen die Parameter *X* und *Y* ins Spiel, die sowohl beim Ereignis *OnDragOver/OnDragDrop* als auch bei *OnMouseDown* Auskunft über die Mausposition geben.

In beiden Fällen läuft die Feststellung des unter der Maus befindlichen Objekts in denselben Schritten ab:

- Grundsätzlich genügt es, die Flächen aller Objekte daraufhin zu überprüfen, ob sie die Mausposition enthalten. Damit das oberste Objekt gewählt wird, wenn sich mehrere Objekte an der Mausposition befinden, müssen die Objekte von oben nach unten durchsucht werden.
- Wenn Sie mit einem virtuellen Koordinatensystem arbeiten, müssen Sie vorher noch die Geräteeinheiten, in denen *X* und *Y* angegeben sind, in virtuelle Einheiten umrechnen.

Im TreeDesigner sehen diese beiden Schritte wie folgt aus:

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  MapMousePos(x, y);
  MouseObject := Document.GetObjectByPoint(Point(X, Y),
                                        GetGripSize, ResizeHandle);
  ...
```

Die Umrechnung der Koordinaten mit *MapMousePos* wird in Kapitel 5.6.3 erläutert, die Methode *TGraphicDoc.GetObjectByPoint* in Kapitel 5.4.2.

Popup-Menüs für Grafikobjekte *R58*

Im Fall der Popup-Menüs kommt noch eine weitere Umwandlung von Koordinaten hinzu: Das Popup-Menü soll an der Mausposition erscheinen, tut dies aber nicht automatisch, sondern verlangt die Koordinaten als Parameter der Methode *Popup*. Die Mausposition, die relativ zur linken oberen Fensterecke angegeben ist, muss daher in Koordinaten relativ zur linken oberen Bildschirmecke umgewandelt werden, was Sie mit der Methode *ClientToScreen* erreichen.

Das folgende Listing zeigt den Ausschnitt der Methode *TDocumentForm.PaintBox-MouseDown*, die für die Popup-Menüs zuständig ist (ein anderer Teil dieser TreeDesigner-Methode ist in Kapitel 5.4.1 gezeigt). *PaintBoxMouseDown* führt die beschriebenen Umwandlungen durch und testet danach, ob die rechte Maustaste gedrückt wurde. Ist das der Fall, gibt es zwei Möglichkeiten:

- Die Maustaste wurde in einem freien Bereich der Zeichenfläche geklickt, wodurch das Popup-Menü der Paintbox (*PaintBoxPopup*) aufgerufen wird.
- Oder die Maustaste hat ein Grafikobjekt getroffen, wodurch das Menü *ObjectPopup* aktiviert wird.

Beide Menüs werden wie jedes automatisch aktivierte Popup-Menü zur Entwurfszeit zum Formular hinzugefügt, werden aber nicht automatisch von der VCL aufgerufen (das *AutoPopup*-Property der Menüs ist *False*, das *PopupMenu*-Property der Paintbox bleibt leer). Beide Menüs werden auch in gleicher Weise aufgerufen, und zwar über die Methode *TPopupMenu.Popup*:

```
procedure TDocumentForm.PaintBoxMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  PopupPos: TPoint;
begin
  PopupPos.x := x; PopupPos.y := y;
  { Objekt an Position feststellen }
      MapMousePos(x, y);
      MouseObject := Document.GetObjectByPoint(Point(X, Y),
                                               GetGripSize, ResizeHandle);
  { rechte Maustaste abfangen }
  if Button = mbRight then begin
    PopupPos := PaintBox.ClientToScreen(PopupPos);
    if Assigned(MouseObject) then
      ObjectPopup.Popup(PopupPos.X, PopupPos.Y)
    else PaintBoxPopup.Popup(PopupPos.X, PopupPos.Y);
  end;
  ...
```

Die Verknüpfung der einzelnen Menüpunkte mit Methoden läuft genauso ab wie bei jedem anderen Popup- und Hauptmenü. Der Inhalt beider Menüs besteht aus Funk-

tionen, die im Laufe des fünften Kapitels bereits auf andere Weise aufgerufen wurden (*Markierte löschen* oder *Nach hinten schieben* für Tastaturbefehle), und aus spezielleren Methoden, die Sie im vollständigen Quelltext auf der CD finden. Das Popup-Menü der Grafikobjekte bietet beispielsweise Befehle zum Löschen von Verbindungen zu anderen Objekten und zum Anordnen der verbundenen Grafikelemente.

5.8.4 Drag&Drop

Ein Funktionsbereich, in dem die Laufzeit-Typinformationen von Object Pascal voll zur Entfaltung kommen können, ist das Drag&Drop (Ziehen und Ablegen). Bereits die Klasse *TControl* enthält alle für Drag&Drop notwendigen Properties und Methoden.

Funktionsweise

Über die Properties können Sie bereits im Objektinspektor nahezu beliebige Komponenten als *Quelle* einer Drag-Operationen bestimmen. Um Komponenten als Ablage*ziel* zu definieren, müssen Sie eine Zeile Code für das Ereignis *OnDragOver* schreiben. Um die Drop-Operation durchzuführen, bearbeiten Sie schließlich das Ereignis *OnDragDrop*.

Die folgende Tabelle fasst die beim Drag&Drop mitwirkenden Properties, Events und Methoden zusammen:

Elementart	***TControl*-Element**	**Aufgabe**
Event	OnDragOver	feststellen, ob eine Komponente das Ablegen gestattet (Parameter: *Sender*, *Source*, *X*, *Y*, *State*, *Accept*).
Event	OnDragDrop	benachrichtigt eine Komponente davon, dass der Benutzer etwas auf ihr abgelegt hat (Parameter: *Sender*, *Source*, *X*, *Y*).
Event	OnEndDrag	benachrichtigt ein Kontrollelement davon, dass ein bei ihm begonnener Drag&Drop-Vorgang beendet ist, erfolgreich oder erfolglos (Parameter: *Sender*, *Target*, *X*, *Y*).
Property	DragMode	wählt zwischen automatischem und manuellem Modus (*dmManual* oder *dmAutomatic*).
Property	DragCursor	gibt eine Mauszeigerform für die Drag-Operationen an, die bei dieser Komponente beginnen. Diese Form wird nur angezeigt, wenn der Mauszeiger sich über einem möglichen Ablageziel befindet.
Methode	BeginDrag	starten das Drag&Drop im Modus *dmManual*.
Methode	EndDrag	bricht das Drag&Drop in beiden Modi ab.

Bevor wir zu einer genaueren Erklärung der Properties und Methoden sowie zu einem Beispiel kommen, sehen wir uns den allgemeinen Ablauf des Drag&Drop an.

Starten des Drag&Drop

Eine Drag&Drop-Operation beginnt damit, dass der Benutzer eine Maustaste auf einem Objekt drückt, das er verschieben will.

Für die Anwendung beginnt die Operation bei einer Komponente. Wenn Sie das Property *DragMode* dieser Komponente auf *dmAutomatic* gesetzt haben, startet die VCL den Vorgang automatisch, sobald der Benutzer mit der linken Maustaste auf diese Komponente klickt, und verhindert, dass das sonst übliche Maus-Event *OnMouseDown* an das Formular gesendet wird. Wollen Sie dieses Event nicht missen, müssen Sie den Vorgang manuell starten (der Modus *dmManual* ist die Voreinstellung). In diesem Fall steht es Ihnen frei, auch andere Events als *OnMouseDown* (theoretisch sogar Tastatureingaben oder Timer-Events) als Startsignal zu verwenden.

Der Inhalt des Drag&Drop

Wichtig ist dabei, dass normalerweise nicht die Quellkomponente selbst verschoben wird. Wenn Sie die Drag&Drop-Events genauer betrachten, erkennen Sie, dass dadurch eigentlich gar nichts verschoben wird, denn die Events *OnDragOver* und *OnDragDrop* kennen keinen Parameter »zu verschiebendes Objekt«, sondern nur den Parameter *Source*, der die Quellkomponente angibt, die irgendetwas an eine andere Komponente senden soll, und eine Zielkomponente, auf der dieses »Etwas« abgelegt werden soll (diese Zielkomponente ist in den Ereignissen *OnDragOver* und *OnDragDrop* im Parameter *Sender* angegeben, denn von ihr gehen diese Ereignisse aus).

Die VCL übernimmt also nur die Vermittlung: Sie teilt der Zielkomponente mit, dass sie etwas von der Quellkomponente empfangen soll. Es liegt daraufhin in der Hand der Zielkomponente (bzw. der Methode, die deren *OnDragDrop*-Ereignis bearbeitet), sich die passenden Informationen selbst vom Parameter *Source* zu holen.

Diese Vermittlung der VCL ist so flexibel, dass Sie Drag&Drop-Operationen zu den verschiedensten Zwecken einsetzen können:

- Die einfachste Art des Drag&Drop ist das Bewegen von Dateien zwischen zwei Orten. Was mit den Dateien geschieht, hängt jedoch von weiteren Umständen ab, z.B. von der Art des Zielfensters oder von eventuell gedrückten Steuertasten der Tastatur: Verschieben, Kopieren, Löschen sind die einfachsten Drag&Drop-Dateioperationen, Übergabe eines Dokuments an eine Anwendung ist eine weitere Möglichkeit.
- Ähnlich wie bei Dateibewegungen können auch Einträge aus Listenfenstern und anderen Quellen leicht per Drag&Drop verschoben oder kopiert werden (zum Drag&Drop innerhalb einer ListBox siehe Kapitel 1.9.3, zu dem innerhalb eines ListViews Kapitel 3.6.2).

- Manchmal werden jedoch keine Objekte kopiert, sondern bestimmte Eigenschaften verändert. Der TreeDesigner verwendet nur diese Möglichkeit des Drag&Drop: Sie können Farben aus der Farbpalette auf die Grafikobjekte ziehen.
- In der abstraktesten Form kann Drag&Drop zu jeglicher Nachrichtenübermittlung zwischen einer vom Anwender ausgewählten Quelle und einem ebensolchen Ziel verwendet werden. Damit ist beispielsweise die Synchronisation zwischen zwei Fenster(bereichen) möglich – beziehen sich beispielsweise zwei Fenster auf dieselbe Information (bzw. dasselbe Dokument), zeigen aber zwei verschiedene Bereiche daraus auf verschiedene Arten an, könnten Sie die aktuelle Position aus dem einen in das andere Fenster ziehen, so dass dann beide Fenster dieselben Bereiche zeigen, aber weiterhin auf zwei verschiedene Arten. (So geht beispielsweise der Execution Trace Analyzer aus einer dem Autor vor scheinbar langer Zeit begegneten OS/2-Entwicklungsumgebung von IBM vor).

Drag

Nach dem »Greifen« des Objekts mit der Maus bewegt der Benutzer dieses an seinen Zielort. Solange der Drag&Drop-Vorgang noch nicht abgeschlossen ist, zeigt die Form des Mauszeigers dem Anwender an, ob das Objekt an der Stelle, an der sich die Maus gerade befindet, abgelegt werden kann.

Für die Delphi-Anwendung besteht der Ziehvorgang aus einer Reihe von *OnDragOver*-Events. Mit diesen fragt die VCL das Formular, ob die Komponente, die sich unter der Maus befindet, das Objekt von der Quellkomponente annehmen kann. Per Voreinstellung geben alle VCL-Komponenten eine negative Antwort. Dies führt dazu, dass die VCL den Mauszeiger in ein Verbotssymbol umwandelt.

Damit eine Komponente als Drop-Ziel dienen kann, müssen Sie eine Methode zum *OnDragOver*-Event dieser Komponente schreiben, in der Sie den Variablenparameter *Accept* auf *True* setzen. Falls es von der Mausposition abhängt, ob das Objekt akzeptiert wird oder nicht, können Sie zusätzlich die Parameter *X* und *Y* abfragen, die die Mausposition angeben. Die VCL startet nach jeder Mausbewegung eine neue Abfrage mit *OnDragOver*, auch wenn der Mauszeiger dadurch innerhalb derselben Komponente bleibt.

Wenn Sie mit *Accept=True* signalisieren, dass das Ablegen des Objekts erlaubt ist, verleiht die VCL dem Mauszeiger die Form, die Sie im *DragCursor* der Quellkomponente angegeben haben. Zum sehr selten benötigten *OnDragDrop*-Parameter *State* sei an dieser Stelle auf die Online-Hilfe verwiesen.

Drop

Wenn der Benutzer die Maustaste loslässt, erfahren sowohl Quell- als auch Zielkomponente davon. Für die Quellkomponente sendet die VCL dem Formular ein *OnDragEnd*-Ereignis, für die Zielkomponente das Ereignis *OnDragDrop*. Für beide Komponenten kann das Formular nun beliebige Aktionen durchführen. Wenn es sich bei beiden Komponenten um *TListBox*-Komponenten handelt, kann es beispielsweise einen Eintrag zwischen beiden Listen verschieben.

Ein Beispielprogramm für Drag&Drop-Varianten

Die Möglichkeiten des Drag&Drop sind so vielfältig, dass ein spezielles Demoprogramm sie am besten demonstrieren kann. Das Programm DragDropDemo, dessen Formular in Abbildung 5.14 gezeigt ist, enthält drei verschiedene Quell- und eine größere Sammlung von Zielkomponenten für Drag&Drop. Bei den meisten der Quellkomponenten wurde das *DragMode*-Property auf *dmAutomatic* gesetzt, lediglich ein Schalter bleibt für später beschriebene Zwecke im manuellen Modus.

Abbildung 5.14: Ein Formular zur Demonstration verschiedener Arten des Drag&Drop

Zielkomponenten R59

Das mittlere Panel (*DragOverTester*) demonstriert, welche grundlegenden Informationen Sie von der Quellkomponente (*Source*) des Drag&Drops erhalten können. Zu jedem Objekt, das seinen Luftraum erreicht, zeigt es zunächst den Klassennamen an. Da es außerdem voraussetzen kann, dass nur Komponenten gezogen werden, kann es

die gezogene Komponente *Source* auch in *TComponent* umwandeln und den Namen der Komponente anzeigen. Die Ausgabe der Informationen findet beim *OnDragOver*-Event statt:

```
procedure TForm1.DragOverTesterDragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  LabelKlassenName.Caption := Source.ClassType.ClassName;
  LabelObjektName.Caption := (Source as TComponent).Name;
  LabelEntfernung.Caption :=
    IntToStr(BaumEbene(Source.ClassType));{ siehe Kapitel 2.3.6 }
  Accept := True;
end;
```

Durch das Setzen von *Active* auf *True* akzeptiert die Komponente alle Objekte. Legt der Benutzer ein solches ab, zeigt *DragOverTester* den Namen dieser Komponente in der letzten Zeile an:

```
procedure TForm1.DragOverTesterDragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  LabelZuletzt.Caption := (Source as TComponent).Name;
end;
```

Das Editierfeld mit der Beschriftung *Datei-Akzeptor* ist wählerischer und akzeptiert das Drag&Drop nur, wenn es von einem Objekt der Klasse *TFileListBox* stammt. Wenn der Benutzer die Maustaste loslässt, zeigt es den Dateinamen, den die Dateilistbox im Property *FileName* aufbewahrt. Um aus den vielen möglichen Quellkomponenten nur die mit dem Typ *TFileListBox* auszuwählen, verwendet sie ebenfalls die Laufzeit-Typinformationen:

```
procedure TForm1.FilesAkzeptorDragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TFileListBox;
end;

procedure TForm1.FilesAkzeptorDragDrop(Sender, Source: TObject; X,
  Y: Integer);
begin
  Edit2.Text := (Source as TFileListBox).FileName;
end;
```

Das Panel »Schwarzes Loch« demonstriert, wie wenig die Komponenten untereinander vor schlechtem Benehmen geschützt sind: Es bereitet ihm keine Probleme, einfach jede Komponente, die um Ablageerlaubnis bittet, an sich zu ziehen und von der Bildfläche verschwinden zu lassen (indem es die Komponente versteckt – ein Löschen mit *Free* würde zu schweren Fehlern führen, da die Drag&Drop-Logik der VCL nicht damit rechnet, dass die Drag-Quelle plötzlich verschwindet).

Manuelles Drag&Drop

Der Unterschied des manuellen Drag&Drop-Modus zum automatischen besteht lediglich darin, dass eine Komponente, deren *DragMode*-Property *dmManual* ist, den Drag&Drop-Vorgang starten muss, indem sie die Methode *StartDrag* aufruft. Der Ablauf der Aktion verändert sich im manuellen Modus nicht, d.h. dass die VCL alle Drag&Drop-Events erzeugt, die auch beim automatischen Ablauf auftreten.

Grundsätzlich können Sie *StartDrag* jederzeit aufrufen, auch wenn keine Maustaste gedrückt ist. Die VCL verfolgt den Lauf der Maus wie bei einem normalen Ziehvorgang, beendet diesen Vorgang jedoch erst dann, wenn eine Maustaste losgelassen wurde (oder Sie auch das Ende manuell setzen, indem Sie die Methode *EndDrag* aufrufen).

Drag&Drop mit der rechten Maustaste

Eine Anwendung für manuelles Starten von Drag&Drop ist beispielsweise, den Ziehvorgang mit der rechten Maustaste zuzulassen:

```
procedure TForm1.Button1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbRight then
    (Sender as TControl).BeginDrag(True);
end;
```

Der Parameter von *StartDrag* besagt, ob der Vorgang sofort gestartet werden soll (*True*) oder ob die VCL so lange warten soll, bis die Maus ein winziges Stück bewegt wurde.

Ein Anwendungsbeispiel für globale Nachrichtenbearbeitung *R164*

In früheren Delphi-Versionen war das Drag&Drop mit der rechten Maustaste nicht so einfach, da die VCL den Vorgang von sich aus nur beendete, wenn die *linke* Maustaste losgelassen wurde. Der Drag&Drop-Vorgang musste also manuell beendet werden, was jedoch nicht so einfach war, da nach dem Starten des Ziehvorgangs keinerlei *OnMouseMove*- oder *OnMouseUp*-Ereignisse an das Formular gesendet werden. Auch wenn diese Situation in Delphi 6 nicht mehr tatsächlich vorkommt, gibt sie doch ein gutes Anwendungsbeispiel zur globalen Bearbeitung von solchen Nachrichten, an die anders kaum heranzukommen ist:

Da während des Drag&Drop die auf Windows-Ebene ohne Frage entstehenden Mausereignisse das Formular nicht erreichen, bleibt nichts anderes übrig, als die Nachricht früher auf ihrem Weg zu unterbrechen. Wie in Kapitel 3.1.4 besprochen, ist die allererste von der VCL angebotene Gelegenheit, eine Nachricht abzufangen, das *OnMessage*-Ereignis des *Application*-Objekts.

Das Beispielprogramm *DragDropDemo* demonstriert diese Art von Drag&Drop mit der rechten Maustaste für die beiden entsprechend beschrifteten Komponenten des Testformulars. Statt also wie oben einfach *BeginDrag* aufzurufen, speichert sie die Komponente, bei der der Ziehvorgang beginnt, in einer Formularvariablen, so dass später die *EndDrag*-Methode dieser Komponente aufgerufen werden kann. Außerdem verknüpft sie das *OnMessage*-Ereignis mit der Formularmethode *GlobalMessageRButtonTest*:

```
if Button = mbRight then begin
  DragControl := Sender as TControl;
  DragControl.BeginDrag(True);
  Application.OnMessage := GlobalMessageRButtonTest;
end;
```

GlobalMessageRButtonTest bleibt nur so lange mit dem *OnMessage*-Ereignis verbunden, bis sie eine *WM_RButtonUp*-Nachricht empfängt. Sobald dies geschieht, beendet sie den Ziehvorgang mit *EndDrag* und löst die Verknüpfung von *OnMessage* zu sich selbst:

```
procedure TForm1.GlobalMessageRButtonTest
                (var Msg: TMsg; var Handled: Boolean);
begin
  if Msg.message = WM_RBUTTONUP then begin
    DragControl.EndDrag(True);
    Application.OnMessage := nil;
  end;
end;
```

Zielgenaues Drag&Drop im TreeDesigner

Im TreeDesigner wird Drag&Drop auf eine etwas feinere Weise eingesetzt als in *DragDropDemo*: Farben können von der Farbpalette auf einzelne Grafikelemente auf der Zeichenfläche gezogen werden und die Frage, ob das »Loslassen« der Farbe erlaubt ist, entscheidet sich nicht allein anhand der Zielkomponente, sondern anhand der Position der Maus über der Zielkomponente. Hierfür muss wie schon in Kapitel 5.8.3 für die Popup-Menüs festgestellt werden, ob sich ein Grafikelement an der Mausposition befindet.

Dabei kommen erneut die Methoden *MapMousePos* und *GetObjectByPoint* zum Einsatz. Zunächst akzeptiert die Methode für das *OnDragOver*-Event der Paintbox das Drag&Drop nur, wenn sich ein Grafikobjekt an der Mausposition befindet:

```
procedure TDocumentForm.PaintBoxDragOver(Sender, Source: TObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean);
var
  Obj: TGraphicElement;
begin
  Accept := False;
  MapMousePos(x, y);
  Obj := Document.GetObjectByPoint(Point(X, Y));
```

```
    if Assigned(Obj) and (Source is TColorPalette) then
      Accept := True;
  end;
```

Die Form des Mauszeigers wird daher sofort angepasst, wenn Sie die Maus von der freien Fläche auf ein Grafikobjekt und umgekehrt bewegen.

Die Methode für das *OnDragDrop*-Event nimmt die mit der linken Maustaste selektierte Farbe (*PenColor*) von der Farbpalette an und setzt die Füllungsfarbe (*Brush.Color*) des Zielelements auf diese. Logischer wäre es natürlich, statt dessen *Pen.Color* zu setzen, des grafischen Effekts wegen wurde jedoch die Änderung der Füllungsfarbe vorgezogen.

```
procedure TDocumentForm.PaintBoxDragDrop(Sender, Source: TObject; X,
  Y: Integer);
var
  Obj: TGraphicElement;
begin
  MapMousePos(x, y);
  Obj := Document.GetObjectByPoint(Point(X, Y));
  if Assigned(Obj) then
    if Source is TColorPalette then
      Obj.Brush.Color := (Source as TColorPalette).PenColor;
      { Aufgrund eines in Kapitel 5.3.3 beschriebenen Mechanismus
        "merkt" das Grafikelement diesen Property-Schreibzugriff
        und kann die Neuausgabe am Bildschirm veranlassen sowie den
        Zähler für ungespeicherte Änderungen erhöhen. }
end;
```

5.9 XML

Im Zuge der zunehmenden Vernetzung der Computer ist in den letzten Jahren der Bedarf an einem flexiblen Datenformat, mit dem Daten problemlos auch zwischen den unterschiedlichsten Rechnerarchitekturen transportiert werden können, immer größer geworden. Diesem Bedarf trug die Entwicklung von XML Rechnung. XML ist zwar an sich noch kein Datenformat, sondern der Bezeichnung »eXtensible Markup Language« nach eher eine Sprache, aber auf Grundlage dieser Sprache lassen sich hervorragend Datenformate definieren, durch die sich beispielsweise das Problem der unterschiedlichen Dateiformate unterschiedlicher Anwendungen oder verschiedener Versionen einer Anwendungen lösen lässt.

Die in Delphi integrierten XML-Fähigkeiten beziehen sich jedoch weniger auf das Format von persistenten (z.B. auf Festplatte gespeicherten) Dateien als auf das Format von Nachrichten, die zwischen verschiedenen Rechnern ausgetauscht werden: Beispielsweise können Informationen aus Datenbanken im XML-Format zwischen einem Server und einem Client übertragen werden (wobei meist der Client ein so genannter

»Thin-Client« ist und der vom Client aus als »Server« erscheinende Rechner in Wirklichkeit nur die zweite Schicht einer mehrschichtigen Anwendung darstellt) oder die Daten einer *ClintDataSet*-Komponente in einer XML-Datei gespeichert werden. Neu in der Enterprise-Ausgabe von Delphi 6 ist die Unterstützung des SOAP-Protokolls, bei dessen zwischen den Rechnern ausgetauschten Nachrichten es sich ebenfalls um XML-Dokumente handelt.

Diese konkret in Delphi implementierten Anwendungsfälle für XML sind jedoch nur Beispiele aus den unzähligen Einsatzgebieten von XML. Dieses Kapitel soll die Grundlagen von XML so darstellen, dass Sie in die Lage versetzt werden, XML in von Ihnen selbst bestimmten Einsatzgebieten zu verwenden:

- In Kapitel 5.9.1 geht es zunächst um den Aufbau von XML-Dateien und die Repräsentation eines XML-Dokuments in Object Pascal durch Klassen bzw. Interfaces, die sich an den DOM-Standard halten.
- Kapitel 5.9.2 beschreibt als kleines Beispiel eine Erweiterung des TreeDesigners, die ein beliebiges XML-Dokument einliest und in einer grafischen Baumstruktur ausgibt.
- Kapitel 5.9.3 befasst sich daraufhin mit dem schon erwähnten Problem der Dateiformate, die durch XML wesentlich benutzerfreundlicher gehandhabt werden können (Stichwort »Kompatibilität« zwischen verschiedenen Formatversionen).

5.9.1 XML-Grundlagen

XML-Dateien sind menschenlesbare Textdateien, die sowohl die eigentlichen Daten enthalten als auch so genanntes *Markup* in Form von *Tag*s. Das Markup dient dazu, die *Bedeutung* der Daten zu kennzeichnen. In einem Dateiformat *ohne* Markup könnte beispielsweise ein Grafikelement des TreeDesigners wie folgt gespeichert werden:

```
TAutoObject/300/380/10/120/1/0/16777215/0/0/20/Arial
```

Der TreeDesigner selbst könnte dieses Dateiformat ohne weiteres lesen, da er genau wüsste, dass das erste Datenelement der Zeile die Beschriftung des Grafikelements angibt, das zweite die X-Koordinate der linken oberen Ecke, das dritte die Y-Koordinate usw. Nach diesem Prinzip verfährt der TreeDesigner ja in der Tat beim Speichern der `tvf`-Dateien durch die in Kapitel 5.3.3 gezeigten Methoden. Da die `tvf`-Dateien binär sind, lassen sie sich lediglich nicht so leicht vom Menschen lesen wie die oben abgedruckte Textzeile.

Nehmen wir nun an, der TreeDesigner 3.5 speichert das oben in der einzeiligen Datei ohne Markup gezeigte Grafikelement in einer XML-Datei (Kapitel 5.9.3 zeigt die dafür verwendeten Methoden). Diese XML-Datei sähe dann wie folgt aus:

```
<TreeViewerDocument Version="10">
  <GraphicElement shapeText="TAutoObject" left="300" right="380" top="10"
   bottom="120" FShapeType="1" penColor="0" brushColor="16777215"
   fontColor="0" penWidht="0" fontHeight="20" fontName="Arial">
  </GraphicElement>
</TreeViewerDocument>
```

Es dürfte leicht einsehbar sein, dass es nicht mehr unbedingt den TreeDesigner dafür erfordert, um die obige Datei interpretieren zu können. Jedes andere Programm, das Textdateien des obigen Formats in eine interne baumartige Datenstruktur überführen kann, könnte mit diesen TreeDesigner-Daten umgehen; unabhängig davon, unter welchem Betriebssystem oder auf welcher Rechnerplattform das Programm läuft (das Textformat dieser Datei ist auch leicht über das Internet übertragbar). Insbesondere könnten zukünftige TreeDesigner-Versionen die Liste der Attribute eines Grafikelements problemlos erweitern. Jedes neue Attribut würde durch einen eigenen Namen erkennbar sein und die früheren Programmversionen könnten das Attribut einfach ignorieren. Liest dagegen eine neue Programmversion eine alte Datei mit »fehlenden« Attributen, so ist auch dies leicht zu erkennen und die neue Programmversion kann diese Attribute mit Standardwerten belegen.

Die Frage ist nur, ob es nicht sehr aufwändig für ein Programm ist, Textdateien im XML-Format einzulesen (zu parsen). Normalerweise wäre dies zu bejahen, doch gehört zum XML-Standard auch die Definition von Programmierschnittstellen für Parser und für eine baumartige Datenstruktur, durch die ein XML-Dokument im Programm repräsentiert werden kann. Mit Hilfe fertiger Bibliotheken, die sich an diese Standards halten, ist es dann nur noch ein sehr kleiner Aufwand für den Entwickler, XML-Dateien zu bearbeiten. Der Aufwand ist trotz der großen Vorteile des flexiblen Datenformats nicht einmal höher als der zum Programmieren der Lese- und Schreibmethoden für binäre Dateien.

Ein sehr auffälliger äußerlicher Unterschied zwischen dem Beispiel-XML-Listing und der davor gezeigten einzeiligen Datenversion liegt wohl darin, dass sich die Größe der Daten durch das Markup vervielfacht, denn das Markup nimmt wesentlich mehr Platz ein als die eigentlichen Daten. Dies ist ein typisches Merkmal der meisten XML-Dateien und es wurde bei der Entwicklung von XML bewusst in Kauf genommen, da man aufgrund stetig wachsender Speicherkapazitäten und Datenübertragungsraten die Zeit für reif gehalten hat, mit dem knausrigen Zählen einzelner Bytes Schluss zu machen. Außerdem lässt sich eine XML-Datei hervorragend komprimieren, da sich der Text der Markups ständig wiederholt (in einem TreeDesigner-Dokument mit hundert Grafikelementen würde der größte Teil der Datei durch die hundert Vorkommnisse der Markup-Strings *ShapeText*, *left*, *right*, *top* etc. verbraucht werden).

Hinweis: Die von Delphi verwendeten Formulardateien enthalten schon seit Delphi 1 ebenfalls eine Art Markup: So werden etwa nicht einfach die Inhalte der Property-Einstellungen des Formulars und seiner Komponenten in die Datei geschrieben, sondern immer auch die zugehörigen Property-Namen. Diese Formulardateien werden zwar bisher nicht im XML-Format geschrieben, aber für den Datenaustausch zwischen den verschiedenen Delphi-Versionen und nun auch von Delphi nach Kylix bewährt sich das flexible Markup-Prinzip hervorragend (lediglich die in Kylix vorgenommenen Erweiterungen am Dateiformat bedeuten eine Einschränkung, da die Kylix-Formulardateien erst ab Delphi 6 wieder gelesen werden können).

Typdefinitionen und Style-Sheets

Die oben gezeigte TreeDesigner-XML-Datei verwendet zahlreiche selbst definierte Markup-Strings: Alle Zeichenketten, die nicht in Anführungszeichen eingeschlossen sind, gehören zum Markup. Die Sprache XML definiert lediglich die allgemeine Struktur der XML-Datei, die Markup-Strings werden von der Anwendung festgelegt. Wenn wir diese Markup-Strings mit den vordefinierten Wörtern z.B. der Sprache Object Pascal vergleichen, können wir jede XML-Datei als in einer speziellen »Sprache« geschrieben betrachten. Die Sprache XML ist damit ein Hilfsmittel zur Definition spezieller Sprachen, daher wird XML auch als *Metasprache* bezeichnet.

Zur »TreeDesigner-Grafikdokumentbeschreibungs-Sprache« gehören beispielsweise die Elementtypen *TreeViewerDocument* und *GraphicElement*, wobei ein *GraphicElement* über ein vordefiniertes Set von Attributen verfügt, wie *shapeText*, *brushColor* und *fontHeight*. Die Struktur des TreeDesigner-XML-Datei wird implizit durch den TreeDesigner definiert. Als Metasprache sieht XML jedoch auch konkrete Formen vor, mit denen die Struktur eines Dokuments sich schriftlich definieren lässt. Auch wenn wir in diesem Kapitel keinen Gebrauch der folgenden beiden Möglichkeiten machen werden, sind sie doch eine Erwähnung wert:

- Die bisher provisorisch standardisierten Document Type Definitions (DTD) und die neueren XML-Schemas sind Dateien, in denen auf vom XML-Standard definierte Weise genau festgelegt wird, welche Elemente in einem XML-Dokument vorkommen dürfen, welche Elemente in welchen anderen Elementen enthalten sein dürfen (in der TreeDesigner-XML-Datei dürfen beispielsweise *Edge*-Elemente, welche die Verbindungen zwischen Grafikobjekten angeben, in *GraphicElement*-Elementen enthalten sein, aber nicht umgekehrt), welche Attribute erlaubt sind und welche Typen diese haben. Mit Hilfe einer solchen DTD bzw. eines solchen Schemas kann der Parser ein Dokument beim Einlesen auf die so genannte *Gültigkeit* überprüfen, noch bevor die Anwendung über das DOM auf den Dokumentinhalt zugreift. Wie beim Parsen eines Object-Pascal-Quelltextes durch den Delphi-Compiler können auch

beim Parsen des XML-Dokuments verschiedene Fehler vorkommen, etwa Attributwerte, die einen falschen Typ aufweisen.

XML definiert zwei Kriterien, nach denen sich ein XML-Dokument auf Korrektheit überprüfen lässt: Wenn es die Regeln einer DTD/eines XML-Schemas komplett erfüllt, wird es wie erwähnt als gültig (*valid*) bezeichnet. Gibt es keine DTD, aber erfüllt das Dokument einige Grundregeln, kann es noch als wohlgeformt (*well-formed*) gelten. Zu diesen Grundregeln gehört etwa, dass jedes Element durch korrespondierende Tags in spitzen Klammern eingeschlossen sein muss (z.B. `<Zahl>1</Zahl>`) oder dass sich Elemente bei der Schachtelung nicht überschneiden, wie es etwa bei den Elementen von HTML-Dokumenten möglich ist.

- Auch ein *Style-Sheet* befasst sich mit den verschiedenen in einem XML-Dokument vorkommenden Elementtypen und Attributen, jedoch aus einer anderen Perspektive: Das Style-Sheet definiert, wie die Daten des Dokuments darzustellen sind. Von HTML ist mit den Cascading Style Sheets (CSS) eine ähnliche Technik bekannt: Dort werden Absatzformate definiert, die jeweils für einen bestimmten Typ von Absatz gelten. Der Typ eines Absatzes wird auch in HTML durch Tags angegeben, so ist etwa *H1* der Typ des Absatzes `<H1>Überschrift</H1>`. Einen ähnlichen Zweck verfolgen auch die XML-Stylesheets, mit dem Unterschied, dass sie viel allgemeiner sind und etwa dazu dienen können, ein und dasselbe XML-Dokument einmal als Tabelle von Zahlen anzuzeigen und ein anderes Mal (unter Verwendung eines zweiten Style-Sheets) als Balkengrafik.

Neben DTD, XML-Schema und Style-Sheets umfasst XML noch zahlreiche andere Technologien und wird auch noch fortlaufend weiterentwickelt. Für weitere Informationen sei hier auf die umfangreiche Spezial-Literatur zu XML verwiesen. Der Bereich des XML-Standards, der in diesem Buch am meisten Aufmerksamkeit geschenkt bekommt, ist das DOM.

XML-DOM

Das *Document Object Model* (DOM) beschreibt eine Programmierschnittstelle, über die ein Programm auf alle Elemente und Daten eines XML-Dokuments zugreifen kann. Auf diese Weise kann ein Programm beispielsweise mit einer einzigen Anweisung ein theoretisch beliebig großes XML-Dokument einlesen und danach über DOM-Klassen auf dessen einzelne Bestandteile zugreifen. Das Parsen der XML-Textdatei liegt alleine in der Verantwortung der verwendeten Bibliothek. Die Klassen, die im DOM-Standard die Namen *Node*, *Element*, *Document*, *Attr* usw. tragen, werden in der Unit *xmldom* von Delphi 6 durch Interfaces repräsentiert, in deren Name noch ein »IDom« vorangestellt wurde, also etwa *IDomNode*, *IDomElement* etc. Allerdings ist die Unit *xmldom* als rein abstrakte Definition zu verstehen, denn sie enthält keine Klassen, die die Interfaces implementieren würden.

Der TreeDesigner verwendet die DOM-Implementation von Microsoft, weil sich diese am einfachsten auch schon in früheren Versionen von Delphi nutzen lässt. Borland hat die XML-Unterstützung von Delphi 6 zwar so flexibel entworfen, dass sich statt der von Microsoft auch andere Implementationen verwenden lassen, jedoch dürfte wohl in der Praxis meist die Microsoft-Implementation Verwendung finden, da diese vermutlich auf den meisten Systemen installiert ist (falls nicht, genügt natürlich wieder einmal die Installation eines aktuellen Internet Explorers, um die Bibliothek nachzurüsten).

MSXML für alle Delphi-Versionen

Die Microsoft-Implementation des DOM befindet sich in einer DLL namens *msxml.dll* und verwendet das COM, um ihre Funktionen für Anwendungen zugänglich zu machen. Dies bedeutet, dass die DLL Co-Klassen bereitstellt, welche die oben erwähnten DOM-Interfaces implementieren (zum theoretischen Hintergrund von COM und Co-Klassen siehe Kapitel 2.7).

Genau genommen implementieren die Co-Klassen von *msxml* nicht die allgemeinen *IDOM...*-Interfaces aus *xmldom.pas*, sondern die *IXMLDOM...*-Interfaces, die in *msxml* selbst definiert sind. Im Folgenden werden beide Varianten jedoch synonym verwendet: Wenn also beispielsweise von *IDomDocument* die Rede ist, wird im Programm tatsächlich eine *IXMLDomDocument*-Schnittstelle verwendet, und statt *IDomElement* kommt *IXMLDomElement* zum Einsatz. Auf Programmebene bedeutet dies:

```
type
  IDomElement = IXMLDomElement;
  IDomNode = IXMLDomNode;
  ...
```

Die Schnittstellen der Microsoft-Bibliothek unterscheiden sich von den allgemeinen der DOM-Bibliothek durch zusätzliche Elemente, die für den praktischen Einsatz der Schnittstelle entscheidend sind, aber im DOM-Standard absichtlich nicht festgelegt wurden (solche Erweiterungen sind beispielsweise Methoden, um ein Dokument in einer Datei oder auch nur in einem String abzulegen; es ist also zumindest in diesem speziellen Fall Microsoft kein Vorwurf zu machen, dass versucht wurde, einen Standard durch proprietäre Erweiterungen zu verwässern). Es ist im Folgenden immer erwähnt, wenn von einer solchen Erweiterung Gebrauch gemacht werden sollte.

Die Datei *msxml.pas* wird zwar erst ab Delphi 6 von Borland mitgeliefert, Sie können sie aber auch von früheren Delphi-Versionen generieren lassen, indem Sie PROJEKT | TYPBIBLIOTHEK IMPORTIEREN wählen, in der Liste des daraufhin erscheinenden Dialogs die Bibliothek *Microsoft XML* auswählen und den Dialog dann mit dem Schalter UNIT ANLEGEN schließen. Delphi erzeugt daraufhin eine Datei namens *msxml_tlb*; der TreeDesigner setzt jedoch voraus, dass diese unter dem Namen *msxml* gespeichert wird.

(Mehr zum theoretischen Hintergrund des Typbibliothek-Imports lesen Sie in Kapitel 8.6.2. Die genannte Import-Funktion steht übrigens schon ab Delphi 3 zur Verfügung, wurde aber vom Autor in Bezug auf *MSXML* erst ab Delphi 5 getestet.)

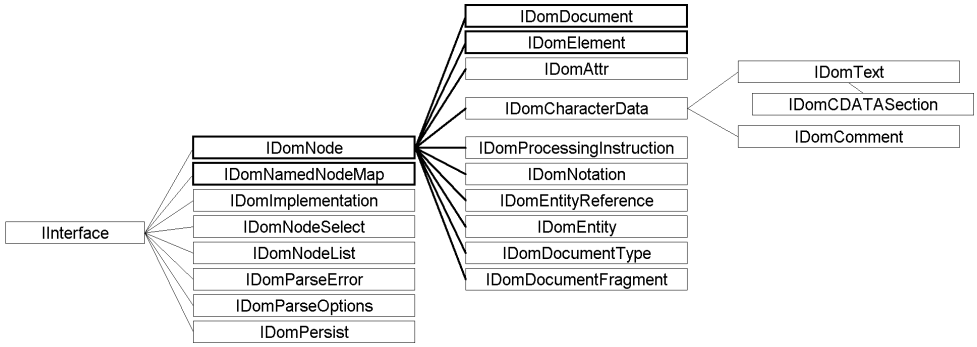


Abbildung 5.15: Die DOM 2.0 Interface-Hierarchie von Delphi 6. Die hervorgehobenen Schnittstellen werden im TreeDesigner verwendet.

Dokumentation

Für Windows-Programmierer bieten sich zwei Online-Dokumentations-Quellen zum DOM an:

- Die Microsoft-Dokumentation zur aktuellen Version der *msxml.dll* ist unter *msdn.microsoft.com/library* zu finden. Hier können Sie beispielsweise über die Suchfunktion direkt zu einzelnen Schnittstellen wie *IXMLDOMNode* springen. Während des Test des Autors stand auf dieser Seite auch ein Navigations-TreeView zur Verfügung, über den sich das Kapitel *Web Development* in strukturierter Weise überblicken ließ (allerdings funktionierte dies nur, wenn man die Ausführung von ActiveX-Komponenten im Browser erlaubte).
- Ultimative Referenz über den DOM-Standard finden Sie auf der Website des W3-Konsortiums unter *www.w3.org*. Da der hier ständig neue Dokumente hinzu kommen, ist es nicht möglich, in diesem Buch eine konkrete, aktuelle Adresse anzugeben. Die in der Abbildung erwähnten Interfaces von DOM 2.0 finden Sie beispielsweise unter *www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/core.html*.

Wenn Sie weiter gehende XML-Funktionen verwenden wollen, können Ihnen diese Referenzen allerdings wenig helfen, hier wäre dann ein spezielles, einführendes Buch zu empfehlen, das einen Überblick über den gesamten XML-Bereich gibt und die Grundprinzipien der verschiedenen Technologien erläutert.

Grundlegende Elemente von IDomNode

Wie Abbildung 5.15 zeigt, sind die DOM-Interfaces in einer Vererbungshierarchie angeordnet, deren Wurzel von *IDomNode* gebildet wird. *IDomNode* repräsentiert einen Knoten in der Baumstruktur eines XML-Dokuments und bietet allgemeine Properties an, um durch diese Baumstruktur zu navigieren:

Property	Bedeutung
parentNode: IDomNode	gibt den übergeordneten Knoten (Elternknoten) des aktuellen Knotens zurück.
firstChild: IDomNode	gibt den ersten Unterknoten (das erste Kind) des aktuellen Knotens an.
nextSibling: IDomNode	gibt den »Geschwisterknoten« an, der dem aktuellen Knoten in der Kinderliste des Elternknotens folgt.
ownerDocument: IDomDocument	weist auf einen Knoten, der das gesamte Dokument repräsentiert.

Mit diesen Methoden kann man, wenn man nur einen Knoten des Dokuments kennt, alle anderen Knoten erfragen: Es genügt, das Dokument-Objekt (*ownerDocument*) in Erfahrung zu bringen, bei diesem mit *firstChild* das erste Kind zu ermitteln und von diesem ausgehend mit *nextSibling* so lange das nächste Kind zu erfragen, bis *nextSibling nil* zurückgibt. Dies wird rekursiv für jeden Knoten wiederholt, der weitere Kinder hat[14], ein Beispiel dafür finden Sie in Kapitel 5.9.2.

Die bisher genannten Properties sind nur lesbar, jedoch gibt es unter anderem die folgenden Methoden, um die XML-Datenstruktur zu verändern:

Methode	Wirkung
function insertBefore(const newChild, refChild: IDomNode): IDomNode;	fügt den Knoten *newChild* vor dem existierenden Knoten *refChild* ein. (Entsprechend gibt es auch eine *insertAfter*-Methode.)
function replaceChild(const newChild, oldChild: IDomNode): IDomNode;	ersetzt den existierenden Knoten *oldChild* durch *newChild*.
function removeChild(const childNode: IDomNode): IDomNode;	entfernt den existierenden Knoten *childNode*.
function appendChild(const newChild: IDomNode): IDomNode;	hängt einen neuen Knoten als letzten Knoten an die Liste der Kind-Knoten an.

Die Rückgabewerte der Funktionen sind im Fehlerfall *nil*, ansonsten wird eine der als Parameter angegebenen *IDomNode*-Referenzen zurückgegeben, was aber im Beispiel-Code dieses Kapitels ignoriert wird.

14 Der Knoten *ownerDocument* hat immer nur ein einziges Kind, das *root*-Element.

Schließlich hat jeder Knoten einer DOM-Struktur auch noch bis zu vier wichtige Daten, die über weitere Properties von *IDomNode* angesprochen werden:

Property	Bedeutung
nodeType	enthält eine Kennung für die verschiedenen Knotentypen. Zu jedem Knotentyp definiert das DOM eine spezielle Klasse, von *Node* abgeleitete Klasse und entsprechend gibt es auch ein spezielles, von *IDomNode* abgeleitetes Interface. Einige dieser speziellen Klassen werden im folgenden Abschnitt behandelt, dabei wird auch erwähnt, durch welche *nodeType*-Kennung sich diese Klassen auszeichnen.
nodeName	Alle im eingangs gezeigten TreeDesigner-Beispiel vorkommenden Markup-Strings sind Knotennamen, die über *nodeName* abgefragt werden können. Der Knotenname kann je nach Art des Knotens auf verschiedene Weisen im XML-Dokument erscheinen: Als Name eines Elements wie *TreeViewerDocument* und *ELEMENT*, als Name eines Attributs wie *ShapeText* oder *Left* oder auf viele andere Weisen, für die auf die spezielle XML-Literatur verwiesen sei. Was jedoch hier noch erwähnt werden soll, ist, dass nicht alle Knoten eines XML-Dokuments über einen individuellen Namen verfügen. So haben beispielsweise alle Text-Knoten den Namen *#text*, alle Kommentarknoten den Namen *#comment* und der Dokumentknoten ist immer mit *#document* benannt.
nodeValue	Auch der Wert dieses Properties hängt von der konkreten Art des Knotens ab. Viele Knotentypen haben gar keinen Wert, bei ihnen ist *nodeValue* immer *nil*, so z.B. bei Element- und Dokument-Knoten. Bei Attribut-Knoten hingegen entspricht der Wert jeweils dem Datenelement, welches im TreeDesigner-Beispiellisting oben in Anführungszeichen angegeben wurde.
text	XML-Elemente können außer Attributen und weiteren Elementen auch noch Text enthalten, der über das Property *text* abgefragt werden kann. Dieser Text wird im XML-Dokument beispielsweise wie folgt angegeben: `<Element>Text des Elements</Element>`.

IDomDocument

Von dem, was *IDomDocument* den von *IDomNode* geerbten Elementen hinzufügt, benötigen wir in diesem Kapitel nur eine Methode und ein Property:

```
// Aus der Deklaration von IDomDocument:
function createElement(const tagName: DomString): IDomElement;
property documentElement: IDomElement
```

Mit *createElement* wird ein neues Element erzeugt, das danach beispielsweise mit *IDomNode.appendChild* an ein bereits existierendes Element angefügt werden kann. Das neue Element wird schon von *createElement* mit einem Namen versehen, die weiteren Eigenschaften des Elements müssen Sie getrennt über die *IDomElement*-Schnittstelle des Elements setzen (z.B. den Text über das *Text*-Property und die Attribute über die im nächsten Abschnitt genannten Methoden). Viele weitere Methoden zum Erzeugen von Elementen spezieller Typen finden Sie außerdem in der Online-Dokumentation beschrieben.

Jedes XML-Dokument muss genau ein Wurzel-Element enthalten, in dem dann alle anderen Elemente enthalten sind. Dieses Element ist nicht das *IDomDocument*-Objekt selbst, sondern es wird über dessen Eigenschaft *documentElement* angegeben. Für jedes XML-Dokument gibt es also ein *IDomDocument*-Objekt und ein *IDomElement*, welches das Wurzel-Element repräsentiert. In diesem sind dann je nach Dokument weitere Elemente und Attribute enthalten. Abbildung 5.16 zeigt die sich hieraus ergebenden Besitzverhältnisse der DOM-Objekte.

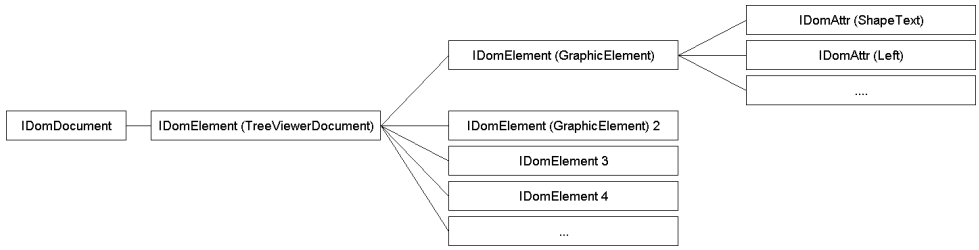


Abbildung 5.16: Die Besitzverhältnisse der DOM-Objekte am Beispiel eines TreeDesigner-Dokuments: Das Wurzelelement (TreeViewerDocument) ist in einem IDomDocument-Objekt enthalten; es enthält seinerseits die einzelnen Grafikobjekte als Unterelemente.

Hinweis: Vielleicht fragen Sie sich, warum es überhaupt ein separates *IDomDocument*-Objekt gibt, wenn doch dessen Inhalt in einem eigenständigen *IDomElement*-Objekt enthalten ist. Der Grund dafür liegt darin, dass *IDomDocument* einige weitere Daten enthält, die auch in der XML-Datei noch außerhalb des Wurzel-Elements angegeben werden. Dazu gehören beispielsweise Kommentare, Angabe der XML-Versionsnummer und die Verbindung des Dokuments mit einer separat gespeicherten DTD-Datei. Natürlich können Sie über das DOM auch auf diese Daten zugreifen, jedoch können diese weiter führenden Eigenschaften des *IDomDocument*-Objekts bzw. die darin enthaltenen Unter-Objekte nicht im Rahmen dieses Buches behandelt werden.

IDomElement

Der folgende Auszug aus der Deklaration von *IDomElement* zeigt einige der wesentlichen Eigenschaften, die ein XML-*Element* von anderen Knotentypen unterscheidet: Ein Tag, das über einen Namen verfügt (*tagName*, der allerdings auch über *IDomNode.nodeName* abgefragt werden kann), sowie eine Menge von Attributen, die über ihren Namen gesetzt und abgefragt werden können:

```
IDomElement = interface(IDomNode)
  ...
  function getAttribute(const name: DomString): Variant;
  procedure setAttribute(const name: DomString;value: Variant);overload;
```

```
    function Get_attributes: IDomNamedNodeMap;
    property tagName: DomString read get_tagName;
  end;
```

Die von *Get_attributes* zurückgelieferte *IDomNamedNodeMap*-Schnittstelle wird uns in Kapitel 5.9.2 die Möglichkeit geben, alle Attribute abzufragen, ohne deren Namen zu kennen. Sie soll an dieser Stelle nicht weiter untersucht werden.

5.9.2 Ein grafischer XML-Betrachter

Aufgrund ihrer Baumstruktur und der Tatsache, dass sie in reinem Textformat mit all den Tags noch nicht gerade bequem zu lesen sind, bieten sich XML-Dateien dem TreeDesigner als ein neues Betätigungsfeld an. Über den Menüpunkt DATEI | XML | BELIEBIGE XML-DATEI ALS GRAFIK ANZEIGEN können Sie eine Datei Ihrer Wahl einlesen. Die später abgedruckte Funktion *CreateTreeDesignerTree* durchläuft den gesamten DOM-Baum des Dokuments und erzeugt für jeden Element-Knoten ein eigenes Grafikelement, wobei natürlich auch die Verbindungen von jedem Element zu seinen Kindelementen in die Grafik übernommen werden. Anschließend werden diese neuen Grafikelemente mit der TreeDesigner-Methode *ArrangeFromObject* zu einem sichtbaren Baum angeordnet (siehe Abbildung 5.17). *ArrangeFromObject* wird auch zur Anordnung der Klassenhierarchiebäume verwendet und soll *nicht* das Thema dieses Kapitels sein.

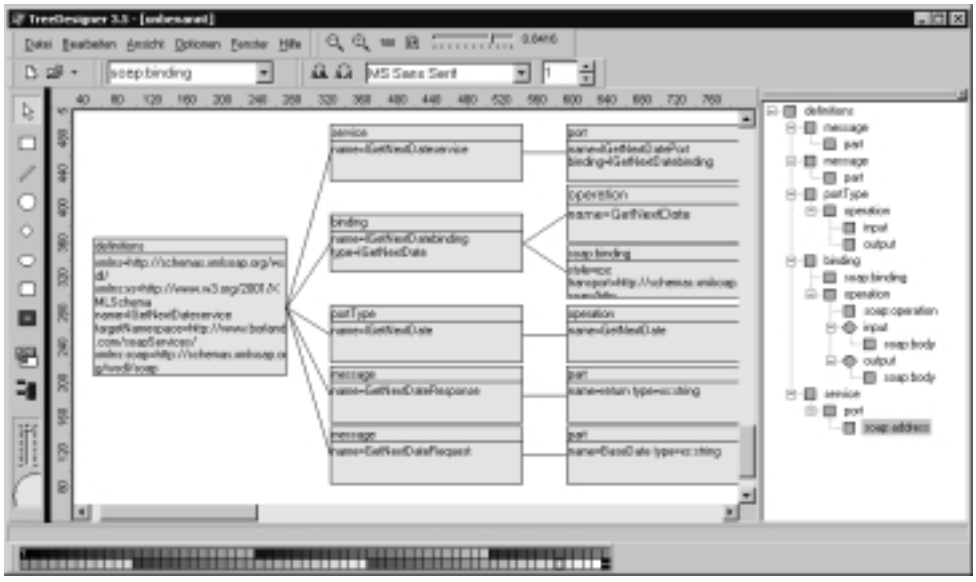


Abbildung 5.17: Auch die zahlreichen von der SOAP-Unterstützung in Delphi Enterprise intern generierten XML-Dokumente können im TreeDesigner visualisiert werden – hier die WSDL-Beschreibung des in Kapitel 8.8.4 entwickelten Beispiels.

Eine neue Grafikelement-Klasse für den TreeDesigner

Damit sich mit der XML-Grafik auch etwas anfangen lässt, sollen für die einzelnen Elemente mehr Informationen dargestellt werden, als sich sinnvoll in einem normalen TreeDesigner-Grafikelement darstellen lassen. Daher wurde eine neue Grafikelement-Klasse namens *TMultiLineElement* eingeführt, deren Deklaration Sie im Folgenden lesen können:

```
type
  TParagraphAlignment = (pfLeft, pfCenter, pfRight, pfBlock);
  TMultiLineElement = class(TGraphicElement)
  private
    FLongText: String;
    FParagraphAlignment: TParagraphAlignment;
    procedure SetParagraphAlignment(const Value: TParagraphAlignment);
    procedure SetLongText(const Value: String);
  public
    procedure LoadFromStream(Stream : TStream); override;
    procedure SaveToStream(Stream : TStream); override;
    procedure PaintText(CanvasRef : TVirtualCanvas); override;
    procedure AttrDialog; override;
    function CountNodeTextLines: integer; override;
    property LongText: String read FLongText write SetLongText;
    property ParagraphAlignment : TParagraphAlignment
       read FParagraphAlignment write SetParagraphAlignment;
  end;
```

Entscheidende Neuerung dieser Klasse ist das Property *LongText*, in dem sich zusätzlich zu dem von *TGraphicElement* geerbten Text-Attribut ein längerer Text speichern lässt, der nicht in der Mitte des Elements angezeigt, sondern von links oben beginnend in mehrere Zeilen umgebrochen wird. Das Property *ParagraphAlignment* ist für zukünftige Erweiterungen vorgesehen und wird normalerweise auf *pfLeft* eingestellt, um den Text linksbündig darzustellen.

Um die korrekte Funktionsweise des Elements zu gewährleisten, überschreibt *TMultiLineElement* außerdem fünf virtuelle Methoden von *TGraphicElement*:

- *LoadFromStream* und *SaveToStream* speichern die beiden neu eingeführten Properties und sorgen dafür, dass die im TreeDesigner bearbeiteten XML-Grafiken in `tvf`-Dateien gespeichert werden können.
- *PaintText* ersetzt die von *TGraphicElement* bekannte zentrierte Textausgabe. Wenn *LongText* nur einen leeren String enthält, ruft *PaintText* die geerbte *PaintText*-Methode zur zentrierten Grafikausgabe aus. Andernfalls sorgt es selbst für die Ausgabe von *LongText* mit automatischem Zeilenumbruch. Wenn zusätzlich zu *LongText* auch das von *TGraphicElement* »Standardtext« -Property *ShapeText* gesetzt

ist, schreibt *PaintText* diesen in die erste Zeile, *LongText* in die folgenden Zeilen und trennt die beiden noch durch eine Linie, wie sie etwa in Abbildung 5.17 in einigen Elementen sichtbar ist.

Wie *PaintText* im Gesamtvorgang des Zeichnens eingesetzt wird, können Sie dem Listing der Methode *TGraphicElement.Paint* aus Kapitel 5.5.2 entnehmen. Da dort bereits ein rechteckiger Clipping-Bereich für die Textausgabe eingestellt wird, kann auch der mehrzeilige Text der neuen Klasse nicht über die Grenzen des Elements hinauswachsen (der automatische Zeilenumbruch sollte dies horizontal sowieso verhindern, doch für das Abschneiden der letzten Zeile, die evtl. nicht mehr ganz in das Element hineinpasst, ist das Clipping ideal).

- *AttrDialog* ruft eine neue Version des Attributdialogs auf, in dem Sie den vom Element angezeigten Text editieren können (aufzurufen über das Popup-Menü des Grafikelements, siehe Kapitel 5.8.3).
- *CountNodeTextLines* wird für die automatische Baum-Anordnungsfunktion benötigt, um die Zeilenzahl eines Elements zu bestimmen. Sie ist für zukünftige Erweiterungen vorgesehen, da bisher immer nur Platz für drei Zeilen reserviert wird (wenn Sie dies ändern möchten, ändern Sie einfach diese Zahl in der Implementierung dieser Funktion!).

Dies soll zur Vorstellung der neuen Klasse genügen. Im Folgenden wird *TMultiLineElement* als gegebenes Werkzeug verwendet, die Implementierung der Methoden von *TMultiLineElement* kann aus Platzgründen nicht hier abgedruckt werden.

Darstellung der XML-Knoten durch Grafikelemente

Durch eine kleine Vorbereitung kann die Umsetzung der XML-Knoten in Grafikelemente erheblich vereinfacht werden. Und zwar enthält die Unit *TDXML* eine Art Übersetzungstabelle, die jedem Typ von XML-Knoten eine bestimmte grafische Form und eine Farbe zuordnet sowie außerdem einen String, der in der Grafik als Typbezeichner erscheinen soll.

```
type
  // Liste der im DOM möglichen Knotentypen:
  TNodeType = (ntInvalid, ntElement, ntAttribute, ntText, ntCData,
    ntEntityRef, ntEntity, ntProcessingInst, ntComment, ntDocument,
    ntDocumentType, ntDocumentFrag, ntNotation);
  // Grafik-Attribute, die mit den Knotentypen verknüpft werden können:
  TGraphicAttributes = record
    TypeText: string;
    ShapeType: TShapeType;
    Color: TColor;
  end;

// "Übersetzungstabelle" für die Knotentypen:
```

```
const
  GraphicAttributesTable : array[ntInvalid..ntNotation]
    of TGraphicAttributes = (
    (TypeText: 'ungültig'; ShapeType: stRect; Color: clWhite),
    (TypeText: 'Element'; ShapeType: stHexagon; Color: clYellow),
    (TypeText: 'Attribut'; ShapeType: stRect; Color: clBlue),
    (TypeText: 'Text'; ShapeType: stRect; Color: clGreen),
    (TypeText: 'CData'; ShapeType: stRect; Color: clWhite),
    (TypeText: 'Entity-Referenz'; ShapeType: stEllipse; Color: clRed),
    ...
  );
```

So werden beispielsweise normale *Element*-Knoten mit dem String »Element« versehen, als Sechseck gezeichnet und mit gelber Hintergrundfarbe versehen. Durch Ändern der Einträge in dieser Tabelle können Sie den Stil der XML-Grafik leicht Ihren eigenen Wünschen anpassen. Wenn Sie weitere »Spalten« in die Tabelle einfügen wollen, erweitern Sie zuerst den Typ *TGraphicAttributes* und berücksichtigen das Attribut auch in der Funktion *CreateTreeDesginerTree* nach dem Muster der schon vordefinierten Attribute.

Hinweis: Bei der automatischen Erzeugung der Grafikelemente wird der in der Tabelle angegebene *ShapeType* ignoriert, falls das Grafikelement mit einem *LongText* versehen wird. Da dieser Text linksbündig im Rechteck des Elements angezeigt wird, würden sich nämlich alle Grafikformen außer dem Rechteck mit dem Text überschneiden.

Erzeugen des XML-Dokument-Objekts

In Kapitel 5.9.1 wurde bisher angenommen, dass bereits irgendein *IDomNode*-Objekt verfügbar war, von dem aus man sich die anderen *IDomNode*- und *IDomElement*-Objekte liefern lassen konnte. Die Erzeugung des ersten DOM-Objekts, des Dokument-Objekts, ist noch offen. Da wir hier mit einer COM-Bibliothek arbeiten, findet die Erzeugung dieses Objekts wie üblich über den Konstruktor einer Co-Klasse statt. Die Deklarationen der Co-Klassen werden von Delphi beim Import der Typbibliothek automatisch generiert bzw. sind in der mit Delphi 6 mitgelieferten *msxml.pas* enthalten.

Die Co-Klasse für *IDOMDocument* heißt *CoDOMDocument* und wird wie folgt verwendet, um eine Variable des Typs *IXMLDomDocument* zu initialisieren:

```
var
  XMLDoc: IXMLDomDocument;
begin
  XMLDoc := CoDOMDocument.Create;
```

Danach können die Methoden von *XMLDoc* aufgerufen werden – etwa zum Einlesen eines Dokuments aus einer Datei, wie es im nächsten Abschnitt gezeigt wird.

Hinweis: Delphi 6 verfügt auch über eine Komponente namens *TXMLDocument*. Diese kann als nicht-visuelle Komponente in ein Formular eingefügt werden und kapselt eine DOM-Dokumentschnittstelle, die über das Property *DOMDocument* abgerufen werden kann. Die Herstellerunabhängigkeit dieser Komponente zeigt sich in den Properties *DOMImplementation* und *DOMVendor*, über die Sie wählen können, welche DOM-Implementation in Ihrer Anwendung verwendet werden soll. Die Funktionen zum Laden und Speichern eines Dokuments sind ebenfalls herstellerunabhängig in *TXMLDocument* verpackt: Wenn Sie in *FileName* einen Dateinamen angeben und die Komponente über das Property *Active* aktivieren, wird eine Datei geladen; für das Speichern gibt es eine *SaveToFile*-Methode. Der Zugriff auf die einzelnen Elemente des XML-Dokuments findet bei *TXMLDocument* genauso über die einzelnen *IDom...*-Schnittstellen statt, wie es in diesem Kapitel demonstriert wird. Für die Zwecke dieses Kapitels würde die Verwendung der Komponente *TXMLDocument* keine Vereinfachung bringen (würde doch im Wesentlichen nur der oben gezeigte Aufruf von *CoDOMDocument.Create* vermieden werden); auch aus Gründen der Kompatibilität mit früheren Delphi-Versionen verzichtet der TreeDesigner daher auf diese Komponente.

Einlesen eines XML-Dokuments

Nachdem ein *IDomDocument*-Objekt erzeugt wurde, kann über eine spezielle Methode dieses Objekts eine XML-Datei geparst werden. Diese Methode wurde in Kapitel 5.9.1 noch nicht erwähnt, weil sie implementationsabhängig ist. In der Microsoft-Implementation heißt sie *load*.

CreateTreeDesignerTree hat die Aufgabe, die Datei *FileName* einzulesen und für die darin enthaltenen XML-Knoten Grafikelemente in das Grafikdokument *Doc* einzufügen. Das Grafikelement, welches dem XML-Wurzel-Element entspricht, wird als Funktionsergebnis zurückgeliefert.

```
function CreateTreeDesignerTree(FileName: String;
            Doc: TGraphicDoc): TMultiLineElement;
// liefert das Grafikelement für das XML-Wurzelelement zurück.
var
  XMLDoc: IxmlDOMDocument;
  DocElem: IxmlDOMElement;
begin
  XMLDoc := CoDOMDocument.Create;
  XMLDoc.validateOnParse := True;
  XMLDoc.async := False;
  XMLDoc.load(FileName);
  DocElem := XMLDoc.documentElement;
  if Assigned(DocElem) then
    Result := CreateSubTree(Doc, DocElem, nil);
end;
```

Die Properties *validateOnParse* sowie *async* und die Methode *load* sind Spezialitäten der Microsoft-Implementation. Mit *validateOnParse* wird erreicht, dass der XML-Parser das eingelesene Dokument validiert, das Abschalten des *async*-Flags bewirkt, dass die *load*-Methode erst dann zurückkehrt, wenn das Parsen abgeschlossen ist.

Nach Einlesen der Datei mit *load* überprüft *CreateTreeDesignerTree* zunächst, ob ein Dokument-Element vorhanden ist (was natürlich nicht der Fall ist, wenn eine Nicht-XML-Datei geladen wird). Dann ruft sie die entscheidende Funktion *CreateSubTree* auf.

Hinweis: Da XMLDoc ein *Interface* ist, brauchen wir uns um die Freigabe nicht zu kümmern. Der vom Compiler erzeugte Code gibt das dahinter stehende Objekt automatisch am Ende der Methode frei (bzw. er verringert den Referenzzähler des Objekts, was wiederum dazu führt, dass dieses sich selbst freigibt, wenn der Zähler bei Null angekommen ist, siehe hierzu auch Kapitel 2.7).

Ein Durchlauf durch den DOM-Baum

Da *CreateSubTree* sich rekursiv selbst aufruft, wird diese Funktion letztlich für *jeden* Knoten im DOM-Baum aufgerufen. Sie erwartet als Parameter das Grafikdokument, den DOM-Knoten, der die Wurzel des zu behandelnden Teilbaums darstellt, und das Grafikelement, mit dem die neuen Grafikelemente verbunden werden sollen (dies ist also das zum Elternknoten gehörende Grafikelement):

```
function CreateSubTree(Doc: TGraphicDoc; Node: IDomNode;
    ParentGraphicElem: TMultiLineElement): TMultiLineElement;
```

CreateSubTree erzeugt immer nur ein einzelnes Grafikelement, nämlich das Grafikelement für den Wurzel-Knoten des Teilbaums (*Node*). Alle weiteren Grafikelemente werden in den rekursiven Aufrufen von *CreateSubTree* erzeugt.

Für die Erzeugung des Grafikelements für *Node* wird zunächst der Typ des Elements abgefragt: Bei einem Elementknoten werden die Werte der Attribute in *TMultiLineElement.LongText* gespeichert, bei Textknoten statt dessen der Textinhalt des Knotens (*Text*-Property). Bei allen anderen Knotentypen soll nur der Typbezeichner des Knotens (aus der oben gezeigten *GraphicAttributesTable*) im Grafikelement angezeigt werden (die Variable *ShortText* wird später an das *ShapeText*-Property des Elements übergeben).

Die im folgenden ersten Teil des Listings verwendeten DOM-Methoden/Properties sind *nodeType*, *tagName* und *Text*. Die Bezeichner *NODE_ELEMENT* und *NODE_TEXT* stehen für die Typkonstanten »1« und »3«, entsprechend der weiter oben gezeigten Typdeklaration von *TNodeType* (dieser Deklaration können Sie bei Interesse auch die anderen möglichen XML-Knotentypen entnehmen, die in *CreateSubTree nicht* gesondert berücksichtigt werden).

```
function CreateSubTree(Doc: TGraphicDoc; Node: IDomNode;
    ParentGraphicElem: TMultiLineElement): TMultiLineElement;
var
  ElementAttributes: TGraphicAttributes;
  Connection: TEdge;
  subNode: IDomNode;
  ShortText, LongText: string;
begin
  ElementAttributes := GraphicAttributesTable[TNodeType(Node.nodeType)];
  LongText := '';
  case Node.nodeType of
    NODE_ELEMENT: begin
      ShortText := (Node as IDomElement).tagName;
      LongText := GetAttributeString(Node as IDomElement); // s.u.
    end;
    NODE_TEXT: begin
      ShortText := ElementAttributes.TypeText;
      LongText := Node.Text;
    end;
    else ShortText := ElementAttributes.TypeText;
  end;
```

Nach diesem ersten Teil der Methode geht es mit der Erzeugung des *TMultiLineElement*-Objekts und seiner Verbindung zum Elternelement weiter. Hierzu werden Methoden und Properties verwendet, die teilweise in Kapitel 5.3 erläutert sind; da sie aber nichts mit dem Thema XML zu tun haben, sind sie im Folgenden ohne weitere Erläuterung aufgeführt:

```
  Result := TMultiLineElement.Create(Doc, Rect(0,0,0,0),
                ElementAttributes.ShapeType);
  // für langen Text muss die Grafikform ein Rechteck sein:
  if LongText <> '' then Result.ShapeType := stRect;
  Result.Text := ShortText;
  Result.Brush.Color := ElementAttributes.Color;
  Result.LongText := LongText;
  if Assigned(ParentGraphicElem) then
    Connection:=TEdge.StandardConnection(ParentGraphicElem, Result);
  Doc.Add(Result); // Das fertige Grafikelement zum Dokument hinzufügen
```

Den Abschluss der Methode bildet der rekursive Aufruf von sich selbst für alle Unterknoten von *Node*. Dabei kommen die DOM-Methoden *firstChild* und *nextSibling* zum Einsatz:

```
  // Unterknoten des aktuellen XML-Knotens durchlaufen:
  subNode := Node.firstChild;
  while subNode <> nil do begin
    CreateSubTree(Doc, subNode, Result);
    subNode := subNode.nextSibling;
  end;
end;
```

Was in dem obigen Listing noch offen geblieben ist, ist die Funktion *CreateAttributeString*. Sie hat die Aufgabe, alle Attribute eines Elements zu erfragen und mit Attributname und -wert in einen String zu schreiben. Der Grund, warum diese Funktion hier gesondert und am Schluss behandelt wird ist der, dass sie ein weiteres, bisher nicht erwähntes DOM-Objekt benötigt, die *NamedNodeMap*. Aus dieser lassen sich mit mit *get_item* die einzelnen Attribute abfragen. Jedes einzelne Attribut wird als *IDomNode* zurückgeliefert und gewährt über *get_nodeName* und *get_nodeValue* Zugriff auf Name und Wert des Attributs:

```
function CreateAttributeString(Node: IDomElement);
var
  AttrMap: IDomNamedNodeMap;
  AttrNode: IDomNode;
  i: integer;
begin
  Result := '';
  AttrMap := Node.get_attributes;
  for i := 0 to AttrMap.Get_length-1 do begin
    AttrNode := AttrMap.get_item(i);
    Result := Result + AttrNode.get_nodeName + '='
                + AttrNode.get_nodeValue+' ';
  end;
end;
```

5.9.3 XML-Dateiformate

Dieses Kapitel zeigt, wie eine Delphi-Anwendung Daten in einer XML-Datei selbst definierten Formats speichern kann. Als Beispiel dient der TreeDesigner, der normalerweise binäre `tvf`-Dateien erzeugt, aber über das Menü DATEI | XML seine Dokumente auch in XML-Dateien speichern und daraus wiederherstellen kann (das Lesen einer TreeDesigner-XML-Datei ist nicht zu verwechseln mit dem Einlesen einer beliebigen XML-Datei und der Anzeige dieser Datei als TreeDesigner-Grafik!)

Hinweis: Da der TreeDesigner bereits über eine komplette Speicherfunktion für Binär-Dateien verfügt, die allen Einsatzzwecken des TreeDesigners gerecht werden sollte, bleibt der XML-Speicherfunktionen nur der Demonstrationszweck. Es wurden daher nicht die gesamten binären Speichermethoden auch für XML bereitgestellt, sondern nur die Methoden zum Speichern einfacher Elemente der Klasse *TGraphicElement* ohne zusätzlichen Bildinhalt. Beim Speichern eines Dokuments mit anderen Grafikelementen oder Inhalten in einer XML-Datei werden diese zusätzlichen Inhalte ignoriert; beim Laden der XML-Datei erscheint dann ein entsprechend »reduziertes« Grafikdokument.

Selbst definierte XML-Dateiformate *R110*

Auf die Vorteile der Speicherung von Daten im XML-Format wurde schon in Kapitel 5.9.1 eingegangen; kurz zusammengefasst zeichnen sich XML-Dateien dadurch aus, dass sie

- menschenlesbar (also nicht binär) sind,
- sich aufgrund der Standardisierung von XML zwischen beliebigen Arten von Computern und zwischen allen denkbaren Arten von Anwendungen übertragen lassen,
- erweiterbar unter Beibehaltung von Aufwärts- und Abwärtskompatibilität des Datenformats sind.

Die Definition eigener Dateiformate kann ganz sorgfältig über eine der ebenfalls in Kapitel 5.9.1 erwähnten DTDs erfolgen, jedoch ist eine solche DTD im Fall des TreeDesigners nicht notwendig. Die wichtigste Definition für das TreeDesigner-Dateiformat stellt daher die Festlegung der Namen für die XML-Elemente und die Attribute dieser Elemente dar. Sie erfolgt in der Unit *GrDoc* über einen Block von Konstanten. Im Programm-Code werden diese Namen nie direkt als Strings angegeben, sondern immer nur über Verwendung der Konstantenbezeichner wie *xmlAttrFontColor*. Dies stellt sicher, dass keine Tippfehler unbemerkt neue Namen kreieren, die dann nicht mehr mit dem anderen Code übereinstimmen (bei falsch geschriebenen Konstanten-Bezeichnern meldet ja der Compiler einen Fehler).

```
(******** Bezeichner für das TreeDesigner-XML-Dateiformat *********)
const
  (*** geschrieben von TGraphicDocument ***)
  xmlTypeGraphicDocument = 'TreeViewerDocument';
  xmlAttrDocVersion = 'Version';
  xmlGraphicElementOPClass = 'ObjectPascalClass';
  (*** geschrieben von TGraphicElement ***)
  xmlTypeGraphicElement = 'GraphicElement';
  xmlAttrShapeText = 'ShapeText';   xmlAttrFontName = 'FontName';
  xmlAttrLeft = 'Left';             xmlAttrRight = 'Right';
  xmlAttrTop = 'Top';               xmlAttrBottom = 'Bottom';
  xmlAttrShape = 'ShapeType';       xmlAttrPenColor = 'PenColor';
  xmlAttrBrushColor = 'BrushColor'; xmlAttrFontColor = 'FontColor';
  xmlAttrPenWidth = 'PenWidth';     xmlAttrFontHeight = 'FontHeight';
```

Abbildung 5.18 zeigt die grafisch geringfügig aufbereitete Darstellung einer TreeDesigner-XML-Datei; aus ihr können Sie die Anwendung der oben gezeigten Namen erkennen. Einige weitere Namen für die *Edge*-Elemente, die in der Abbildung auftauchen, wurden im obigen Listing der Übersichtlichkeit halber weggelassen; sie kommen auch in den folgenden Listings nicht vor.

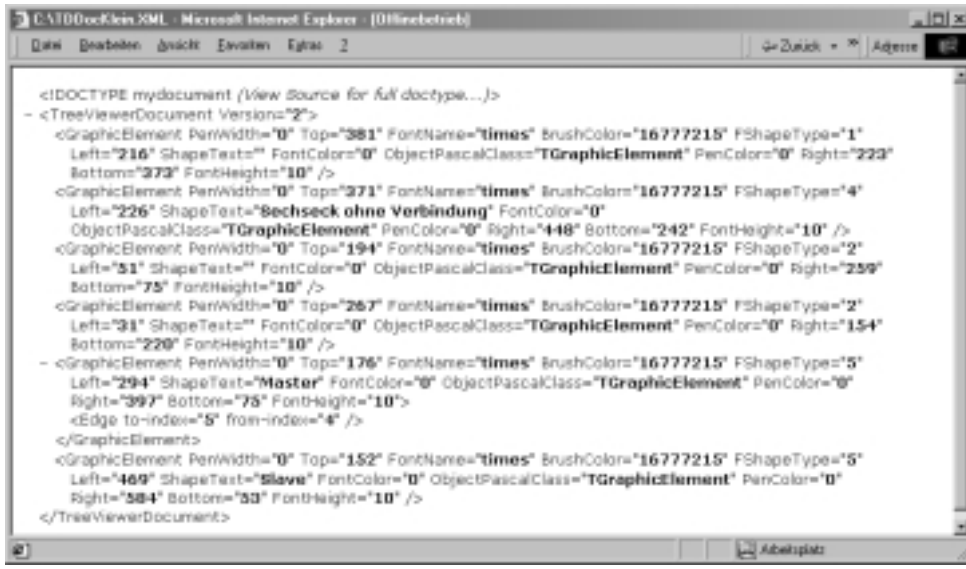


Abbildung 5.18: Um Verwirrung zu vermeiden, wird die TreeDesigner-XML-Datei hier nicht im TreeDesigner selbst als XML-Grafik angezeigt, sondern im Internet Explorer, der sich ebenfalls gut als XML-Dateibetrachter eignet.

Erzeugen einer XML-Datei

Da eine XML-Datei eine Textdatei ist, können Sie sie theoretisch auch wie eine normale Textdatei erzeugen, z.B. über die Standard-Pascal-Dateioperationen. Sie müssten sich dann selber darum kümmern, dass die XML-Syntax, also beispielsweise die Klammerungen und die Positionen der Tags, mit dem XML-Standard konform gehen.

Dies entfällt, wenn Sie das XML-Dokument im Speicher mit Hilfe des bereits in Kapitel 5.9.2 verwendeten *Document*-Objekts aufbauen und dieses dann beauftragen, sich selbst in eine Datei zu schreiben.

Das Dokumentobjekt wird im TreeDesigner wie in Kapitel 5.9.2 wieder über einen Aufruf von *CoDOMDocument.Create* erzeugt. Dieses Dokument ist zunächst vollkommen leer, enthält also noch keinen XML-Elementknoten. Ein solcher wird nun in der folgenden Methode mit der schon in Kapitel 5.9.1 erwähnten Methode *createElement* erzeugt. Als Parameter ist der Tag-Name des Elements anzugeben, im TreeDesigner wird dafür eine der oben definierten Konstanten (*xmlTypeGraphicDocument*) verwendet. Das neu erzeugte Element muss nun mit *appendChild* an einen bestehenden Knoten angehängt werden, wobei zu Anfang ja nur das Dokumentobjekt selbst als Elternknoten in Frage kommt.

Das so vorbereitete Dokumentobjekt kann nun gefüllt werden, wofür im TreeDesigner die separate und später besprochene Methode *SaveToXML* zuständig ist. Wir beginnen hier mit einem Listing der Methode *SaveToXMLFile*, die das Dokument-Objekt erzeugt, *SaveToXML* aufruft und damit endet, das fertige Dokument mit der *save*-Methode zu speichern:

```
procedure TGraphicDoc.SaveToXMLFile(FileName: String;
  OnlyMarked: boolean); //OnlyMarked: nur die markierten Objekte speichern
var
  DocElem: IDomElement;
  n: IDomNode;
  Doc: IDomDocument;
begin
  Doc := CoDOMDocument.Create;
  Doc.validateOnParse := True;
  Doc.async := False;
  DocElem := Doc.createElement(xmlTypeGraphicDocument);
  Doc.appendChild(DocElem);
  SaveToXML(DocElem, OnlyMarked);
  Doc.save(FileName);
end;
```

Dieses Listing verwendet wieder spezielle Elemente von MSXML: Außer den schon bekannten Properties *validateOnParse* und *async* ist es hier die *save*-Methode, die in der vom DOM vorgegebenen *IDomDocument*-Schnittstelle fehlt und für die Sie einen Ersatz suchen müssen, falls Sie eine andere DOM-Implementation als die von Microsoft verwenden sollten.

Speichern der Grafikelementdaten in einem XML-Element

Oben wurde bereits die Methode *SaveToXML* aufgerufen, die das gesamte Grafikdokument als Unterbaum an ein bereits vorbereitetes XML-Element anhängen soll. Da das Grafikdokument aus seinen einzelnen Grafikelementen besteht, werden wir nun vom Kleinen zum Großen schreiten und mit der *SaveToXML*-Methode für ein einzelnes Grafikelement beginnen. Auch sie erwartet im ersten Parameter ein XML-Element, das als Elternelement für die neu erzeugten XML-Elemente dienen soll.

Die erste Etappe im Arbeitsablauf der Methode bildet die Erzeugung eines neuen Knoten-Objekts. Dieses wird wie in der zuletzt gezeigten Methode mit *IDomDocument.createElement* erzeugt und über ein *IDomElement*-Interface angesprochen. Als Tag-Name dieses Elements wird *xmlTypeGraphicElement* gesetzt, so dass in der letztlich resultierenden XML-Datei dafür ein Block nach dem Muster

```
<GraphicElement ...>
</GraphicElement>
```

entsteht, wobei der Teil »...« noch durch die Attribute gefüllt wird. Diese Attribute werden von *SaveToXML* über mehrfache Aufrufe von *setAttribute* gesetzt:

```
function TGraphicElement.SaveToXML(ParentElem: IDomElement): IDomElement;
var
  Doc: IDomDocument;
  newElem: IDomElement;
  i : Integer;
begin
  Doc := ParentElem.ownerDocument;
  if Doc = nil then Doc := ParentElem as IDomDocument;
  newElem := Doc.createElement(xmlTypeGraphicElement);
  newElem.setAttribute(xmlAttrShapeText, FText);
  newElem.setAttribute(xmlAttrLeft, Points.Left);
  newElem.setAttribute(xmlAttrRight, Points.Right);
  newElem.setAttribute(xmlAttrTop, Points.Top);
  newElem.setAttribute(xmlAttrBottom, Points.Bottom);
  newElem.setAttribute(xmlAttrShape, integer(FShapeType));
  newElem.setAttribute(xmlAttrPenColor, Pen.Color);
  ... (Speichern weiterer Attribute und der Edge-Liste)...
  ParentElem.appendChild(newElem);
  Result := newElem;
end;
```

Ein Beispiel für ein auf diese Weise erzeugtes vollständiges XML-Element finden Sie in Abbildung 5.18.

XML-Schreibmethoden

Nun lässt sich die Schreibmethode für das gesamte Dokument auf den wiederholten Aufruf der Schreibmethode für ein einzelnes Element zurückführen. Als einziges Attribut des XML-Elements für das gesamte TreeDesigner-Dokument wird eine Versionskennung gesetzt, die jedoch im TreeDesigner bisher nicht weiter verwendet wird:

```
procedure TGraphicDoc.SaveToXML(XMLDocElem: IDomElement;
  OnlyMarked: boolean);
var
  i: Integer;
begin
  XMLDocElem.SetAttribute(xmlAttrDocVersion, XMLFileVersion);
  for i := 0 to Count-1 do
    if (not OnlyMarked) or Items[i].Marked then begin
      Items[i].SaveToXML(XMLDocElem);
    end;
end;
```

Hintergrund: Gegenüber der binären Schreibmethode *SaveToStream* muss hier nicht die Zahl der Elemente in die Datei geschrieben werden. Dafür werden im vollständigen Code die Elemente vor dem Speichern durchnummeriert, damit »Zeiger« von einem Element auf andere Elemente möglich werden (schließlich sollen auch die Verbindungen zwischen den Elementen gespeichert werden, was in der vollständigen Methode *TGraphicElement.SaveToXML* stattfindet).

Zurücklesen der XML-Datei

Aus Platzgründen können die Methoden zum Lesen einer TreeDesigner-XML-Datei nicht ebenfalls so ausführlich abgedruckt werden wie die Schreibmethoden. Zunächst muss wieder ein Dokument-Objekt erzeugt werden, das mit der *load*-Methode zum Einlesen und Parsen der Datei aufgefordert werden kann (siehe Kapitel 5.9.2). Dann wird der gesamte XML-Baum durchlaufen, um für jedes XML-Element `<GraphicElement>` ein *TGraphicElement* zu erzeugen. Dieses überträgt dann die Attribute des XML-Elements in seine eigenen Properties. In Analogie zu den oben gezeigten *setAttribute*-Aufrufen sind nun *getAttribute*-Aufrufe angesagt:

```
procedure TGraphicElement.LoadFromXML(Elem: IDomElement);
begin
  FText := Elem.getAttribute(xmlAttrShapeText);
  Points.Left := Elem.getAttribute(xmlAttrLeft);
  FShapeType := Elem.getAttribute(xmlAttrShape);
  Pen.Color := Elem.getAttribute(xmlAttrPenColor);
  ...
```

An dieser Stelle kann sich die große Flexibilität eines XML-Dokuments übrigens auch nachteilig auswirken: Auf die gerade skizzierte Weise können auch Dokumente gelesen werden, die mehr Informationen enthalten, als verwertet werden. Z.B. könnte die Datei von einer zukünftigen TreeDesigner-Version geschrieben worden sein und Attribute enthalten, die oben nicht mit *getAttribute* ausgelesen werden. Dass diese Datei auch von früheren Versionen des TreeDesigners verstanden werden kann, ist zunächst einer der Vorteile von XML.

Beim Speichern dieser Datei im XML-Format würden jedoch die zusätzlichen Informationen verloren gehen. Auch Kommentare, die vielleicht von einem Menschen in die XML-Datei eingefügt wurden, werden beim maschinellen Erzeugen der XML-Datei durch eine Anwendung nicht wiederhergestellt, es sei denn, die Anwendung liest die Kommentare aus den Element-Objekten des DOM aus und merkt sie sich (was theoretisch ja möglich wäre).

Für dieses Problem bieten sich unter anderem die beiden folgenden Lösungswege an:

- Zum Teil entsteht das Problem gar nicht erst, da viele XML-Dokumente nur als Nachricht zwischen Computern ausgetauscht werden und nach der Verarbeitung

der Nachricht beim Empfänger-Computer ohnehin gelöscht werden. (Das SOAP-Protokoll verwendet beispielsweise Nachrichten im XML-Format. Dabei sendet ein Sender ein XML-Dokument zu einem Empfänger, der unter Umständen nicht alle Informationen der Nachricht verwerten kann, jedoch als Antwort eine komplett neues XML-Dokument generiert.)

- Wenn XML-Dokumente für Benutzerdateien verwendet werden, können die Techniken verwendet werden, die auch schon vor der Erfindung von XML für Benutzerdokumente Anwendung fanden, z.B. Berücksichtigung der Dateiformat-Versionsnummer und Sicherheitsabfrage vor dem Überschreiben einer Datei in einem neueren Format; oder Warnhinweise am Anfang der Datei, dass keine manuellen Erweiterungen (wie Kommentare) hinzugefügt werden sollen, da diese beim Speichern vom Programm gelöscht werden (in Delphi finden Sie solche Warnungen z.B. in den Units, die beim Importieren von Typenbibliotheken erzeugt werden).

Der XML-Datenbindungs-Experte

Die obigen Listings könnten noch ein wenig vereinfacht werden, wenn wir die ab Delphi 6 zur Verfügung stehende Hilfe des XML-Datenbindungs-Experten in Anspruch nehmen würden. Wir bräuchten dazu lediglich eine bereits erzeugte TreeDesigner-XML-Datei als Muster.

Der Experte (zu finden unter DATEI | NEU | WEITERE | NEU | XML-DATENBINDUNG) liest eine solche Musterdatei ein und analysiert ihre XML-Struktur (siehe Abbildung 5.19), insbesondere die verwendeten Knotentypen, die darin vorkommenden Attribute und die Beziehungen zwischen den Knotentypen (dazu gehört etwa, dass die *Graphic-Element*-Knoten dem Dokumentelement untergeordnet sind und eine Liste von *Edge*-Knoten enthalten). Aus diesen Informationen fertigt der Experte neue Schnittstellendeklarationen an, über die Sie die Attribute und Unterelemente eines Knotens direkt ansprechen können (siehe die *Code-Vorschau* in Abbildung 5.19). Dabei können Sie zahlreiche Optionen wie etwa die Namen dieser Schnittstellen, die verwendeten Präfixe, Object-Pascal-Typen etc. einstellen.

Die von diesem Experten erzeugte Unit dient als Zwischenschicht zwischen den bisher direkt verwendeten DOM-Schnittstellen und dem Code Ihrer Anwendung. Diese Zwischenschicht benutzt außerdem die Delphi-Unit *XMLDoc*, die erst ab Delphi 6 verfügbar ist (dies ist die Unit, in der auch die früher einmal erwähnte *TXMLDocument*-Komponente enthalten ist).

Um die erhöhte Komplexität durch die Zwischen-Unit zu vermeiden und den Code kompatibel mit früheren Delphi-Versionen zu halten, verwendet der TreeDesigner die DOM-Schnittstellen direkt (wie oben gezeigt).

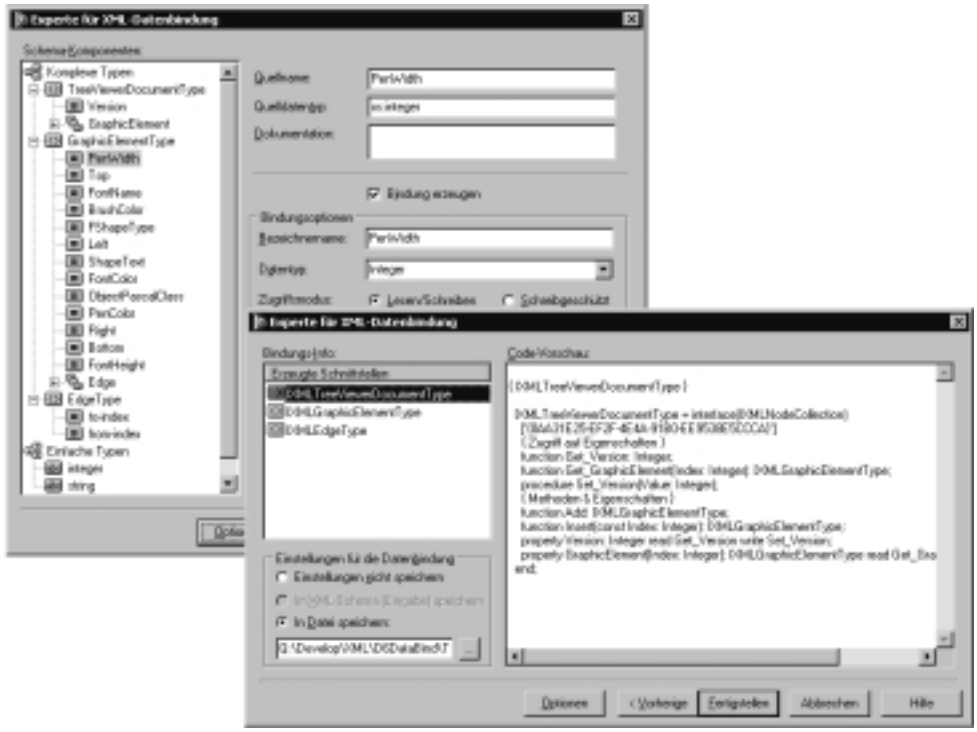


Abbildung 5.19: Nachdem der XML-Datenbindungs-Experte die Knotentypen und Attribute eines bestehenden TreeDesigner-XML-Dokuments analysiert hat (hinteres Fenster), schlägt er für jeden erkannten Knotentypen eine Object-Pascal-Schnittstellendeklaration vor (vorderes Fenster), mit der der direkte Aufruf der DOM-Schnittstellen vermieden werden kann.

Zum Abschluss sei jedoch demonstriert, wie sich der Code durch die Nutzung des XML-Datenbindungsexperten verändern würde – am Beispiel der oben gezeigten Methode *TGraphicElement.SaveToXML*, die für einen XML-Knoten ein Grafikelement erstellt. Zunächst werden die DOM-Schnittstellen ersetzt:

- Der Elternknoten wird nicht mehr als *IDomElement*, sondern als *IXMLTreeViewerDocumentType* angegeben.
- Der neu erzeugte Grafikelement-Knoten – früher ebenfalls als *IDomElement* angesprochen – wird nun durch ein *IXMLGraphicElementType* repräsentiert.

Um einen neuen Grafikelement-Knoten zu erzeugen, wird einfach die *Add*-Methode des Dokumentknotens aufgerufen; die Attribute werden nun über Properties des Elementknotens gesetzt. Im folgenden Listing sind vor diesen neuen Anweisungen auch die alten Anweisungen als Kommentar angegeben:

```
function TGraphicElement.SaveToXML(Doc: IXMLTreeViewerDocumentType)
                                    : IXMLGraphicElementType;
var
  newElem: IXMLGraphicElementType;
begin
//  newElem := Doc.createElement(xmlTypeGraphicElement);
    newElem := Doc.Add;

//  newElem.setAttribute(xmlGraphicElementOPClass, self.ClassName);
    newElem.ObjectPascalClass := self.ClassName;

//  newElem.setAttribute(xmlAttrShapeText, FText);
    newElem.ShapeText := FText;
  ...
```

Insbesondere fällt auf, dass durch die Verwendung der neuen Schnittstellen die String-Konstanten wie *xmlTypeGraphicElement* und *xmlAttrShapeText* nicht mehr angegeben werden müssen. Der weiter oben abgedruckte Deklarationsblock für diese Stringkonstanten wäre bei Verwendung des Datenbindungs-Experten nicht mehr erforderlich, da der Experte die Namen von Elementen und Attributen in der XML-Datei ja analysiert hat und sie fest in die neu erzeugte Unit einbaut.

Ein Ausschnitt aus der vom Experten erzeugten Unit soll dies verdeutlichen. Wenn wie im obigen Beispiel das Property *newElem.ShapeText* gesetzt wird, kommt es intern zu einem Aufruf der *Set_ShapeText*-Methode der Klasse *TXMLGraphicElementType*, welche die Schnittstelle *IXMLGraphicElementType* implementiert (diese Klasse arbeitet im Hintergrund, wird also im obigen Listing nur indirekt über das Interface angesprochen). Der XML-Datenbindungsexperte hat dort den Namen des *ShapeText*-Attributs fest verdrahtet:

```
procedure TXMLGraphicElementType.Set_ShapeText(Value: WideString);
begin
  SetAttribute('ShapeText', Value);
end;
```

Entsprechendes gilt auch für alle anderen Attribute. Die Namen der XML-Elemente (beispielsweise *xmlTypeGraphicElement*) werden vom Experten ebenfalls fest verdrahtet, jedoch sind die internen Abläufe hier komplizierter – sie seien dem an solchen Implementierungs-Details interessierten Leser zur eigenen Erforschung offen gelassen.

6 Komponentenentwicklung

Der Begriff der »Komponente« kann unter Delphi von verschiedenen Ebenen aus verstanden werden. Er kann sich speziell auf die Bestandteile von Delphis Komponentenpalette beziehen oder ganz allgemein auf sprachunabhängige Software-Komponenten.

Dieses Kapitel konzentriert sich auf die speziellere Bedeutung des Komponentenbegriffs, also auf die »VCL-Komponenten«, womit sowohl die mit Delphi mitgelieferten Komponenten als auch die selbst geschriebenen gemeint sein sollen. Es beschreibt das Komponentenkonzept der VCL und zeigt die internen Abläufe, auf denen VCL-Komponenten basieren, sowie die Techniken, mit denen sie entwickelt werden. Schließlich beschreibt es die Entwicklung von einigen Beispiel-Komponenten, die Sie sinnvoll in Ihren Anwendungen einsetzen können und die teilweise auch im TreeDesigner Verwendung finden.

Eine besondere Form sprachunabhängiger Komponenten lässt sich in Delphi aus den VCL-gebundenen Komponenten herstellen: Delphi kann (ab der Professional-Ausgabe) eine für die VCL entwickelte visuelle Komponente automatisch in ein ActiveX-Steuerelement konvertieren, das Sie dann in völlig anderen Entwicklungssystemen einsetzen können. Da ActiveX-Komponenten in Delphi weiterhin auf VCL-Komponenten basieren, gelten die meisten der in diesem Kapitel beschriebenen Grundlagen auch für die Entwicklung von ActiveX-Komponenten. Kapitel 6.8 wird wieder auf ActiveX zurückkommen und eine der Beispielkomponenten – die Farbpalette – in eine ActiveX-Komponente konvertieren.

Die Komponenten dieses Buchs

Nachdem Sie den Inhalt der CD zu diesem Buch auf der Festplatte installiert haben, haben Sie zwei verschiedene Möglichkeiten, die in diesem Kapitel entwickelten Komponenten zu installieren:

- Wählen Sie KOMPONENTE | PACKAGES INSTALLIEREN..., drücken Sie den Schalter HINZUFÜGEN... und fügen Sie das Package der Buch-Komponenten AWD6EW.bpl aus dem `Buch-Verzeichnis\complib` hinzu. Daraufhin fügt Delphi das Package in die Liste des Package-Installationsdialogs ein, selektiert und lädt es automatisch. Schließen Sie den Dialog daraufhin mit *Ok*. Die Komponenten werden auf der Seite

EW-Buch in der Komponentenpalette installiert, die Sie natürlich im Eigenschaftsdialog der Palette beliebig umbenennen können.

- Alternativ können Sie auch DATEI | ÖFFNEN wählen und den Package-Quelltext AWD6EW.dpk im Verzeichnis [Buch-Verzeichnis]\complib öffnen. Diese Vorgehensweise erlaubt Ihnen, sich das Package vor der Installation im Package-Editor genauer anzusehen. Drücken Sie in diesem Editor den Aktionsschalter INSTALL., um das Package in die Komponentenpalette aufzunehmen.

Für frühere Delphi-Versionen wählen Sie nach Anleitung der Readme-Datei auf der CD-ROM ein anderes Package bzw. einen anderen Installationsvorgang.

6.1 Delphis Komponentenkonzept

Delphis Komponentenkonzept erfordert zunächst einmal eine neue Begriffsunterscheidung: Die Bezeichnung *Benutzer* meint in diesem Kapitel normalerweise den Entwickler, der die fertigen Komponenten bei seiner Anwendungsentwicklung benutzt, und nicht den *Anwender*, der mit der fertigen Anwendung arbeitet. Über dem *Komponentenentwickler* befinden sich also zwei Ebenen von Benutzern, die in diesem speziellen Kapitel durch die etwas willkürliche Terminologie (*Komponenten-*)*Benutzer/(End-)Anwender* unterschieden werden sollen (wobei der Anwender in diesem Kapitel kaum vorkommt).

Komponenten bedeuten für den Benutzer und den Entwickler der Komponenten unterschiedliche Dinge:

- Für den Benutzer betont Delphi besonders stark die ereignisorientierte Programmierung, während die objektorientierte Programmierung etwas in den Hintergrund tritt. Der Benutzer braucht nicht sehr viel von OOP zu verstehen, da er nur innerhalb einer Klasse, der Formularklasse, arbeitet und von dieser aus die einzelnen Komponenten anspricht (im einfachsten Fall braucht der Benutzer noch nicht einmal das Klassenkonzept zu kennen, sondern kann die Klasse mit dem zur Laufzeit erscheinenden Fenster gleichsetzen). Außerdem stellen die Komponenten alle Ereignisse als Events im Objektinspektor zur Verfügung. Das Überschreiben von virtuellen Methoden ist nicht erforderlich, und interne Abläufe innerhalb der Komponenten bleiben dem Benutzer weitgehend verborgen.
- Für den Entwickler von Komponenten wird die objektorientierte Sichtweise des Programms noch weiter verstärkt, denn Komponenten können als Objekte mit besonders scharfen Umrissen angesehen werden. Natürlich sollten Objekte im OOP immer »deutliche Umrisse« haben, Delphis Komponentenkonzept fördert es jedoch besonders, dass Komponenten unabhängig von anderen Komponenten als eigenständige Software-Bausteine verwendbar sind. Der Komponentenentwickler

sollte sich gut mit der Sprache Object Pascal und mit deren OOP-Konzepten auskennen. (Zu den schon in Kapitel 2 beschriebenen Spracheigenschaften kommen in diesem Kapitel nur noch wenige hinzu.)

6.1.1 Das Wesen einer Komponente

Komponenten sind spezielle Objekte, die sich von normalen Objekten dadurch unterscheiden, dass Sie sie zur Entwurfszeit in das Formular einfügen und im Objektinspektor bearbeiten können. Dazu bedarf es keiner neuen Eigenschaften der Programmiersprache: Auch Komponenten nutzen lediglich die normalen Object-Pascal-Sprachelemente, insbesondere natürlich die Klassen und die Properties (allerdings kann kaum ein Zweifel daran bestehen, dass einige Merkmale von Object Pascal speziell zur Unterstützung der Komponenten eingeführt wurden).

Um den Weg in die Delphi-IDE zu finden, müssen Komponentenklassen mindestens zwei Voraussetzungen erfüllen:

- Sie müssen direkt oder indirekt von der Klasse *TComponent*, die in der VCL definiert ist (siehe Kapitel 3.1.2), abstammen. Die Vererbung sorgt dafür, dass Ihre eigenen von *TComponent* abgeleiteten Klassen von der Delphi-IDE genauso behandelt werden wie die von Delphi definierten Komponenten, die natürlich auch alle von *TComponent* abgeleitet sind.
- Nach diesem ersten Schritt verfügen Sie bereits über eine Komponente, die Sie zur Laufzeit wie jede andere Klasse verwenden können. Um die Verbindung zur Delphi-IDE herzustellen, damit die Komponente auch schon zur Formular-Entwurfszeit einsatzfähig ist, müssen Sie Ihre Komponentenklasse bei Delphi registrieren.

Obwohl Sie die fertigen Komponenten zur Entwurfszeit eines Formulars auf vielfältige Weise manipulieren können und der Objektinspektor genau über die Properties Bescheid wissen muss, genügt es, für die Registrierung jeder Komponente eine einzige Zeile Code zu schreiben, alles andere erledigt die Vererbung.

VCL-Klassen und die Komponenten

Alle Klassen auf dem Vererbungspfad der VCL von *TObject* bis *TComponent* spielen eine wichtige Rolle:

- Die Laufzeit-Typinformationen sind der Grund dafür, dass der Objektinspektor den Typ der Properties erkennt und diese in einer angemessenen Form darstellen kann. Diese Typinformationen sind schon in der Klasse *TObject* vorhanden, allerdings gehört diese Klasse weniger zur VCL, als vielmehr zur Sprache Object Pascal.
- *TPersistent* spielt eine wichtige Rolle beim Erzeugen der `dfm`-Formulardateien und ist somit einer der Schlüssel zur Rekonstruktion der Formulare zur Laufzeit.

- *TComponent* definiert die Verbindungen unter den Komponenten, sorgt also quasi für den Zusammenhalt zwischen Formular und Komponenten.
- *TWinControl* ist zwar nicht erforderlich, um das Komponentenkonzept durchzusetzen, ist aber die Schlüsselklasse, ab der eine Komponente richtig visuell wird, also als Fenster oder Steuerelement sichtbar wird. Für die automatische Erweiterung zu einem ActiveX-Steuerelement muss eine Komponente von dieser Klasse abgeleitet sein.

6.1.2 Überblick über die Komponentenentwicklung

Der Code von Komponenten nutzt normalerweise mehr Sprachelemente von Object Pascal, als die Teile der Anwendung, die Komponenten lediglich benutzen. Der grobe Ablauf der Komponentenentwicklung ist jedoch nicht komplizierter als der Ablauf zum Entwickeln einer normalen Delphi-Anwendung:

- Am Anfang steht hier nicht der visuelle Entwurf (es sei denn, die Komponente kapselt ein Formular, siehe Kapitel 6.7), sondern das Implementieren der Komponente in einer Unit. Hier können Sie genauso verfahren, als wollten Sie eine neue Objektklasse entwerfen, die Sie direkt im Programm nutzen wollen. (Die Klasse muss allerdings wie schon erwähnt direkt oder indirekt von *TComponent* abgeleitet sein.)
- Sobald diese Unit sich wie jede andere Unit übersetzen lässt, muss sie um wenige Zeilen Registrierungs-Code erweitert werden, durch den die Delphi-IDE sich Zugriff auf die Komponente verschaffen kann. Dadurch wird die Unit bereits zu einer fertigen Komponenten-Unit, die Sie beispielsweise mit KOMPONENTE | KOMPONENTE INSTALLIEREN... in die Palette aufnehmen können.

Der zweite Schritt kann auch etwas umfangreicher als wenige Zeilen ausfallen, wenn Sie weitere Schnittstellen der IDE ansprechen oder Editoren für Ihre Komponente oder für Properties registrieren wollen.

Die Schnittstelle zu Delphi

Die minimale Schnittstelle zu Delphi besteht aus einer Prozedur wie der folgenden:

```
procedure Register;
begin
  RegisterComponents('Zum Buch', [THotkeyManager]);
end;
```

Sie registriert eine Komponente für die Komponentenpalette der IDE. Zusätzlich stehen Ihnen die folgenden weiteren Optionen zur Verfügung:

- Für die Registrierung von Editoren für Properties oder andere, nicht im Objektinspektor sichtbare Eigenschaften der Komponente gibt es weitere Registrierungsmethoden.
- Um Ihre Komponente in der Komponentenpalette durch eine eigene Bitmap darzustellen, legen Sie eine Ressourcen-Datei im DCR-Format an.
- Um Online-Hilfstexte zur Verfügung zu stellen, wie sie für die vorgefertigten Komponenten bestehen, schreiben Sie unter Berücksichtigung zusätzlicher Konventionen eine Hilfedatei.
- Bei dem Vorhaben, in Ihrem Programm mit der Delphi-IDE Verbindung aufzunehmen, stehen Ihnen Schnittstellen des *Open Tools API* zur Verfügung (ab der Professional-Ausgabe von Delphi).
- Mit den letztgenannten *Open Tools*-Schnittstellen lassen sich auch unabhängig von der Komponentenentwicklung die verschiedensten Erweiterungen der Delphi-IDE programmieren. Mehr zu diesem Thema können Sie in Anhang A erfahren.

Die genannten Themen werden ausführlicher in den Kapiteln 6.4 und 6.6.5 besprochen. Lediglich die Hilfedateien können hier nicht besprochen werden, denn wie diese erzeugt werden, hängt davon ab, welches der unzähligen auf dem Markt angebotenen Tools Sie verwenden (und die Verwendung eines solchen Tools ist normalerweise sehr zu empfehlen).

6.1.3 Das Package-Konzept

Das Package-Konzept ist für die Komponentenentwicklung von großer Bedeutung. Da Packages im Normalfall die meiste Zeit im Hintergrund wirken und Sie nur bei der Entwicklung und der Installation von Komponenten zwingend mit Packages und den damit verbundenen Erweiterungen der Delphi-IDE in Berührung kommen, wurde es im bisherigen Verlauf des Buchs noch nicht erläutert. Trotzdem können Sie aus den Packages auch für normale Anwendungen gewichtige Vorteile ziehen, siehe hierzu den abschließenden Abschnitt zu den *Laufzeit-Packages*.

In den ersten beiden Versionen von Delphi war die Komponentenbibliothek sehr einfach organisiert: Alle Komponenten, die in der IDE installiert wurden, fügte Delphi zu einer einzigen großen Datei hinzu, die schon in der Basisausstattung fast 1,5 MByte groß war[15]. Die einzelnen, zur Bibliothek gehörenden Units konnten nur statisch zu den ausführbaren Dateien von Anwendungen hinzugebunden werden.

Seit Delphi 3 besteht die Komponentenbibliothek aus mehreren getrennten Bibliotheken, den *Packages*. Eine Übersicht über alle zurzeit in der IDE installierten Packages

15 Dies war vor fünf Jahren noch eine ganze Menge. Nicht auszudenken, wie riesig diese Datei wäre, wenn auch jetzt noch alle Komponenten darin gespeichert werden würden.

erhalten Sie mit KOMPONENTE | PACKAGES INSTALLIEREN; wählen Sie aus der Liste ein Package aus, können Sie sich mit dem Schalter *Komponenten* auch noch eine Übersicht über die in diesem Package enthaltenen Komponenten anzeigen lassen.

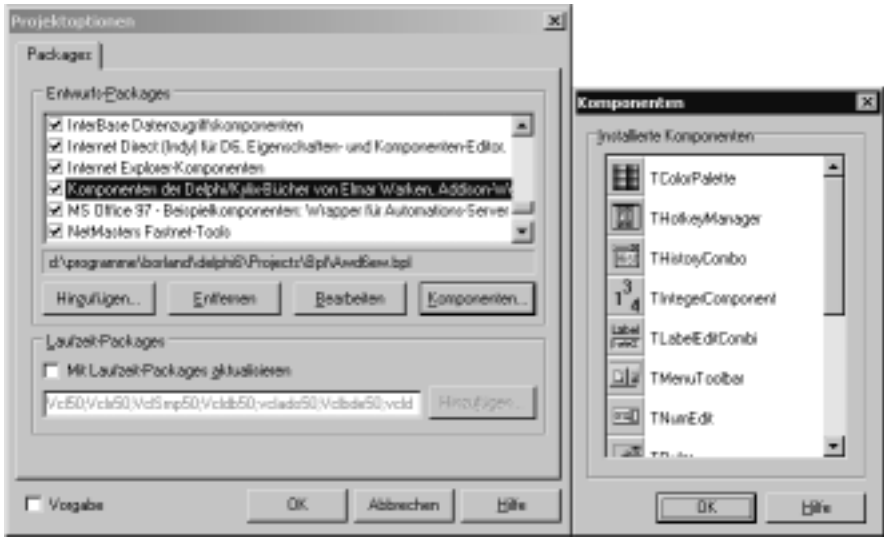


Abbildung 6.1: Der Dialog zur Wahl der aktiven und zum Installieren neuer Packages

Laufzeit-Packages

So wie Delphi selbst beim Entwurf der Formulare dynamisch geladene Packages verwendet, können auch Sie die Komponenten dynamisch zu Ihren Anwendungen binden lassen. Per Voreinstellung erzeugt Delphi EXE-Dateien, in die alle benötigten Komponenten direkt eingebunden werden. Wenn Sie aber in den Projektoptionen die Option MIT LAUFZEIT-PACKAGES COMPILIEREN einschalten, erzeugt Delphi eine EXE-Datei, die Laufzeit-Packages verwendet. Für die Weitergabe einer Anwendung bedeutet das einen etwas größeren Aufwand, da neben der ausführbaren Datei auch noch die gerade benötigten Laufzeit-Packages weitergegeben werden müssen; dem stehen aber auch Vorteile gegenüber:

- Bei sehr kleinen Programmen mit z.B. nur einem Formular und hundert Zeilen Code wird die Größe der ausführbaren Datei drastisch reduziert: Statt mit 300 – 400 KByte können Sie nun eher mit 20 – 60 KByte rechnen, die nur die Formulare Ihrer Anwendung und den von Ihnen geschriebenen Code enthalten.
- Haben Sie viele kleine Anwendungen, ergibt sich auf der Festplatte eine deutliche Platzersparnis, da die Laufzeit-Packages nur einmal für alle Anwendungen gespeichert werden müssen.

- Laufen mehrere solcher Anwendungen gleichzeitig auf einem System, gilt dasselbe auch für den Verbrauch an Hauptspeicher, denn in Windows werden mehrfach benutzte DLLs nur einmal geladen.

In Kapitel 6.2.3 kommen wir wieder auf die Laufzeit-Packages zurück; dort werden wir uns auch den Aufbau eines solchen Packages genauer ansehen.

Entwurfszeit-Packages

Entwurfszeit-Packages sind spezielle Packages zur Einbindung der Komponenten in die Komponentenpalette der Delphi-IDE. Im einfachsten Fall kann die IDE dasselbe Laufzeit-Package verwenden wie auch Ihre Anwendung. Wenn aber in der Delphi-IDE zusätzliche Funktionalität installiert werden soll, die im normalen Laufzeit-Package unnötig ist, bietet es sich an, eine spezielle Package-Version als Entwurfszeit-Package zu erstellen. So bestehen die zum Lieferumfang von Delphi gehörenden Entwurfszeit-Packages (deren Namen zumeist mit dem Präfix *dcl* beginnen) unter anderem aus den Komponenten- und Property-Editoren, die Sie in der IDE zu den verschiedenen Komponenten aufrufen können. Die Entwurfszeit-Packages enthalten aber normalerweise *nur* diese zusätzliche Funktionalität, darum muss die IDE neben den Entwurfszeit-Packages auch noch die normalen Laufzeit-Packages laden. Intern ist dies dadurch automatisiert, dass die Laufzeit-Packages in der *requires*-Klausel (siehe Kapitel 6.2.3) der Entwurfszeit-Packages angegeben sind.

Ein Vorteil der Verwendung von Packages auch für die Delphi-IDE liegt in der erheblich schnelleren Installation und Deinstallation von Komponenten und Komponentengruppen in die Komponentenpalette der IDE. Statt jedes Mal die komplette Komponentenbibliothek neu zu kompilieren, genügt es nun, einen kleinen Bestandteil davon als Package in den Speicher zu laden oder aus ihm zu entfernen. Im schon erwähnten Package-Installationsdialog brauchen Sie nur das Markierungsfeld neben einem Package anzuklicken, um das Package zu aktivieren oder zu deaktivieren.

Hinweis: Durch das Deaktivieren nicht benötigter Packages kann die Ladezeit der Delphi-IDE erheblich beschleunigt werden.

6.2 Beispiele und Installation

Bevor wir in die Details der Komponentenentwicklung einsteigen, soll dieses Kapitel ein sehr einfaches Beispiel geben, das bereits aus einer sinnvoll einsetzbaren Komponente besteht. Das erste Beispiel soll jedoch eine Minimalkomponente sein.

6.2.1 Starthilfe durch den Komponenten-Experten

Den in Kapitel 6.1.2 beschriebenen Grundlagen folgend benötigen wir zuerst eine neue Unit, die die Komponente beherbergen soll. Um das Grundgerüst dieser Unit nicht selbst schreiben zu müssen, geben wir ein solches beim Komponenten-Experten in Auftrag, der sich in der Objektablage auf der Seite *Neu* unter dem Namen *Komponente* bzw. hinter dem Menüpunkt KOMPONENTE | NEUE KOMPONENTE verbirgt.

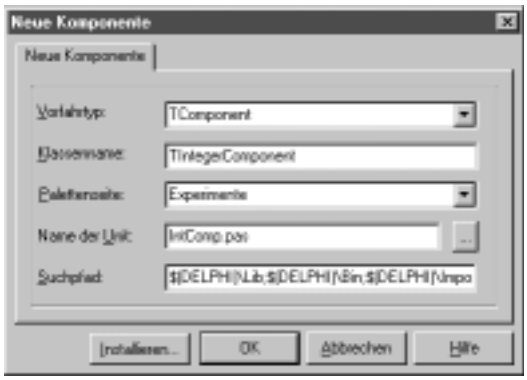


Abbildung 6.2: Die Expertendaten für die erste Beispielkomponente

Die Schnittstelle des Experten besteht aus der in Abbildung 6.2 gezeigten Dialogbox, die Sie zu fünf Angaben auffordert:

- Im ersten Eingabefeld geben Sie die Basisklasse der neuen Komponente an,
- im darunter befindlichen Feld den Namen der neuen Komponentenklasse,
- im dritten Feld eine Seite der Komponentenpalette, in die die neue Komponente bei der später erfolgenden Installation eingetragen werden soll.
- Unter NAME DER UNIT können Sie der neuen Komponente bereits eine Speicherposition zuweisen, was den Vorteil hat, dass der Bibliothekssuchpfad (unterstes Dialogfeld) automatisch erweitert wird, wenn Sie die Komponente in einem Verzeichnis speichern, das noch nicht in diesem Pfad enthalten ist.
- Das Feld SUCHPFAD ist quasi ein Spiegel des in den Umgebungsoptionen eingestellten Suchpfades. Änderungen dieses Feldes wirken sich nicht auf die erzeugte Komponenten-Unit, sondern nur auf die Umgebungsoptionen aus.

Durch Drücken von OK starten Sie die Expertentätigkeit, die sich jedoch sehr schnell in einem kleinen Gerüst einer noch unbenannten Unit erschöpft, das einen Klassenrumpf und eine Registrierungsroutine enthält. Ein solches, um zwei Zeilen erweitertes Unit-Gerüst finden Sie im nächsten Abschnitt.

Die Angaben, die Sie in der Dialogbox gemacht haben, sind nur ein- bzw. zweimal in der neuen Unit aufgeführt. Da Delphi die Eingaben in die Dialogbox schnell wieder vergisst, macht es nichts aus, wenn Sie Klassenname, Basisklasse und Palettenseite nachträglich ändern.

6.2.2 Eine Minimalkomponente

Eine Minimalkomponente eignet sich sehr gut dazu, in der Praxis zu erfahren, welche Entwurfszeit-Fähigkeiten selbst die einfachste von *TComponent* abgeleitete Klasse von ihrer Basisklasse erbt. Die wirklich minimale Komponente besteht zwar aus dem leeren Gerüst, wie es vom Komponenten-Experten erstellt wird, unsere erste Komponente soll jedoch wenigstens ein eigenes Property besitzen.

Der folgende Code der Unit *IntComp* wurde geschrieben, indem das vom Komponenten-Experten erstellte Gerüst um die beiden mit »(**)« markierten Zeilen erweitert wurde. :

```
unit Intcomp;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs;

type
  TIntegerComponent = class(TComponent)
  private
    { Private-Deklarationen }
  protected
(**)    FValue: LongInt;
    { Protected-Deklarationen }
  public
    { Public-Deklarationen }
  published
    { Published-Deklarationen }
(**)    property Value: LongInt read FValue write FValue;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Zum Buch', [TIntegerComponent]);
end;

end.
```

TIntegerComponent soll dazu dienen, im Property *Value* eine Zahl zu speichern. Sie benötigt keine Methoden, um das Property zu lesen oder zu schreiben, sondern gibt dem Compiler mit den *read*- und *write*-Direktiven an, dass er Lese- und Schreibzugriffe auf das Property direkt an die Variable *FValue* weiterleiten soll.

An sich besteht also kein Unterschied zwischen der Variablen und dem Property. Um jedoch Werte einer Komponente für den Objektinspektor zugänglich zu machen, ist es immer notwendig, ein Property dafür zu deklarieren, selbst wenn sich dahinter nur eine Variable verbirgt.

TIntegerComponent in der Praxis

TIntegerComponent wird automatisch mit allen anderen Komponenten dieses Buchs über installiert, wie in Kapitel 6.2.2 beschrieben. Wie jede andere Komponente finden Sie sie jedoch nicht nur in der Komponentenpalette, sondern auch in der *Komponentenliste* (Ansichtsmenü).

Sie können mehrere *TIntegerComponent*-Exemplare in ein Formular einfügen und deren *Value*-Property zur Entwurfszeit ändern. Durch den Standard-Konstruktor von *TObject* wird dieses Property (bzw. die zugrunde liegende Variable *FValue*) vorher immer mit 0 initialisiert. Jedes Exemplar von *TIntegerComponent* wird zur Entwurfszeit bereits als Icon dargestellt und verfügt über die geerbten Properties *Name* und *Tag*. Auch das Speichern des neuen Properties in der Formulardatei läuft automatisch ab.

Zur Laufzeit lässt sich die (dann unsichtbare) Komponente kaum zu mehr verwenden als eine normale Integer-Variable.

6.2.3 Installation von Packages und Komponenten

Zur Installation von Komponenten haben Sie in Delphi zwei Punkte im Menü KOMPONENTE zur Auswahl:

- Mit PACKAGES INSTALLIEREN gelangen Sie in den Dialog, in dem Sie die schon installierten Packages aktivieren und deaktivieren können (ab Delphi 3, siehe Abbildung 6.1). Dort können Sie auch neue Komponenten installieren, sofern diese schon als Packages vorliegen.
- In allen Delphi-Versionen gibt es den Menüpunkt KOMPONENTE INSTALLIEREN.... Hier geht es darum, eine Unit auszuwählen, die in der Komponentenpalette aufgenommen werden soll. Da Sie jedoch seit Delphi 3 nur Packages installieren können, müssen Sie neben dem Namen der zu installierenden Unit auch den Namen eines Packages angeben, in das die Unit aufgenommen werden soll (Abbildung 6.3). Sie haben hier die Wahl, ein neues Package zu erzeugen oder ein bestehendes Package zu erweitern, wobei Delphi Ihnen hierzu das zunächst einmal leere Package DCLUSR60.DPK aus seinem LIB-Verzeichnis vorschlägt. Auf diese Weise gelangen Sie

in den Package-Editor, in dem es jedoch wahrscheinlich nichts zu editieren gibt, da Delphi Ihnen normalerweise bereits anbietet, alle erforderlichen Schritte wie Neukompilieren und Installieren selbst auszuführen.

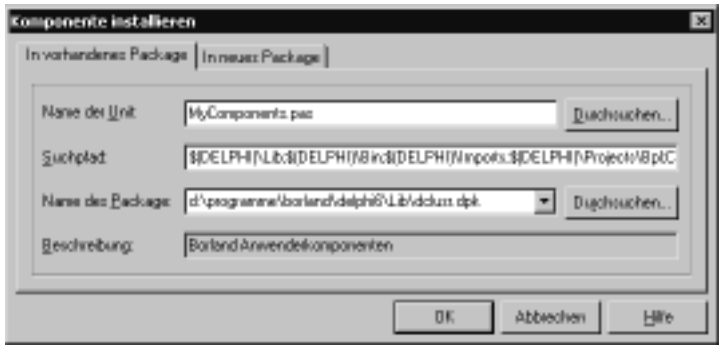


Abbildung 6.3: Installieren neuer Komponenten in das von Delphi vorbereitete Package dcluser.dpk

Der Package-Editor

Delphi verfügt über einen Package-Editor, in dem Sie alle zu einem Package gehörenden Daten verändern können, sofern Sie das Package selbst erzeugt haben oder zumindest eine DPK-Datei für das Package besitzen (Abbildung 6.4). Zu einem bestehenden Package können Sie diesen Editor entweder über den normalen *Datei-öffnen*-Dialog oder im schon in Abbildung 6.1 gezeigten Package-Installationsdialog über den Schalter *Bearbeiten...* öffnen. Für ein neues Package finden Sie in DATEI | NEU auf der ersten Seite einen passenden Eintrag.

Das in Abbildung 6.4 gezeigte Package enthält die Komponenten zu diesem Buch. Jede Komponente wurde in einer eigenen Unit implementiert, die Sie in der Liste unter dem *Contains*-Knoten finden und die mit dem Schalter HINZUFÜGEN in das Package aufgenommen wurde. Einige Komponenten verwenden noch weitere Units, die ebenfalls in das Package übernommen wurden (z.B. *HList* für das interne *THistoryList*-Objekt der Komponente *THistoryCombo* und *ColorPalEdit* für den Property-Editor der Farbpalette). Auch wenn Sie eine vom Package verwendete Unit nicht explizit in die *Contains*-Liste eintragen, wird sie vom Compiler automatisch zum Package hinzugefügt; allerdings erhalten Sie dann eine Warnung, dass die Unit nicht explizit in das Package aufgenommen wurde.

Die *Contains*-Liste in Abbildung 6.4 enthält neben Units auch noch eine DCR-Datei. Diese enthält Bitmaps für die einzelnen Komponenten und wird in Kapitel 6.4.4 erläutert werden.

Abbildung 6.4: Der Package-Editor von Delphi beim Installieren der Komponenten dieses Buchs

Der Inhalt eines Packages

Bei einer DPK-Datei handelt es sich um den menschenlesbaren Quelltext eines Packages, der von Delphi in eine binäre BPL-Datei (Borland Package Library) übersetzt wird, die dann in die Delphi-IDE bzw. in Ihre Anwendung eingebunden wird. Für die Anwendung des Package-Editors kann es hilfreich sein, einen solchen Package-Quelltext zu kennen. Für das Package der Komponenten dieses Buchs sieht der Quelltext wie folgt aus (gekürzt):

```
package Awd6ew;

{$R *.res}
{$R 'Allcomps.dcr'}
{$ALIGN ON}
{$ASSERTIONS ON}
... weitere Compiler-Anweisungen wurden hier gekürzt ...
{$DESCRIPTION 'Komponenten des Delphi-Buchs von Elmar Warken'}
{$IMPLICITBUILD ON}

requires
  vcl60;

contains
  Intcomp in 'Intcomp.pas',
  NumEdit in 'NumEdit.pas',
  Rulers in 'Rulers.pas',
```

```
  ColorPal in 'ColorPal.pas',
  ColorPalEdit in 'ColorPalEdit.pas' {PalEditForm},
  ColorPalDesignTime in 'ColorPalDesignTime.pas',
  ... Liste weiterer Units ...
  MenuToolBar in 'MenuToolBar.pas';
```

Jeder Bestandteil dieses Quelltextes lässt sich durch den Package-Editor festlegen:

- Die bereits erwähnten eingebundenen Units unter dem Knoten *Contains* bzw. im *contains*-Abschnitt des Package-Quelltexts.
- Auch zum Knoten *Requires* im Package-Editor gibt es im Package-Quelltext eine direkte Entsprechung. Dieser Abschnitt nennt alle Packages, die zum Betrieb des aktuellen Packages vorausgesetzt werden. Anwendungen, die dieses Package benutzen, müssen auch alle in der *requires*-Klausel aufgeführten Packages laden. Normalerweise benötigt jedes Package zumindest die Standard-Packages, die den Kern der VCL und den Grundvorrat an Komponenten bereitstellen (in Delphi 6 sind dies die Packages *rtl* und *vcl*). Entwurfszeit-Packages, die auf die Funktionen der Delphi-IDE zugreifen wollen, müssen seit Delphi 6 außerdem das Package *designide* per *requires*-Klausel einbinden. Sie finden die genannten und alle weiteren mit Delphi mitgelieferten Packages übrigens im *Lib*-Verzeichnis der Delphi-Installation.
- Die Compiler- und Linkeroptionen, die bei der Übersetzung der Units verwendet werden sollen, können Sie in einem Dialog einstellen, wenn Sie den Schalter *Optionen* im Package-Editor drücken. Die Seiten dieses Dialogs stimmen zum größten Teil mit dem Dialog der Projektoptionen überein.
- Über die üblichen Projektoptionen hinaus gehen die Einstellungen auf der ersten Seite dieses Dialogs. Hier genügt es meistens schon, wenn Sie eine Beschreibung für Ihr Package eintragen, für das Package der Buchkomponenten wurde jedenfalls nur diese Beschreibung geändert.

Die Aktionsschalter des Package-Editors

Eine sehr angenehme Einrichtung sind die Schalter zum Kompilieren und Installieren eines Packages:

- Mit dem Installieren-Schalter fügen Sie ein Package in die Komponentenpalette ein. Dadurch, dass die meisten Packages relativ klein sind, geht das erheblich schneller als zu den Zeiten vor dem Package-Konzept.
- Mit dem Kompilieren-Schalter übersetzen Sie ein Package nicht nur neu – falls es sich um ein Package handelt, das bereits in der Komponentenpalette installiert ist, installiert Delphi das neu kompilierte Package automatisch in der Komponentenpalette. Auf diese Weise können Sie Verbesserungen und Fehlerkorrekturen in Ihren Komponenten sehr schnell in die Palette übernehmen.

Sie können ein im Package-Editor angezeigtes Package wie ein »Nebenprojekt« ansehen, das Sie gleichzeitig mit einem »Hauptprojekt« (dem Projekt, das in der Titelzeile der IDE angezeigt wird) bearbeiten können. Mit START | START können Sie so z.B. jederzeit die Hauptprojekt-Anwendung starten und ohne das aktuelle Projekt zu schließen, können Sie auch das Package neu übersetzen, indem Sie im Package-Editor den Kompilieren-Schalter drücken.

Hinweis: Achten Sie beim Neu-Kompilieren eines Packages auf eventuelle Compiler-Fehlermeldungen im Editorfenster. Unter besonderen Umständen (wenn die IDE z.B. zweimal geöffnet ist und das neu kompilierte Package installiert ist) kann Delphi eine Package-Datei nicht neu erzeugen und außer einer versteckten Meldung im Editorfenster deutet nichts darauf hin, dass die Komponentenpalette noch nicht aktualisiert wurde.

Probleme mit fehlenden Komponenten

Sollte die Installation eines Packages aus irgendeinem Grund fehlschlagen, vergessen werden oder sollten Sie ein Package absichtlich deinstallieren, können Sie Formulare, die Komponenten aus diesem Package enthalten, nicht mehr vollständig laden.

Delphi zeigt Ihnen in diesen Fällen eine Meldung und lässt Sie wählen, ob das Laden des Formulars abgebrochen oder ob der Fehler ignoriert werden soll. Beim Ignorieren werden die nicht gefundenen Komponenten aus dem Formular entfernt, so dass Sie diese Option nur wählen sollten, wenn Sie diese Komponenten nicht mehr brauchen. Wenn Sie die Komponenten auf diese Weise löschen, bietet Delphi Ihnen beim nächsten Speichern der Formulardatei übrigens an, auch die überzähligen Komponentendeklarationen aus der Klassendeklaration des Formulars zu entfernen.

6.2.4 Eine sinnvolle Beispiel-Komponente

Unsere zweite Beispielkomponente wird eine visuelle Komponente sein, die eine sehr leichte, aber sinnvolle Aufgabe zu erfüllen hat. *TNumEdit* soll eine spezialisierte Version des Eingabefelds *TEdit* sein, die nur für die Eingabe von Integer-Zahlen gedacht ist. Wenn Sie eine Zahleneingabe aus einem *TEdit* oder *TMaskEdit* auslesen und im Programm auch als Zahl verwenden wollen, müssen Sie den String *T(Mask)Edit.Text* zuerst in einen Zahlenwert umwandeln, beispielsweise mit der Standardprozedur *StrToInt*. Diese Arbeit soll *TNumEdit* automatisch erledigen, und zwar unter Verwendung eines *LongInt*-Properties mit dem Namen *Number*.

Benutzung von TNumEdit

Wenn Sie *Number* einen Wert zuweisen, änden Sie auch den Textinhalt des Editierfelds. Umgekehrt erhalten Sie die Zahl, die dem Textinhalt entspricht, indem Sie *Number* auslesen:

```
{ Auslesen der Zahl }
  EingegebeneZahl := NumEdit1.Number;
{ Setzen von NumEdit.Text auf '10' }
  NumEdit1.Number := 10;
```

Wichtig ist dabei, dass *Number* immer dem geerbten Property *Text* entspricht. Wenn Sie also *Number* ändern, passt *TNumEdit* das Property *Text* automatisch an. Ändern Sie dagegen *Text*, so merkt das Property *Number* dies zwar nicht sofort, wenn Sie es aber abfragen, verwendet die Lesemethode für *Number* in jedem Fall den neuesten *Text*-Wert als Grundlage für die String-Konvertierung. Sie erhalten also mit *Number* immer den aktuellen Textwert. Die Properties *Number* und *Text* sind damit nach außen hin vollständig synchron.

Implementierung

TNumEdit ist von der Klasse *TEdit* abgeleitet und erweitert diese um das schon erläuterte Property sowie um die beiden Zugriffsmethoden, die die Konvertierung übernehmen:

```
unit Numedit;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, StdCtrls;

type
  TNumEdit = class(TEdit)
  protected
    function AsInt: LongInt;
    procedure SetInt(i: LongInt);
  published
    property Number: LongInt read AsInt write SetInt;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Zum Buch', [TNumEdit]);
```

```
end;

function TNumEdit.AsInt: LongInt;
begin
  try
    Result := StrToInt(Text);
  except
    on EConvertError do
      Result := 0;
  end;
end;

procedure TNumEdit.SetInt(i: LongInt);
begin
  Text := IntToStr(i);
end;

end.
```

Während *AsInt* den String mit Hilfe der oben schon erwähnten Prozedur *StrToInt* zu einem *LongInt* macht, nutzt *SetInt* eine zweite Standardprozedur. Wenn sich der Text im Eingabefeld nicht in eine Zahl umwandeln lässt, erzeugt *StrToInt* eine Exception, die von *TNumEdit* jedoch wieder unter den Teppich gekehrt wird – *TNumEdit* zeigt dem Benutzer also keine Meldung über die Exception an, in der Annahme, dass dieser selbst sieht, dass es sich bei seiner Eingabe nicht um eine Zahl handelt. Statt dessen gibt *TNumEdit* bei einer falschen Eingabe den Wert 0 zurück (ohne diesen aber in den Text des Editierfelds zu schreiben).

Über eine kleine Erweiterung einer bestehenden Komponentenklasse können Sie also sehr schnell zu einer neuen Komponente gelangen, die voll funktionsfähig in die Delphi-IDE eingebunden ist. Von dieser Basis ausgehend können Sie immer weitere Änderungen vornehmen wie Änderung des Paletten-Bitmaps, Hinzufügen von Properties, Registrieren von Property-Editoren usw. Kapitel 6.5 wird sich mit etwas größeren Erweiterungen bestehender Komponenten befassen.

6.3 Die Schnittstelle zum Benutzer

Die Schnittstelle einer Komponente zum Benutzer (mit »Benutzer« ist auch hier der Entwickler gemeint, der die Komponente verwendet) besteht zur Entwurfszeit aus den Properties und den Events, die im Objektinspektor editiert werden können. Im Programmcode gehören auch die Methoden zu dieser Schnittstelle, sie brauchen hier jedoch nicht mehr beschrieben zu werden, da dies für alle Klassen inklusive Komponenten bereits in Kapitel 2.5 geschehen ist.

6.3.1 Properties

Die Benutzung der Properties und die allgemeine Funktionsweise mit *read* und *write*-Methoden ist in Kapitel 2.2.5 beschrieben, für diesen Abschnitt bleibt eine Zusammenfassung der eher bei Komponenten anzutreffenden Property-Themen. Zunächst haben Sie durch Variationen des Zugriffsschutzes und der *read*- und *write*-Methoden die Wahl zwischen den folgenden grundlegenden Property-Typen:

- *Read/Write-Properties* sind die »normalen« Properties, die sowohl gelesen als auch geschrieben werden können.
- *Read only-Properties*: Properties, die nur zum Lesen geeignet sind, verfügen zwar über eine *read*-Methode, enthalten aber keine *write*-Direktive in ihrer Deklaration. Viele Nur-Lesen-Properties geben feste Tatsachen an, die nicht ohne weiteres geändert werden können, z.B. die Liste der am Bildschirm verfügbaren Schriftarten (*TScreen.Fonts*). Das Lesen solcher Properties kann auch mit der Ausführung komplizierterer Funktionen verknüpft sein.
- *Write only-Properties*: Properties, die nicht gelesen werden können, sind relativ selten, sie dienen eher dazu, Aktionen auszulösen, als Werte zu speichern. In Kapitel 5.3.2 finden Sie mit den Properties von *TGraphicDocument* ein Beispiel dafür.
- *Runtime-Properties*: Dies sind Properties, die Sie nicht im Objektinspektor editieren können, sondern nur zur Laufzeit. Ein solches Property erhalten Sie, wenn Sie es nicht als *published*, sondern als *public* deklarieren. Dieser Property-Typ kann natürlich mit dem Nur-Lesen- oder Nur-Schreiben-Typ kombiniert werden.

Überschreiben von Properties

Eine Eigenschaft der Properties, die sie von den Methoden und Daten eines Objekts unterscheidet, ist, dass Sie die Zugriffsbeschränkung in abgeleiteten Klassen ändern können. Die VCL-Klassen machen dies vielfach vor: So enthält beispielsweise die Klasse *TControl* eine Reihe von Properties, die als *protected* deklariert sind. Es steht den abgeleiteten Klassen frei, diese geschützten Properties zu veröffentlichen.

Dazu müssen Sie diese erneut in der Klassendeklaration erwähnen, allerdings nur mit Nennung ihres Namens. So veröffentlicht beispielsweise die Klasse *TColorPalette* das Property *DragMode* mit einer Zeile in ihrem *published*-Abschnitt:

```
published
  ...
  property DragMode;
```

Dieser Vorgang wird als *Überschreiben* des Properties bezeichnet. Sie können einem Property beim Überschreiben auch neue *read*- und *write*-Methoden zuweisen, indem Sie diese wie gewohnt hinter dem Property-Namen angeben. Auch die in Kapitel 6.4.3 beschriebenen weiteren Property-Direktiven *stored*, *default* und *nodefault* sind überschreibbar.

Hinweis: Das Zuweisen einer neuen *read*- oder *write*-Methode hat nicht dieselbe Wirkung wie das Überschreiben einer virtuellen Methode der Basisklasse, denn die Basisklasse ruft grundsätzlich immer die in ihrer eigenen Deklaration definierten Zugriffsmethoden auf. Diese können jedoch normale virtuelle Methoden sein, die Sie in den abgeleiteten Klassen überschreiben können (ohne das Property neu deklarieren zu müssen).

Property-Kategorien *R197*

Borland hat sich in Delphi 6 von der erst in Delphi 5 eingeführten komplizierten Art, neue Kategorien zu registrieren, schnell wieder verabschiedet. Wo man bisher extra eine neue Klasse von *TPropertyCategory* ableiten musste, nur um einen neuen Kategorie-Namen festzulegen, wird dieser Name nun in einem einfachen Prozeduraufruf direkt angegeben.

Sofern Sie sich nicht darum kümmern, ordnet Delphi die von Ihnen definierten Properties automatisch in die im Objektinspektor genannte Kategorie »Verschiedene«. Properties, die eine neue Komponente von einer VCL-Komponente erbt, bleiben in der Kategorie, in der sie von der VCL eingeordnet wurden.

Davon abweichend können Sie in der seit Kapitel 6.2.2 bekannten *Register*-Prozedur weitere Kategorie-Zuweisungen vornehmen. Sie können sowohl vordefinierte als auch selbst definierte Kategorien verwenden. Wie beides geht, zeigt der folgende Code-Auszug für die *TColorPalette*-Komponente, die in Kapitel 6.6 entwickelt wird.

Er dient dazu, die Properties der Farbpalette, welche die gewählten Farben für Stift, Pinsel und Schriftart angeben, in der Kategorie »Gewählte Palettenfarben« und sechs weitere Properties unter der Kategorie »TColorPalette-spezifisch« zusammenzufassen. Im Aufruf von *RegisterPropertiesInCategory* werden der Kategorie-Name, die Komponentenklasse und ein Array mit den Namen aller in die angegebene Kategorie fallenden Properties übergeben:

```
procedure Register;
begin
  RegisterPropertiesInCategory('Gewählte Farbpalettenfarben',
    TColorPalette, ['PenColor', 'BrushColor', 'FontColor']);
  RegisterPropertiesInCategory('TColorPalette-spezifisch',
    TColorPalette, ['Columns', 'Lines', 'SwapAlign', 'NoSelection',
    'RGBSteps', 'ColorDummy']);
```

Dies ist jedoch nur das Grundprinzip, von dem Sie abweichen können, indem Sie:

- statt einer selbst definierten Kategorie eine vordefinierte verwenden (in der Unit *DesignIntf* finden Sie unter der Überschrift *resourcestring* Namenskonstanten für mehr als zehn vordefinierte Kategorien, z.B. *sActionCategoryName* und *sLayoutCategoryName*);
- weitere Registrierungsmethoden verwenden, mit denen Sie Properties auch einzeln oder mit Platzhalterzeichen (*color** für alle Properties, deren Name mit »Color« beginnt) oder durch Angabe ihrer Typen (statt ihrer Namen) oder durch Kombinationen aus Komponentenklasse, Property-Typ und Property-Name registrieren können. Hierzu sei auf die Online-Hilfe zu den Funktionen *RegisterPropertyInCategory* und *RegisterPropertiesInCategory* verwiesen.

Wichtig ist vor allem noch, dass die Registrierung der Property-Kategorien eine reine Entwurfszeitfunktion ist und somit in einer Unit definiert werden sollte, die nur zur Entwurfszeit geladen wird. Die Kategorien der Farbpalette beispielsweise werden zusammen mit den Komponenten- und Property-Editoren (siehe Kapitel 6.6.5) in der Unit *ColorPalDesignTime* definiert.

6.3.2 Events

Die Verknüpfung von Komponentenereignissen mit Methoden des Formulars kommt ohne dynamische oder virtuelle Methoden aus und verwendet statt dessen Methodenzeiger und Properties, die zusammen eine Automatisierung von Routineaufgaben im Objektinspektor erlauben.

Definition des Events

Die Begriffe Event / Ereignis und Message / Nachricht / Botschaft sind im Zusammenhang mit ereignisorientierten Programmen meistens austauschbar und nicht auf einen speziellen Weg der Nachrichtenübermittlung festgelegt. In Delphi hat der Begriff »Event« jedoch eine feste Bedeutung. So meint Event immer ein Ereignis, das von einer Komponente ausgelöst wird und bei dem die Komponente eine Methode aufruft, die in einem Methodenzeiger gespeichert ist. Meistens weist der Methodenzeiger auf die Methode eines Formulars. Sie können jedoch an der Automatik des Objektinspektors vorbei zur Laufzeit auch Methoden anderer Klassen oder anderer Formulare mit den Event-Properties verknüpfen (wie beispielsweise bei der Bearbeitung eines Dokumentfenster-Menüpunkts durch das MDI-Hauptfenster, siehe Kapitel 5.7.4).

Deklaration eines Events

Die Deklaration eines Events besteht aus mehreren Teilen an verschiedenen Stellen der Komponenten-Unit:

- einem Methodenzeiger-Typ, der die Parameter der Ereignisbearbeitungsmethode festlegt (dieser Typ befindet sich außerhalb der Komponentendeklaration im normalen *type*-Block),
- einem Methodenzeiger innerhalb der Komponente, der den eben beschriebenen Typ hat
- und dem Property, das dem Objektinspektor Zugriff auf diesen Methodenzeiger verschafft

Für den Methodentyp machen wir einen kleinen Rückblick auf die Ereignisbearbeitungs-Methoden, die Sie aus dem Objektinspektor automatisch erstellen können, z.B.:

```
procedure TForm1.DateiOeffnenClick(Sender: TObject);
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

Die dazugehörigen Methodenzeiger-Typen sind in den Units der VCL wie folgt deklariert:

```
{ Unit Classes }
  TNotifyEvent = procedure(Sender: TObject) of object;
{ Unit Controls: }
  TMouseEvent = procedure(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer) of object;
```

Die Variablen dieser Typen befinden sich in den Komponenten, die Events dieser Art implementieren, so z.B. in den Menüpunktkomponenten für die *OnClick*-Ereignisse:

```
type
  TMenuItem = class(TComponent)
    ...
    FOnClick: TNotifyEvent;
    ...
```

Das zugehörige Property ist nur dazu da, die Variable *FOnClick* im Objektinspektor zugänglich zu machen. Es benötigt daher keine *read*- und *write*-Methoden, sondern leitet Anfragen direkt an *FOnClick* weiter:

```
property OnClick: TNotifyEvent read FOnClick write FOnClick;
```

Die anderen Regeln, die für Properties gelten, treffen auch bei Event-Properties zu. So können die Event-Properties als *protected* deklariert sein und erst später in abgeleiteten Klassen veröffentlicht werden. Weniger sinnvoll erscheinen allerdings nur lesbare oder nur beschreibbare Events.

Ein Beispiel-Event

Zweck eines neuen Events könnte es beispielsweise sein, das Formular über die Wahl einer Farbe aus einer Farbpalette zu benachrichtigen. Die in Kapitel 6.6 besprochene

Komponente *TColorPalette* definiert für die Farben von Stift, Pinsel und Schrift jeweils ein Event des Typs *TColorChangeEvent*. Alle dafür notwendigen Deklarationen sind im folgenden Quelltextausschnitt zusammengefasst:

```
type
  TColorChangeEvent =
    procedure(Sender: TObject; Color: TColor) of object;
  TColorPalette = class(TCustomControl)
  private
    FOnPenColorChange,
    FOnBrushColorChange,
    FOnFontColorChange: TColorChangeEvent;
  published
    property OnPenColorChange: TColorChangeEvent
       read FOnPenColorChange write FOnPenColorChange;
    ...
```

Dabei gibt der erste Parameter wie üblich die *TColorPalette*-Komponente an, bei der das Ereignis entstanden ist, im zweiten Parameter befindet sich die neu gewählte Farbe. Der Parameter *Sender* könnte natürlich auch wie folgt deklariert sein, da das Event hier ausschließlich von der Klasse *TColorPalette* generiert wird:

```
  TColorPalette = class; { Vorwärtsdeklaration }
  TColorChangeEvent =
    procedure(Sender: TColorPalette; Color: TColor) of object;
```

Konventionen für Events

Wenn Sie eigene Events für Komponenten schreiben, sollten Sie sich in der Namensgebung an die Konventionen der VCL halten. Demnach beginnen alle Events mit dem Präfix *On*, die zugehörigen Methodenzeigervariablen sind als *private* deklariert und stellen dem Namen des Properties ein *F* voran, z.B. *FOnClick*.

Für den Komponentenbenutzer ist noch ein zweiter Bezeichner sichtbar: der Name der Methoden, die mit dem Event verknüpft werden. Dieser Name wird jedoch von Delphi (bzw. dem Property-Editor für die Event-Properties) automatisch erzeugt und setzt sich aus dem Namen der Komponente, deren Methodenzeiger mit dieser Methode verknüpft wird, und aus dem Namen des Events, allerdings ohne das *On*, zusammen (z.B. *Button1Click*). Für Events des Formulars setzt Delphi statt des Formularnamens einfach *Form* ein, z.B. *FormMouseDown*.

Eine weitere Konvention für Events ist, dass sie immer als Prozedur deklariert sind. Wenn die verknüpfte Methode einen Wert zurückgeben soll, so sollte das Event dafür einen Variablenparameter bereitstellen. Ein solcher Variablenparameter kann vor dem Aufruf der Bearbeitungsmethode bereits auf einen sinnvollen Wert voreingestellt werden – anders als das Ergebnis einer Funktion. Wenn die Bearbeitungsmethode vergisst,

den Ergebniswert zu setzen, bleibt der Variablenparameter bei seinem voreingestellten Wert, während ein nicht zugewiesenes Funktionsergebnis ein zufälliger Wert ist.

6.3.3 Events auslösen

Um das Event auszulösen, müssen Sie die mit dem Methodenzeiger verknüpfte Ereignisbearbeitungs-Methode aufrufen (die aus der Sicht der Komponente auch als *Event-Handler* bezeichnet wird). Dieser Aufruf unterscheidet sich ein wenig von einem normalen Prozeduraufruf. Da wahrscheinlich die meisten Events im Objektinspektor gar nicht mit Methoden verknüpft werden, bleiben die meisten Methodenzeiger bei ihrem voreingestellten Wert *nil*. Gegenüber dem normalen Prozeduraufruf kommt beim Aufruf von Events also immer eine Abfrage hinzu, ob das Event überhaupt verknüpft wurde.

Angenommen, in einer Komponente der Klasse *TColorPalette* aus dem obigen Beispiel ändert sich die Stiftfarbe. Die Komponente ruft dann das zugehörige Event wie folgt auf und gibt sich selbst im *Sender*-Parameter an:

```
if Assigned(FOnPenColorChange)
  then FOnPenColorChange(self, NewPenColor);
```

Leere Event-Handler

Die Tatsache, dass die mit dem Ereignis verknüpfte Methode leer sein kann, führt nicht nur dazu, dass Sie die Methode als Prozedur statt als Funktion deklarieren sollten, sondern auch dazu, dass Sie die obige *if*-Anweisung nicht wie folgt erweitern sollten:

```
if Assigned(FOnPenColorChange)
  then FOnPenColorChange(self, NewPenColor)
else Ersatzaktion;
```

Dies würde dazu führen, dass das Vorhandensein einer völlig sinnlosen Bearbeitungsmethode *FOnPenColorChange* den Aufruf der *Ersatzaktion* verhindert. Eine sicherere Lösung wäre, die Ausführung der *Ersatzaktion* von einer um einen Variablen-Parameter erweiterten Bearbeitungsmethode abhängig zu machen; der Benutzer der Komponente müsste diesen Parameter dann bewusst auf *False* setzen, um die *Ersatzaktion* abzuschalten.

```
ErsatzErforderlich := True; { Voreinstellung }
if Assigned(FOnPenColorChange)
  then FOnPenColorChange(self, NewPenColor, ErsatzErforderlich);
if ErsatzErforderlich
  then Ersatzaktion;
```

6.4 Komponenten intern

Bei der Programmierung einer Komponente gibt es einige Unterschiede zum Programmieren eines Formulars, zu denen die folgenden gehören:

- Um Nachrichten selbst zu bearbeiten, kann sich eine Komponente nicht der Events bedienen, da diese für den Benutzer reserviert sind, sondern sie muss virtuelle bzw. dynamische Methoden überschreiben oder *message*-Methoden definieren.
- Wenn eine Komponente andere Komponenten enthalten soll, müssen Sie diese selbst dynamisch erzeugen, da Komponenten nicht wie Formulare entworfen werden können.
- Auf Komponenten können weitere Aufgaben zukommen, wie das Teilnehmen am Speicherungsprozess des Formulars, die Bereitstellung von speziellen Property-Editoren usw.

6.4.1 Ereignisse in den Komponenten

Standen im letzten Kapitel die Ereignisse im Mittelpunkt, die die Komponente auslöst (die Events), geht es hier um die Ereignisse, die sie selbst bearbeitet.

Hinweis: *Event* meint hier ausschließlich Methodenzeiger-Properties, die zur Verknüpfung eines Komponentenereignisses mit der Methode eines beliebigen Objekts (meistens eines Formulars) dienen. Der Begriff *Ereignis* steht hier allgemein für Ereignisse aller Art.

Events und Ereignisse

So wie ein Formular auf äußere Ereignisse reagieren muss, können auch Komponenten Ereignisse empfangen. Einige Beispiele sind:

- Die *TScrollBox*-Klasse bearbeitet die Ereignisse, die von den beiden Bildlaufleisten erzeugt werden, und verschiebt als Reaktion darauf die Kindfenster der Scrollbox.
- Viele Komponenten, die nicht von Windows gezeichnet werden, reagieren auf einen Neuzeichnen-Auftrag, wie Formulare das beispielsweise in einer *OnPaint*-Bearbeitung tun können.
- Die Timer-Komponente bearbeitet die vom Betriebssystem gesendeten Timer-Botschaften.

Manche Komponenten geben diese Ereignisse an den Komponentenbenutzer weiter. So reagiert die Timer-Komponente auf die *WM_Timer*-Nachricht, indem sie die Methode aufruft, die der Komponentenbenutzer mit dem Ereignis *OnTimer* verknüpft

hat. *TEdit* verarbeitet die empfangenen Tastendrücke und gibt sie an die mit *OnKeyDown*, *OnKeyPress* und *OnKeyUp* verbundenen Methoden weiter.

Andere Nachrichten werden nicht weitergegeben. So finden Sie im Objektinspektor für viele Komponenten kein *OnPaint*-Event, obwohl diese von Windows eine *WM_Paint*-Botschaft erhalten. Normalerweise ist der einsehbare Grund dafür, dass sich die Komponente bereits selbst auf den Bildschirm malt und das dabei entstandene Kunstwerk nicht von einer Formularmethode verfälscht sehen will.

Die zwei Erscheinungsformen der Ereignisse

Die Ereignisbearbeitung einer Komponente unterscheidet sich grundlegend von der Reaktion eines Formulars auf Events. Während Sie bei einem Formular den Objektinspektor zur Verfügung haben und (ohne es wissen zu müssen) mit Methodenzeigern arbeiten, fangen Sie Ereignisse in Komponenten meistens auf eine der beiden folgenden Arten ab:

- Unabhängig davon, ob die Basisklasse eine virtuelle Methode für ein Ereignis besitzt, haben Sie immer die Möglichkeit, mit der *message*-Direktive eine neue Ereignisbearbeitungsmethode zu definieren. Sie brauchen nicht zu wissen, ob die Basisklasse das Ereignis bereits bearbeitet, sondern können in jedem Fall die Anweisung *inherited* aufrufen, die eine solche geerbte Bearbeitungsmethode aufruft, falls sie vorhanden ist, und die in allen anderen Fällen zu *DefaultHandler* verzweigt (siehe Abbildung 3.2 auf Seite 324).
- Oft bearbeitet die Basisklasse die Windows-Nachricht bereits und ruft in dieser Bearbeitung eine virtuelle Methode auf, die speziell dafür gedacht ist, dass abgeleitete Klassen sich durch Überschreiben derselben in die Bearbeitung des Ereignisses einklinken. Sie brauchen dann nur diese Methode zu überschreiben, um das Ereignis zu bearbeiten.

In Abbildung 3.2 sind diese und andere Stellen des Nachrichtenflusses gekennzeichnet, an denen Sie eine Windows-Nachricht abfangen können. Für die Komponentenprogrammierung genügen normalerweise jedoch die beiden eben genannten Möglichkeiten.

Virtuelle Methoden

Die virtuellen Methoden der Klassen *TControl*, *TWinControl* und *TCustomControl*, die in Verbindung mit Ereignissen auftreten, sind in der folgenden Tabelle zusammengefasst:

Ereignis-Art	virtuelle Methoden	ab Klasse
Klicks	*Click*, *DblClick*	*TControl*
Drag&Drop	*DragDrop*, *DragOver*, *DoStartDrag*, *DoEndDrag*	*TControl*
Maus	*MouseDown*, *MouseMove*, *MouseUp*	*TControl*
Tastatur	*KeyDown*, *KeyPress*, *KeyUp*	*TWinControl*
Tastaturfokus	*DoEnter*, *DoExit*	*TWinControl*
Docking	DoDock, DoStartDock, DoEndDock	TControl
Ereignisse für DockSites	GetSiteInfo, DockOver, DockDrop, DoUndock	TWinControl
Zeichnen	*Paint*	*TCustomControl*

Beispiele für die Verwendung von *DoExit* finden Sie in Kapitel 6.5.3, Beispiele für *Paint* und *MouseDown* in Kapitel 6.6.3.

Während die obige Tabelle nur die allgemeinen, schon in *T(Win)Control* definierten virtuellen Methoden nennt, finden Sie in den konkreten Komponentenklassen oft eine Reihe weiterer virtueller Methoden, mit denen Sie Ereignisse behandeln können, die normalerweise als Events im Formular bearbeitet werden. So sind das beispielsweise für die *OnEdit*-, *OnChange*- und *OnCanChange*-Ereignisse von *TTreeView* die virtuellen Methoden *Edit*, *Change* und *CanChange*.

Es empfiehlt sich also in jedem Fall vor dem Ableiten einer neuen Klasse von einer bestehenden Klasse, sich in der Online-Hilfe, im VCL-Quelltext oder im Browser der IDE einen Überblick über die Erweiterungsmöglichkeiten zu verschaffen.

Message-Methoden *R198*

Die Komponente *TScrollBoxEx* wird zwar erst in Kapitel 6.5.2 besprochen, liefert hier aber schon vorab zwei Beispiele für *message*-Methoden. Und zwar geht es darum, die Windows-Botschaften *WM_HScroll* und *WM_VScroll* abzufangen, um über jede Bewegung der beiden Scrollbars informiert zu werden. Die beiden Methoden sind wie folgt deklariert:

```
procedure WMHScroll(var Param: TWMScroll); message WM_HScroll;
procedure WMVScroll(var Param: TWMScroll); message WM_VScroll;
```

Zwei Teile dieser Deklaration sind charakteristisch und für jede *message*-Methode erforderlich: ein Variablenparameter, der auf die unten beschriebene Struktur weist, und die *message*-Direktive, die nach dem üblichen Prozedurkopf folgt. Nach *message* geben Sie den *Nachrichtencode* an, eine Konstante, die die zu bearbeitende Windows-Botschaft identifiziert. Die normalen Fensterbotschaften beginnen beispielsweise mit »WM_« und sind in der Online-Hilfe zum Windows-API dokumentiert.

Was Ihnen überlassen bleibt, sind der Name der Methode und des Parameters. Natürlich ist es am einfachsten, den Namen der Methode aus dem Bezeichner des Nachrichtencodes zu bilden und dabei lediglich das Zeichen »_« zu entfernen.

Der Parameter einer Message-Methode

Eine *message*-Methode wird immer mit einem Zeiger aufgerufen, der auf eine *TMessage*-Struktur zeigt. Wenn Sie den Parameter wie in den Beispielen als Variablenparameter deklarieren, brauchen Sie sich nicht um die Adresse zu kümmern, sondern können direkt auf die Elemente des *TMessage*-Records zugreifen.

Der Aufbau dieser Struktur hängt von der bearbeiteten Botschaft ab. Die VCL-Unit *Messages* enthält für alle üblichen Windows-Botschaften spezielle Recordtypen, die diesen Aufbau in einer verständlichen Form beschreiben. Für die Botschaften *WM_HScroll* und *WM_VScroll* ist das die Struktur *TWMScroll*:

```
TWMScroll = record
  Msg: Cardinal;
  ScrollCode: Smallint;
  Pos: Smallint;
  ScrollBar: HWND;
  Result: LongInt;
end;
```

Im ersten Feld befindet sich der Nachrichtencode, der mit dem hinter *message* angegebenen Nachrichtencode übereinstimmt (aufgrund dieser Übereinstimmung ist das Feld *Msg* nur außerhalb einer *message*-Methode, beispielsweise in einer *WndProc*-Methode, von Interesse). Auch das letzte Element ist für jede Nachricht gleich: Es nimmt das Ergebnis der Ereignisbearbeitung auf, so dass Sie die *message*-Methoden nicht als Funktion deklarieren müssen.

Sollte es zu einer speziellen Nachricht einmal keine passende Struktur in der Unit *Messages* geben, können Sie entweder einen eigenen Recordtyp definieren oder den Typ *TMessage* verwenden: Der Parameter einer *message*-Funktion entspricht *immer* dem Aufbau von *TMessage*, die speziellen Recordtypen wie *TWMScroll* interpretieren lediglich die beiden Elemente *WParam* und *LParam* auf ihre eigene Weise (siehe auch Kapitel 3.1.4).

Die geerbte Ereignisverarbeitung

Sowohl in einer *message*- als auch in einer virtuellen Methode sollten Sie normalerweise immer die von der Basisklasse geerbte Ereignisbearbeitung aufrufen. In einer *message*-Methode besteht dieser Aufruf in einer besonders kurzen Zeile:

```
inherited;
```

Es macht also nichts aus, wenn die in der Basisklasse definierte *message*-Methode mit *private* geschützt ist oder Sie deren Namen nicht kennen.

Das Überschreiben der virtuellen Methoden läuft dagegen standardmäßig ab, Sie müssen die geerbte Methode also bei ihrem Namen nennen:

```
inherited MouseDown(Button, Shift, X, Y);
```

6.4.2 Komponenten in der Formulardatei

Dieses Kapitel gibt einen Überblick über die Speicher- und Ladeprozesse der Komponenten und wie Sie in diese eingreifen können. Es liefert damit unter anderem den Hintergrund zu Kapitel 6.4.3.

Speichern von Komponenten

Das Datei-Abbild einer Komponente besteht im Wesentlichen aus den Properties und aus den Unterkomponenten der Komponente, wobei die Klasse *TWriter* den größten Teil der Speicherung erledigt. Der gesamte Prozess des Speicherns besteht, wie der VCL-Quellcode zu erkennen gibt, aus einem komplizierten Wechselspiel zwischen *TComponent* und *TWriter*. Für die Entwicklung eigener Komponenten brauchen Sie jedoch die Details davon nicht zu kennen, daher verzichtet die folgende Zusammenfassung darauf, zu erläutern, welche Klasse der VCL die jeweiligen Schritte durchführt.

Während Sie beim Laden einer Komponente drei Gelegenheiten haben, Code Ihrer Komponentenklasse auszuführen, kommt beim Speichern der Komponente normalerweise nur die virtuelle Methode *DefineProperties* zum Überschreiben in Frage. In der folgenden groben Skizzierung der Komponentenspeicherung wird *DefineProperties* im dritten Schritt aufgerufen:

- Zuerst schreibt die VCL die Klasseninformationen in den Stream, damit auch polymorphe Komponenten unverändert wiederherstellbar sind.
- Dann speichert sie alle Properties, die gewisse Regeln erfüllen, zu denen wir in Kapitel 6.4.3 kommen.
- Sie ermöglicht daraufhin den abgeleiteten Komponentenklassen über die virtuelle Methode *DefineProperties* (definiert in *TPersistent*), weitere Daten zu speichern, die außerhalb der automatisch gespeicherten Properties liegen (siehe Kapitel 6.6.6 für ein Beispiel).

- Den Properties lässt sie eine Liste der in der Komponente enthaltenen Komponenten folgen (Property *Components*), von denen jedoch die Steuerelemente zunächst ausgeschlossen bleiben, denn diese werden in einer eigenen Liste gespeichert:
  - *TWinControl*-Komponenten speichern schließlich auch die in ihnen enthaltenen Steuerelemente (Property *Controls*).

Da die Klasse *TWinControl* bereits in der Lage ist, ein komplettes Fenster mit all seinen Unterfenstern, Komponenten und Properties zu speichern, muss die Klasse *TForm* diesem Vorgang nichts mehr hinzufügen.

Laden von Komponenten

Das Laden der Komponente ist nicht einfach nur eine Umkehrung des Schreibens, sondern enthält einige zusätzliche Stationen, die Sie zur korrekten Initialisierung Ihrer Komponente verwenden können (Schritte 2, 4 und 6):

- Zum Beginn des Ladens ist noch gar keine Komponente vorhanden. Die VCL liest zuerst die Klasseninformation, die im ersten Schritt des Speicherns geschrieben wurde. Anhand dieser Daten sucht sie dann mit *FindClass* (siehe Kapitel 4.3) die zugehörige Klassenreferenz, mit deren Konstruktor sie eine neue Objektinstanz erstellt.
- Wenn Sie den virtuellen Konstruktor der Basisklasse überschreiben, erhalten Sie an dieser Stelle die erste Gelegenheit, in den Prozess einzugreifen. Sie können innerhalb des Konstruktors über das Property *ComponentState* sogar feststellen, ob die Komponente gerade aus einer Formulardatei gelesen wird oder ob es sich um einen normalen Konstruktoraufruf handelt.
- Erst nach dem Konstruktoraufruf kann die VCL die automatisch gespeicherten Properties laden.
- Es folgt Ihre zweite Möglichkeit, am Ladeprozess teilzunehmen: Nachdem diese Properties geladen sind, erhalten Sie in der Methode *DefineProperties* die Möglichkeit, weitere Daten zu lesen.
- Der VCL-Code setzt das Laden dann gemäß der Speicherreihenfolge fort und liest die Komponentenliste und eventuell die Liste der untergeordneten Fenster.
- Zum Abschluss gibt Ihnen die VCL noch einmal die Gelegenheit, eigene Initialisierungen vorzunehmen: Wenn alle Daten geladen sind, die Komponente also soweit wiederhergestellt ist, wie es der VCL möglich war, ruft sie die virtuelle Methode *Loaded* auf. Überschreiben Sie diese, um abschließende Initialisierungen vorzunehmen (ein Beispiel finden Sie in Kapitel 6.7.3).

6.4.3 Steuerung der Property-Speicherung

Bereits in der Deklaration eines Properties haben Sie Einfluss auf die Dateioperationen der VCL: Wenn Sie in ihr nur die nötigsten Angaben machen, speichert die VCL genau die Properties, die auch als *published* deklariert sind (wobei Array-Properties nicht *published* sein dürfen). Um von dieser Standardverhaltensweise abzuweichen, können Sie die Direktiven *stored* und *default* einsetzen sowie weitere Daten zur Laufzeit als Property »verkleiden«, um sie ebenfalls speichern zu können.

Default

Mit *default* geben Sie der VCL weitere Informationen, anhand derer diese selbst entscheiden kann, ob das Property gespeichert wird. Diese Direktive ist nur bei Properties erlaubt, die einen ordinalen oder einen Mengentyp haben. (Bei Array-Properties hat *default* eine andere Bedeutung, siehe Kapitel 2.2.5).

Hinter dem Wort *default* geben Sie einen konstanten Wert an, der dadurch als voreingestellter Wert definiert wird. Sie teilen der VCL dadurch mit, dass sie das Property nicht speichern muss, wenn es diesen Wert aufweist.

Zu einer solchen *default*-Angabe gehört immer eine passende Zuweisung im Konstruktor der Komponente. Mit der folgenden Deklaration verpflichtet sich *TForm* beispielsweise, das Property *Position* im Konstruktor mit dem Wert *poDesigned* vorzubelegen:

```
property Position: TPosition read FPosition
  write SetPosition default poDesigned;
```

Da beim Speichern eines Property auch viele Verwaltungsinformationen (darunter auch der Name des Properties) in die Datei geschrieben werden müssen, ein nicht gespeichertes Property aber überhaupt keine Spur in der Datei hinterlässt, kann man sich ausrechnen, dass durch die Vergabe von Default-Werten ein deutlicher Geschwindigkeitsgewinn erzielt wird: Oft werden nur wenige Voreinstellungen der Properties geändert, daher müssen auch nur wenige Properties geschrieben werden. Für die anderen Properties wird noch nicht einmal ein Flag in die Datei geschrieben, welches angibt, dass das Property nicht gespeichert wurde.

Object Pascal macht sich außerdem die Mühe, die Direktive *nodefault* zu definieren. Sie können diese verwenden, um das Fehlen eines *default*-Werts zu verdeutlichen oder beim Überschreiben eines Properties den geerbten *default*-Wert auszuschalten.

Stored

Mit der Direktive *stored* legen Sie fest, ob das Property grundsätzlich gespeichert werden soll (falls sein Wert nicht mit dem *Default*-Wert übereinstimmt):

```
public
  property Zahl: Integer; read Get write Put stored True;
```

Mit »stored True« können Sie erreichen, dass ein nicht veröffentlichtes Property gespeichert wird, während *stored False* sinnvoll ist, um veröffentlichte Properties von der Speicherung auszuschließen (dies könnte bei Properties der Fall sein, die zur Entwurfszeit nützliche Funktionen beinhalten, zur Laufzeit aber nicht gebraucht werden).

Die direkte Angabe einer booleschen Konstante ist jedoch nur eine Möglichkeit für eine *stored*-Direktive. Alternativ dazu können Sie auch ein Element der Klasse, das zu einem booleschen Ergebnis führt, angeben. Dieses Element kann entweder eine *Boolean*-Variable bzw. ein Property sein oder eine Funktion, die keine Parameter, aber ein *Boolean*-Ergebnis hat. In einer solchen Funktion können Sie noch ausgefeiltere Bedingungen zur Speicherung eines Werts stellen, als dies durch die Angabe eines *Default*-Wertes möglich ist.

Nicht zu vergessen ist, dass die VCL nach der *stored*-Angabe auch noch die *default*-Angabe auswertet. Wenn die Überprüfung des *stored*-Teils also grünes Licht für die Speicherung gibt, kann eine Übereinstimmung des Properties mit dem voreingestellten Wert die Speicherung immer noch verhindern.

Überschreiben von Stored und Default

Wenn Sie ein Property überschreiben, können Sie nicht nur die Lese- und Schreibmethoden, sondern auch die *stored*- und *default*-Werte ändern. Daher könnte es sogar sinnvoll sein, dass Sie ein Property mit »stored False«, aber trotzdem mit *Default*-Wert deklarieren. Wenn eine abgeleitete Klasse sich entschließt, das Property doch zu speichern, braucht diese Klasse nur die *Stored*-Direktive zu überschreiben; durch den geerbten *Default*-Wert ist trotzdem eine effektive Speicherung gewährleistet.

Selbst speichern

Wie der letzte Abschnitt erläutert hat, findet bereits eine differenzierte Abwägung statt, *ob* ein Property gespeichert werden soll – was einen großen Effektivitätsgewinn zur Folge hat. Da die VCL weder Array-Properties speichern kann, noch irgendwelche Daten, die nicht als Property vorliegen, müssen Sie ab und zu selbst für die Speicherung sorgen.

Bei Betrachtung der *read*- und *write*-Direktiven eines Properties läge es nahe, der Deklaration von Array-Properties einfach eine *fileread*- und eine *filewrite*-Direktive hinzuzufügen und so Methoden zum Speichern und Laden dieser Properties festzulegen. Borland führt diese Direktiven jedoch nicht in Object Pascal ein, sondern verlässt sich auf eine noch flexiblere Methode, die von der VCL bereitgestellt wird (und die auch die Daten, die nicht als Property vorliegen, berücksichtigt):

Sie können durch Überschreiben der *TPersistent*-Methode *DefineProperties* beliebige Datenblöcke definieren, die – quasi als Property getarnt – in eine Datei geschrieben bzw. aus dieser gelesen werden können. *DefineProperties* bezeichnet diese Datenblöcke nicht zur Verwirrung des Programmierers als Properties, sondern weil die Handhabung dieser selbst definierten Datenblöcke sehr ähnlich zur Handhabung von Properties ist. Allerdings gelten die mit *DefineProperties* definierten Properties ausschließlich beim Lesen und Schreiben der Datei als Properties.

Kapitel 6.6.6 erklärt anhand einer Beispielkomponente, wie eine solche *DefineProperties*-Methode im Detail funktioniert.

6.4.4 Die Schnittstelle zur Delphi-IDE

Um eine oder mehrere Komponenten in die Komponentenpalette einzubinden, genügt bereits ein Aufruf von *RegisterComponents*, den Sie in einer Prozedur namens *Register* vornehmen. Nach der Registrierung steht Ihre Komponente auf der gewünschten Seite der Palette und die veröffentlichten editierbaren Properties sind im Objektinspektor zugänglich. Dem Beispiel für einen *RegisterComponents*-Aufruf aus Kapitel 6.2.2 ist hier nichts mehr hinzuzufügen. Seit Delphi 4 können Sie zusätzlich auch Standardaktionen für Ihre Komponenten registrieren, die bei der Verwendung Ihrer Komponenten leicht in die Menüs einer Anwendung eingebunden werden können. Ein Beispiel hierzu finden Sie in Kapitel 6.6.7. In Zusammenhang mit der Benutzerschnittstelle erwähnenswert sind schließlich noch die schon in Kapitel 6.3.1 besprochenen und besonders leicht zu definierenden Property-Kategorien.

Bitmaps

Die einfachste Aktion, die Sie zusätzlich zur Registrierung durchführen können, besteht im Zeichnen einer Bitmap für die Komponentenpalette. Legen Sie dazu mit dem Bildeditor (TOOLS | BILDEDITOR) eine DCR-Datei an und speichern Sie in dieser für jede Ihrer Komponenten eine 24x24 Pixel große Bitmap.

Um die Verbindung zu Ihrer Komponente herzustellen, geben Sie dieser Bitmap den Namen der Komponente, umgeformt in Großbuchstaben. Um die Verbindung zu der Komponenten-Unit herzustellen, geben Sie der DCR-Datei denselben Namen wie der Unit und speichern sie im selben Verzeichnis. Schließlich nehmen Sie die DCR-Datei noch in die *Contains*-Liste des Package-Editors auf, indem Sie dort HINZUFÜGEN drücken und die Datei auswählen. Falls Sie die Komponente über KOMPONENTE | KOMPONENTE INSTALLIEREN... installieren möchten, binden Sie die DCR-Datei mit einer *$R*-Anweisung direkt in diese Unit ein.

Editoren

Zur Implementierung von komplexeren Komponenten genügen oft weder die einfachen Datentypen, die Sie direkt im Objektinspektor editieren können, noch die komplexeren Typen, für die die VCL bereits Editoren in eigenen Fenstern zur Verfügung stellt (z.B. *TStringList*, *TBitmap*, *TMenuItem*). Daher gibt Ihnen Delphi die Möglichkeit, eigene Editoren sowohl für einzelne Properties als auch für ganze Komponenten zu registrieren.

Property-Editoren

Property-Editoren dienen dazu, für einen Property-Typ, der bisher nicht editierbar war, weil er nicht in einer der üblichen Formen dargestellt werden konnte, einen neuen Editor zur Verfügung zu stellen. Dieser Editor kann in einem eigenen Formular definiert sein, das als Dialog aufgerufen wird, oder lediglich die Darstellung des Textes im Objektinspektor beeinflussen. Sie können einen Editor auch für Properties schreiben, die schon über einen Editor verfügen, und damit den bisherigen Editor ersetzen. Property-Editoren müssen nicht für alle Properties eines Typs gelten, sondern können sich auch auf ein bestimmtes Property beschränken.

Um einen eigenen Property-Editor zu schreiben, leiten Sie eine neue Klasse von *TPropertyEditor* ab und installieren diesen mit der Prozedur *RegisterPropertyEditor*, die der Prozedur *RegisterComponents* ähnelt und wie diese in der *Register*-Prozedur der Komponenten-Unit aufgerufen wird. Ein konkretes Beispiel dazu gibt Kapitel 6.6.5. Nach erfolgreicher Registrierung verwendet Delphi diesen Editor für alle Properties, die den angegebenen Typ haben, oder für das speziell angegebene Property.

Komponenten-Editoren

Komponenten-Editoren werden zur Entwurfszeit entweder durch einen Doppelklick auf die Komponente oder durch die Auswahl eines eigenen Menüpunkts im Popup-Menü der Komponente aufgerufen. Ein Komponenten-Editor hat, ähnlich wie ein Property-Editor, nichts mit einem herkömmlichen Editor zu tun.

Nach der Definition der abstrakten Klasse *TComponentEditor* ist ein Komponenten-Editor eine Klasse, deren Hauptaufgabe darin besteht, verschiedene zusätzliche Befehle (so genannte Verben) für eine bestimmte Komponentenklasse zur Verfügung zu stellen und auszuführen. Beispielsweise stellt der Komponenten-Editor für *TTreeView*-Komponenten einen Eintrags- und einen Spalteneditor zur Verfügung.

Sie erstellen einen neuen Komponenten-Editor, indem Sie eine neue Klasse von *TComponentEditor* ableiten und die Routine *RegisterComponentEditor* aufrufen. Auch hierfür finden Sie in Kapitel 6.6.5 ein Beispiel.

Komponenten zur Entwurfszeit

Eine der großen internen Neuerungen von Delphi gegenüber früheren Pascal-Entwicklungsumgebungen ist die Tatsache, dass Teile des von Ihnen entwickelten Codes nahtlos in die IDE eingebunden werden können, indem Sie Ihren Code an die Komponentenschnittstelle anpassen. Tatsächlich könnte sogar der gesamte Code Ihrer Unit schon dann aktiv werden, wenn der Benutzer Ihrer Komponenten sich noch im Entwurfsmodus befindet. Die Komponenten-Units sind auf eine viel engere Art in die Delphi-IDE eingebunden, als dies mit normalen Windows-DLLs möglich wäre.

Der einzige Grund, warum die Komponenten zur Entwurfszeit auf den ersten Blick etwas bewusstlos erscheinen, ist, dass der VCL-Code die allermeisten Ereignisse der IDE überlässt. Mauseingaben werden zum Verschieben der Komponente oder zum Aufruf des Popup-Menüs verwendet und Tastatureingaben aktivieren den Objektinspektor. Beide Ereignisse gelangen nicht mehr zu den von Ihnen geschriebenen Methoden.

Trotzdem wird auch zur Entwurfszeit ständig ein Teil des von Ihnen geschriebenen Codes aufgerufen: Wenn Sie ein Property im Objektinspektor verändern, ruft Delphi die zugehörige Schreibmethode auf. Wenn Sie beispielsweise bei der Farbpalette aus Kapitel 6.6 das Property *Columns* ändern, ruft die Palette *Invalidate* auf, so dass die Entwurfszeit-Darstellung der Palette sofort angepasst wird.

Hinweis: Da sich die IDE die für sie reservierten Eingabenachrichten nicht von selbst holt, sondern von der zur Entwurfszeit aktiven VCL herausfiltern lässt, können Sie sogar die für die IDE reservierten Nachrichten zur Entwurfszeit in Ihren Komponenten bearbeiten. Sie müssen dazu lediglich die *WndProc*-Methode überschreiben, so dass Sie die Nachrichten bearbeiten können, bevor die VCL sie der IDE zuspielen kann.

Lagebestimmung

Damit die VCL die reservierten Nachrichten nicht auch zur Laufzeit des Programms herausfiltert, muss sie natürlich feststellen, ob die Komponente sich zurzeit im Formular-Designer oder in der freien Natur befindet. Dazu bedarf es keiner Geheimabsprache zwischen VCL und Delphi-IDE, sondern lediglich der Abfrage einer dokumentierten Schnittstelle. Wenn Sie das Verhalten Ihrer Komponente ebenfalls davon abhängig machen wollen, ob die Komponente sich noch im Formular-Designer befindet, können Sie dieselbe Schnittstelle benutzen.

Dazu gehört zuerst einmal das Property *TComponent.ComponentState*, das eine Menge von Flags enthält, zu denen auch *csDesigning* gehören kann. Ist das der Fall, befindet sich die Komponente mitsamt ihren Eltern, Besitzern und Kindern im Entwurfsprozess

innerhalb der IDE. Sie können dann sogar Zugriff auf eine abstrakte Formular-Designer-Schnittstelle erhalten, welche in der Unit *Forms* deklariert ist und über die Sie in gewissem Umfang mit der IDE kommunizieren können.

Kommunikation mit dem Designer

Wird das Formular in der Delphi-IDE bearbeitet, so weist sein Property *Designer* auf eine Schnittstellenvariable des Typs *IDesignerHook*, über die Sie mit einem Designer-Objekt der IDE kommunizieren können. Diese Kommunikation ist natürlich der Einschränkung unterworfen, dass Sie maximal die in der abstrakten Klasse definierten Methoden aufrufen können, und nicht alle Methoden, die der Designer tatsächlich beinhaltet.

IDesignerHook enthält beispielsweise die Methode *IsDesignEvent*, die den Designer fragt, ob er eine bestimmte Nachricht bearbeitet. Mit dieser Methode filtert die *TWinControl.MainEventFilter* zur Entwurfszeit die meisten Eingabenachrichten aus dem Nachrichtenfluss heraus.

Für die Programmierung einer Komponente kann besonders die Methode *Modified* wichtig sein (definiert im Interface *IDesignerNotify*). Sie teilt der IDE mit, dass sich die Komponente geändert hat, so dass beispielsweise der Objektinspektor Werte neu ausgeben kann (siehe Kapitel 6.6.6).

Hinweis: Bis Delphi 5 heißt der Schnittstellentyp *IDesigner*. Die oben genannten Methoden sind jedoch auch in *IDesigner* definiert.

6.5 Erweiterung bestehender Komponenten

In diesem Kapitel wenden wir uns wieder der Praxis zu, und zwar der einfachsten Art, eigene Komponenten zu schreiben. Diese besteht darin, eine bereits existierende Komponente zu verändern oder zu erweitern. Besonders einfach ist die erstgenannte Möglichkeit, da es zum Verändern einer Komponente bereits genügt, Property-Definitionen zu überschreiben.

6.5.1 Verändern bestehender Komponenten

In diesem Abschnitt ist mit dem »Verändern« einer Komponente all das gemeint, was ohne das Hinzufügen von neuen Methoden, Variablen oder Properties geschehen kann. In diesem Sinne war die *TNumEdit*-Komponente aus Kapitel 6.2.4 schon zu groß, denn sie erweiterte ja die vordefinierte *TEdit*-Komponente um ein neues Property.

Häufig genutzter Ansatzpunkt für Veränderungen an bestehenden Komponenten sind die schon vorhandenen Properties. In einer abgeleiteten Komponentenklasse können Sie die folgenden Eigenschaften der geerbten Properties ändern:

- den Zugriffsschutz,
- mit der *default*-Direktive als »voreingestellt« deklarierte Werte,
- die mit der *stored*-Direktive vorgenommene Speicherungseinstellung sowie
- Lese- und Schreib-Methoden.

Die wohl einfachste Veränderung an Komponenten besteht darin, bisher nicht veröffentlichte Properties zu veröffentlichen. Damit der Benutzer nicht von einer extrem langen Eigenschaftsliste im Objektinspektor überrascht wird, lassen viele Komponenten der VCL einen großen Teil der Properties unveröffentlicht. Meistens geschieht das auch aus dem Grund, dass die Properties bei der jeweiligen Komponente keine sinnvolle Funktion ausüben. Wie bereits in Kapitel 6.3.1 gezeigt, brauchen Sie in Ihrer Komponentenklasse ein geerbtes Property nur kurz in dem Schutzbereich (*public*, *published* usw.) zu nennen, in dem Sie es haben wollen.

Ändern der Default-Werte

Wenn Sie mit den Vorbelegungen der Properties einer bestimmten Komponente im Objektinspektor unzufrieden sind, können Sie eine neue Komponente von dieser ableiten, um den Default-Wert zu überschreiben. Im Prinzip brauchen Sie den gewünschten Wert nur in einem neuen Konstruktor der Komponente in das Property zu schreiben. Wenn Sie z.B. eine Panel-Komponente ohne anfängliche Beschriftung wünschen, könnten Sie folgende Komponente definieren:

```
type
  TEmptyPanel = class(TPanel)
  public
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TEmptyPanel.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Caption := ' ';
end;
```

In diesem speziellen Fall muss der String mindestens ein Leerzeichen enthalten, weil die Beschriftung sonst beim Einzeichnen der Komponente in das Formular wieder auf den vorläufigen Namen des Panels gesetzt wird. Das obige Beispiel ist allerdings für neue Default-Werte unvollständig, weil es keine *Default*-Direktive verwendet. Bei Pro-

perties, die einen ordinalen Typ, einen Zeigertyp oder einen Mengentyp haben, können Sie den Default-Wert wie folgt deklarieren (zusätzlich zu einer Zuweisung im Konstruktor):

```
property ModalResult default mrOK;
```

Die Folgen einer solchen Deklaration sind in Kapitel 6.4.3 erläutert. Beispiele für Default-Werte gibt die Klasse *TColorPalette* in Kapitel 6.6; auch *TScrollBoxEx* in Kapitel 6.5.2 enthält einen *Default*-Wert.

Hinweis: Komponenten, in denen nur die Property-Voreinstellungen geändert sind, können Sie, ohne eine neue Komponentenklasse zu definieren, auch als Komponentenvorlage definieren, indem Sie die Komponente einmal in einem Formular vorbereiten und dann mit KOMPONENTE | KOMPONENTENVORLAGE ERZEUGEN in die Komponentenpalette einfügen.

Überschreiben virtueller Methoden (Schriftwahl-Kombobox) *R49*

Das folgende Beispiel einer Kombobox, die eine Liste der im System installierten Schriftarten enthält und dabei die Schriftarten gleichzeitig anzeigt (Abbildung 6.5), erweitert die vorhandene Komponente *TComboBox* ebenfalls ohne neue Properties und Methoden. Sie überschreibt zwei geerbte virtuelle Methoden und den Konstruktor:

```
type
  TFontComboBox = class(TComboBox)
  protected
    procedure Loaded; override;
  public
    constructor Create(AOwner: TComponent); override;
    procedure DrawItem(Index: Integer; Rect: TRect;
      State: TOwnerDrawState); override;
  end;
```

Abbildung 6.5: Eine TFontComboBox-Komponente mit DropDownCount = 5 zur Laufzeit

Im Konstruktor wird die Kombobox durch Ändern des *Style*-Properties zu einer besitzergezeichneten Komponente gemacht:

```
constructor TFontComboBox.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Style := csOwnerDrawFixed; // besitzergezeichnet mit fester Höhe
end;
```

Nachdem alle automatischen Initialisierungen der Komponente während des Ladevorgangs abgeschlossen sind, kopiert sie in der *Loaded*-Methode die Liste der Schriftarten aus dem globalen *TScreen*-Objekt in ihr eigenes *Items*-Property:

```
procedure TFontComboBox.Loaded;
begin
  inherited Loaded;
  Items.Assign(Forms.Screen.Fonts);
end;
```

Nun muss sie noch dafür sorgen, dass jeder Eintrag auch mit seiner eigenen Schriftart ausgegeben wird. Der Name für die Schriftart des *i*-ten Eintrags ist also in *Items[i]* selbst gespeichert. Die allgemeinen Grundlagen besitzergezeichneter Komponenten sind in Kapitel 4.4.4 beschrieben, ein Beispiel für eine besitzergezeichnete ListBox finden Sie in Kapitel 4.5.4. Im Stil *csOwnerDrawFixed* erhalten alle Listeneinträge die Höhe, die der Benutzer im Property *ItemHeight* angibt.

Um auf die Nachricht zum Zeichnen eines Eintrags zu reagieren, verwendet die Komponente nicht das *OnDrawItem*-Ereignis, sondern überschreibt eine diesem Ereignis entsprechende virtuelle Methode:

```
procedure TFontComboBox.DrawItem(Index: Integer; Rect: TRect;
  State: TOwnerDrawState);
begin
  Canvas.FillRect(Rect);
  Canvas.Font.Name := Items[Index];
  Canvas.TextOut(Rect.Left, Rect.Top, Items[Index]);
end;
```

Da dies nur ein kleines Beispiel sein soll, belässt es die Methode beim Setzen des Schriftart-Namens. Andere Schriftattribute in *Canvas.Font*, vor allem die Größe, werden im Vertrauen auf eine sinnvolle Einstellung unverändert übernommen. So müssen Sie, um eine größere Schrift zu erhalten, im Objektinspektor *zwei* Properties von *TFontComboBox* manuell einstellen: *ItemHeight* und *Font.Height*.

6.5.2 Erweiterung von TScrollBox

Erstes Beispiel für eine kleine Ergänzung der Schnittstelle einer bestehenden Komponente soll die Klasse *TScrollBoxEx* sein, die im TreeDesigner benötigt wird:

- Das neue Ereignis *OnScroll* wird in Kapitel 5.5.4 dazu verwendet, während des fließenden Scrollvorgangs auch die Anzeige der Lineale des TreeDesigners zu aktualisieren (Methode *TDocumentForm.ScrollBoxScroll*).
- Die von *TScrollBox* geerbten, aber erst in *TScrollBoxEx* veröffentlichten Tastaturereignisse werden in Kapitel 5.8.1 genutzt, um Tastatureingaben in die Zeichenfläche des TreeDesigners abzufangen.

Noch nicht erwähnt wurde das Property *SmoothScrolling*, über das Sie das fließende Scrolling an- und ausschalten können. Es ist allerdings seit Delphi 4 überflüssig, da Sie nun auch über die Properties *TScrollBox.VertScrollBar/HorzScrollBar.Tracking* in den Genuss eines fließenden Scrollings kommen können.

Das folgende Listing zeigt die komplette Komponenten-Unit für *TScrollBoxEx* und gibt Beispiele für einfache, schon besprochene Komponentenaufgaben: von der Definition eines Default-Werts über das Abfangen von Windows-Ereignissen mit *message*-Methoden bis zum Aufruf eines Events:

```
unit ScrollboxEx;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics,
  Controls, Forms, Dialogs;

type
  TScrollNotifyEvent =
    procedure (Sender: TObject; Code: Word;  Horz: Boolean)
      of object;

  TScrollBoxEx = class(TScrollBox)
  private
    FSmoothScrolling: Boolean;
    FOnScroll: TScrollNotifyEvent;
  public
    constructor Create(AnOwner: TComponent); override;
    { Windows-Botschaft, die beim Bewegen der horizontalen
      Bildlaufleiste gesendet wird: }
    procedure WMHScroll(var Param: TWMScroll); message WM_HScroll;
    { selbiges für die horizontale Bildlaufleiste }
    procedure WMVScroll(var Param: TWMScroll); message WM_VScroll;
  published
    property SmoothScrolling: Boolean read FSmoothScrolling
```

```
                write FSmoothScrolling default True;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property OnScroll: TScrollNotifyEvent read FOnScroll
                write FOnScroll;
  end;

procedure Register;

implementation

constructor TScrollBoxEx.Create(AnOwner: TComponent);
begin
  inherited Create(AnOwner);
  FSmoothScrolling := True;
end;

procedure TScrollBoxEx.WMHScroll(var Param: TWMScroll);
{ Param enthält unter anderem die neue Scrollbar-Position (Pos)
  und den ScrollCode, der mit dem Scrollcode von
  TScrollBar.OnScroll übereinstimmt }
begin
  inherited;
  if Assigned(FOnScroll) then
    FOnScroll(self, Param.ScrollCode, True);
  if (Param.ScrollCode = SB_THUMBTRACK) and
     (FSmoothScrolling) then
  { die folgende Property-Zuweisung führt zur
    fließenden Verschiebung des Bildinhalts: }
    HorzScrollBar.Position := Param.Pos;
end;

procedure TScrollBoxEx.WMVScroll(var Param: TWMScroll);
begin
  inherited;
  if Assigned(FOnScroll) then
    FOnScroll(self, Param.ScrollCode, False);
  if (Param.ScrollCode = SB_THUMBTRACK) and
     (FSmoothScrolling) then
    VertScrollBar.Position := Param.Pos;
end;

procedure Register;
begin
  RegisterComponents('Zum Buch', [TScrollBoxEx]);
end;

end.
```

Die *TScrollBoxEx*-Komponente auf der CD ist gegenüber der obigen Version um ein weiteres Ereignis erweitert worden: *OnWantSpecialKey*, das Sie benötigen, wenn Sie Tastaturereignisse, die normalerweise zur Fokussteuerung im Dialog verwendet werden, in den Tastaturereignis-Methoden bearbeiten wollen; siehe hierzu Kapitel 5.8.1 und das Beispielprojekt *ScrollboxExTest*.

6.5.3 Die automatische History-Kombobox

Die Delphi-IDE macht regen Gebrauch von so genannten History-Listen, das sind Editierfelder, die sich die zuletzt gemachten Eingaben merken und in einer aufklappbaren Liste zur Wahl stellen, so dass Sie denselben Text nicht mehrmals eingeben müssen. Besonders hilfreich sind History-Listen, wenn sie zwischen zwei Sitzungen nicht verloren gehen (Delphi speichert sie in der Windows-Registry bzw. in seiner Konfigurations-Datei). Beispiele für häufig verwendete History-Listen sind die des *Suchen*- und die des *Auswerten/Ändern*-Dialogs.

Um eine History-Liste selbst zu verwalten, müssen Sie zur Programmlaufzeit alle Eingaben, die der Benutzer in das Eingabefeld macht, zur Liste der Eingabestrings hinzufügen, dabei doppelte Strings entfernen und die Liste wieder verkürzen, wenn sie zu lang wird. Besonders wichtig ist auch, dass Sie solche Listen zwischen zwei Programmläufen speichern.

THistoryCombo *R63*

Dies alles erledigt die Komponente *THistoryCombo* automatisch für Sie: Sie müssen die Komponente nur in ein Formular einfügen und in zwei Properties festlegen, unter welchem Schlüssel der Registry die Liste gespeichert werden soll, und schon erledigt *THistoryCombo* die gesamte Verwaltungsarbeit automatisch.

Diese beiden Properties sind *RegistryPath* und *IniSection*. Da die History-Listen normalerweise immer mit bestimmten Benutzern und nicht mit der Arbeitsstation verknüpft sind, verwendet *THistoryCombo* den voreingestellten Schlüssel *HKey_Current_User* (siehe Beschreibung der Registry in Kapitel 4.2). Unter diesem legt sie den Pfad *RegistryPath* an (z.B. *Software\MeineAnwendung*) und erzeugt *IniSection* als Unterknoten (z.B. *SuchdialogHistory*). *IniSection* entspricht damit einem Abschnitt einer INI-Datei, *RegistryPath* entspricht einer ganzen INI-Datei, in der Sie alle Informationen zu Ihrer Anwendung ablegen. Zwar ist ein Property wie *IniSection* für die Registry nicht unbedingt notwendig, da man *IniSection* einfach an *RegistryPath*+'\' anhängen könnte. Die Unterscheidung zwischen *RegistryPath* und *IniSection* fördert jedoch die Übersichtlichkeit und die Kompatibilität der Komponente mit Kylix. Die Kylix-Version der Komponente kann nämlich nur in eine INI-Datei schreiben (diese wird im Property *IniFileName* angegeben) und ein unter Delphi für Windows eingestellter Wert von *IniSection* lässt sich dann ohne Änderung weiterverwenden.

Von der Basisklasse *TComboBox* erbt *THistoryCombo* das Aussehen auf dem Bildschirm, die Bedienungsweise und das wesentliche Property *Items*, das den Inhalt der aufklappbaren Liste als *TStrings*-Objekt zur Verfügung stellt. Wir beginnen diesmal mit der vollständigen Klassendeklaration von *THistoryCombo*:

```
type
  THistoryCombo = class(TComboBox)
  { Combobox, die beim Laden aus einem Stream automatisch
    einen in der Registry oder einer INI-Datei gespeicherten
    Zustand wiederherstellt und die Liste bei Programmende
    wieder speichert. }
  private
    property Items;
  protected
    FRegistryPath: string;
    FIniFileName: string;
    FIniSection: string;
    FHistoryList: THistoryList;

    constructor Create(AOwner: TComponent); override;
     { in Destroy wird die Liste gespeichert: }
    destructor Destroy; override;
    procedure DoExit; override; { hier wird die Liste aktualisiert }
    procedure SetIniData(index: Integer; str: string);
{ Laden von HistoryList in die eigenen Items: }
    procedure UpdateComboList;
  public
     { Verwaltung der Stringliste (automatische Längenbeschränkung etc): }
    property HistoryList: THistoryList read FHistoryList;
    procedure UpdateList;
  published
    { Für Windows: Pfad der Speicherung in der Registrierung: }
    property RegistryPath: string index 0 read FRegistryPath
                                          write SetIniData;
    { Für Linux: Name der zu verwendenden INI-Datei: }
    property IniFileName: string index 1 read FIniFileName
                                          write SetIniData;
    { Name des zu verwendenden Registry-/INI-Datei-Abschnitts: }
    property IniSection: string index 2 read FIniSection
                                          write SetIniData;
  end;
```

THistoryList

Nun geht es darum, wie die Liste der vergangenen Strings verwaltet wird. *THistoryCombo* verwendet dazu nicht die geerbte *Items*-Liste, sondern ein Objekt der Klasse *THistoryList* aus den Kapiteln 4.1.2 und 4.2.4, die ebenfalls in der Unit *HList* definiert ist. *THistoryList* ist von *TStringList* abgeleitet und erweitert diese Klasse um das Management der INI-Dateien sowie um die Methode *AddString*, die einen String

genau so in die Liste einfügt, wie das für History-Listen erforderlich ist (als ersten Eintrag, unter Beibehaltung der Maximallänge der Liste und unter Beseitigung doppelter Einträge).

Der wichtigste Vorteil davon, für die History-Listenverwaltung eine eigene Klasse zu definieren, liegt darin, dass sich diese Klasse auch zu anderen Zwecken außerhalb von *THistoryCombo* verwenden lässt. So zeigt beispielsweise Kapitel 4.6.2, wie Sie *THistoryList* zur Buchführung über die zuletzt geladenen Dateien verwenden können (*THistoryList* kann die Einträge nämlich automatisch an ein Menü anhängen.)

Anlässe zum Verlängern der History-Liste

Die History-Kombobox soll die History-Liste völlig selbstständig verwalten, das heißt, dass Sie die neuen Einträge nicht von außen durch einen Methodenaufruf (beispielsweise vom Formular aus) hinzufügen müssen. Da jeder neue Eintrag in ihrem eigenen *Text*-Property steht, bereitet es der Kombobox keine Schwierigkeit, die neuen Einträge ausfindig zu machen.

Problematisch ist lediglich, den richtigen Zeitpunkt, genauer gesagt, das passende Event zur Erweiterung der Liste zu finden. Die üblichen Events wie *OnKeyDown* und *OnChange* sind nicht der passende Anlass für die Verlängerung der Liste, denn schließlich will der Benutzer nicht für jedes eingegebene Zeichen einen eigenen Eintrag in der Liste vorfinden. Zur Lösung dieses Problems sind mehrere Ansätze denkbar, die Unit *HList* löst die Aufgabe wie folgt:

Sie fügt den *Text* bei jedem *OnExit*-Ereignis in die History-Liste ein. *OnExit* tritt jedes Mal auf, wenn die Liste den Fokus verliert, wenn der Benutzer also z.B. auf einen normalen Schalter drückt oder zum nächsten Eingabefeld wechselt. Da *TWinControl* für das *OnExit*-Event die virtuelle Methode *DoExit* bereitstellt, überschreibt *THistoryCombo* diese:

```
procedure THistoryCombo.UpdateComboList;
begin
  { Die Kombobox-Liste mit der THistoryList neu aufbauen: }
  Items := FHistoryList;
end;

procedure THistoryCombo.UpdateList;
begin
  { Einfügen des Textes in die privat verwaltete THistoryList: }
  FHistoryList.AddString(Text);
  { Füllen der Kombobox-Liste mit FHistoryList: }
  UpdateComboList;
end;

procedure THistoryCombo.DoExit;
{ Fokus-Wechsel -> Eintrag speichern. }
```

```
begin
  inherited DoExit;
  UpdateList;
end;
```

Wenn Sie *THistoryCombo* in der Praxis ausprobieren, werden Sie wahrscheinlich feststellen, dass es sich damit gut arbeiten lässt. Trotzdem ist das alleinige Abfangen von *OnExit* nicht immer die optimale Lösung: Unter bestimmten Voraussetzungen kann der Benutzer den aktuellen Eintrag auch für irgendeine Aktion verwenden, ohne dass die Kombobox den Fokus verliert, so z.B. wenn er per ↵ den Standardschalter der Dialogbox betätigt oder wenn er einen SpeedButton anklickt, der ja den Tastaturfokus nicht erhalten kann.

Automatisches Registry-Management

THistoryCombo enthält die oben beschriebenen Properties, um die Speicherposition der Liste in der Windows-Registry bzw. in einer INI-Datei festzulegen. Wenn Sie diese im Objektinspektor setzen, brauchen Sie sich nicht mehr weiter um die Funktion und die Speicherung der History-Liste zu kümmern.

Bei der Implementierung dieser Automatik kommt es für *THistoryCombo* wieder auf die passenden Ereignisse an, bei denen die INI-Datei gelesen und geschrieben werden soll, wobei als »Ereignisse« in erster Linie wieder virtuelle Methoden in Betracht kommen.

Das Lesen der INI-Datei soll bei Programmstart stattfinden bzw. dann, wenn die Kombobox zu existieren beginnt. Im Konstruktor ist es jedoch noch zu früh dafür: Die Properties sind zu diesem Zeitpunkt noch nicht mit den im Objektinspektor eingestellten Werten geladen. Ein möglicher Zeitpunkt ist die virtuelle Methode *Loaded* (siehe Kapitel 6.4.2), die dann aufgerufen wird, wenn alle Properties aus der Formulardatei geladen wurden.

In diesem Fall genügt es jedoch, die History-Liste zu laden, wenn die Properties gesetzt werden; wir benötigen also eine Property-Schreibmethode, die die Liste beim Setzen der Properties aus der Datei lädt. Wie Sie der obigen Klassendeklaration entnehmen können, übernimmt eine einzige Methode, *SetIniData*, das Schreiben aller drei Properties. Über den Parameter *index* und den jedem Property zugewiesenen Index kann sie zwischen den drei Properties unterscheiden.

Innerhalb der Methode wird zwischen den Delphi-Versionen unterschieden: Nur in den Versionen für 32-Bit-Windows verwendet die Methode das Property *FRegistryPath*, da auch die Methode *THistoryList.LoadFromIni* aus Kapitel 4.2.4 nur dann die Registry verwendet. Außerdem stellt die Methode sicher, dass die INI-Datei nicht jedes Mal wenn eines der Properties gesetzt wird, sondern nur einmal geladen wird. Dies

geschieht dann, wenn *RegistryPath* und *IniSection* (unter Windows) bzw. *IniFileName* und *IniSection* (unter Linux) mit Werten gesetzt wurden:

```
procedure THistoryCombo.SetIniData(index: Integer; str: string);
begin
  case index of
    0: FRegistryPath := str;
    1: FIniFileName := str;
    2: FIniSection := str;
  end;
  { Automatisches INI-Datei-Lesen, sobald alle notwendigen
    Angaben gemacht sind: }
{$ifdef CompileForRegistry}
  if (FRegistryPath <> '') and (FIniSection <> '') then begin
    FHistoryList.LoadFromIni(FRegistryPath, FIniSection);
    UpdateComboList;
  end;
{$else}
  if (FIniFileName <> '') and (FIniSection <> '') then begin
    FHistoryList.LoadFromIni(FIniFileName, FIniSection);
    UpdateComboList;
  end;
{$endif}
end;
```

Einfach ist es auch, ein »Ereignis« für das Speichern der Liste zu finden: Hier bietet sich der Destruktor an, der quasi als Allerletzter das sinkende Schiff der freigegebenen Komponente verlässt (auch beim Destruktor handelt es sich um eine virtuelle Methode, die mit der *override*-Direktive überschrieben werden muss). Er ruft die *THistoryList*-Methode *SaveToIni* auf, gibt die History-Liste frei und ruft natürlich den geerbten Konstruktor auf:

```
destructor THistoryCombo.Destroy;
begin
  { Automatische Speicherung, wenn die notwendigen
    Angaben gemacht sind: }
  UpdateFile;
  { da FHistoryList nicht in der Komponentenliste ist,
    muss sie manuell freigegeben werden: }
  FHistoryList.Free;
  inherited Destroy;
end;

procedure THistoryCombo.UpdateFile;
begin
{$ifdef CompileForRegistry}
  if (FRegistryPath<>'') and (FIniSection<>'') then begin
    FHistoryList.UseRegistry:=True;
    FHistoryList.SaveToIni(FRegistryPath, FIniSection);
  end else
```

```
{$else}
  if (FIniFileName<>'') and (FIniSection<>'') then begin
    FHistoryList.SaveToIni(FIniFileName, FIniSection);
  end;
{$endif}
end;
```

Der Konstruktor

Das Einzige, was jetzt noch fehlt, ist ein Konstruktor, der das *FHistoryList*-Objekt der Komponente, deren Freigabe sich bereits im letzten Listingabschnitt befindet, per Konstruktor-Aufruf initialisiert. Properties mit Default-Werten sind nicht vorhanden, so dass dies die einzige neue Aufgabe für den Konstruktor bleibt:

```
constructor THistoryCombo.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FHistoryList := THistoryList.Create;
end;
```

Testen der Komponente

Wie schon erwähnt, genügt es, eine *THistoryCombo*-Komponente in das Formular einzufügen und die Properties für die Registry zu setzen, um die Komponente in Betrieb zu nehmen. Das Testprogramm *HlTest* (Abbildung 6.6) enthält außer der History-Liste einen Schalter, dem Sie zur Laufzeit mit Tab den Tastaturfokus verleihen können. Die History-Liste wird dann immer aktualisiert, wenn Sie sie mit Tab verlassen, und außerdem natürlich, wenn das Programm beendet wird (denn auch dann tritt ein *OnExit*-Ereignis auf). Unter Windows speichert *HlTest* die History-Liste im Pfad *Software\HistList* der Registry, unter Linux verwendet es die Datei *Hist.ini*.

Hinweis: Die History-Liste auf der CD ist um ein weiteres Detail erweitert: Wenn Sie die Erweiterung der Liste manuell vornehmen wollen, z.B. falls der Benutzer per ↵ einen vordefinierten Schalter drückt, können Sie die oben gezeigte Methode *UpdateList* manuell aufrufen (beispielsweise in der *OnClick*-Methode des Schalters). Damit *UpdateList* dann nicht von *DoExit* noch ein zweites Mal aufgerufen wird, setzen Sie das oben nicht gezeigte *THistoryCombo*-Property *ManualUpdate* auf *True* (dieses Property steht auch im Objektinspektor zur Verfügung).

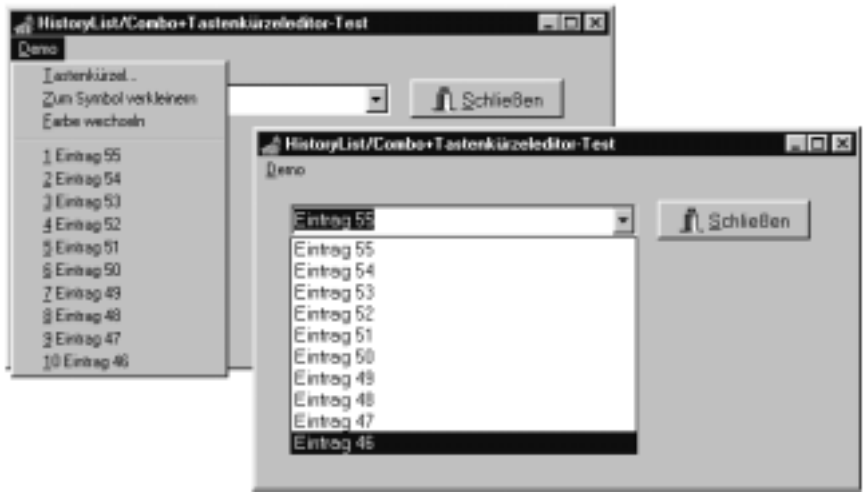


Abbildung 6.6: Das THistoryCombo-Testprogramm

6.5.4 Zusammenfassen mehrerer Komponenten

Als letztes kleines Beispiel für das Verändern bestehender Komponenten dient die Kombination zweier existierender Komponenten zu einer Komponente: die häufig vorkommende Kombination aus *TLabel* und *TEdit*.

Hinweis: Die hier entwickelte Komponente *TLabelEditCombi* unterscheidet sich grundlegend von der in Delphi 6 neu hinzugekommenen *TLabeledEdit*: Während die Delphi-Komponente, wie es ihr Name andeutet, ein spezialisiertes Editierfeld darstellt, also von *TCustomEdit* abgeleitet ist, besitzt *TLabelEditCombi* keine näheren Verwandtschaftsbeziehungen zu den Editierfeld-Klassen (sie soll ja auch als Beispiel dafür dienen, wie sich aus verschiedenen vorhandenen Komponenten völlig neue Komponenten zusammensetzen lassen). Dies wirkt sich auch auf die Handhabung der Komponenten beim Formularentwurf aus: Während Sie bei *TLabeledEdit* die Fläche des Editierfeldes vorgeben und das Label je nach Einstellung des *LabelPosition*-Properties automatisch an einer der vier Seiten des Editierfeldes positioniert wird, stellen Sie beim Formularentwurf mit der hier entwickelten Komponente eine Grundfläche ein, innerhalb der zwei Teilkomponenten automatisch angeordnet werden. Mit anderen Worten: *TLabelEditCombi enthält* ein Editierfeld, *TLabeledEdit ist* ein Editierfeld. *TLabeledEdit* nutzt außerdem (im Gegensatz zu *TLabelEditCombi*) die in Delphi 6 neu hinzugekommene Möglichkeit, Unterkomponenten im Objektinspektor anzuzeigen (hier unter dem Namen *EditLabel*).

Label und Edit als TCustomControl *R196*

Der Benutzer soll zur Entwurfszeit beide über einen gemeinsamen Rahmen verschieben und in der Größe ändern können. Um diesen Rahmen zur Verfügung zu stellen, muss die neue Klasse *TLabelEditCombi* selbst ebenfalls ein Steuerelement sein. Sie wird daher von der vordefinierten *TCustomControl* abgeleitet, die von sich aus nach außen unsichtbar ist, im Entwurfsmodus aber an der Markierung des besagten Rahmens erkannt werden kann (siehe Abbildung 6.7).

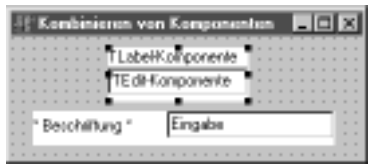


Abbildung 6.7: Die weiße Hintergrundfarbe der beiden TLabelEditCombi-Komponenten soll die von diesen eingenommene Fläche hervorheben; die in ihnen enthaltenen Einzelkomponenten lassen sich im Entwurfsmodus nicht selektieren.

Ein Nachteil gegenüber der in Kapitel 3.7 behandelten Frame-Technik ist, dass sich die beiden in *TLabelEditCombi* enthaltenen Komponenten nicht visuell zusammenstellen lassen wie beim Entwurf eines Formulars. Dafür sind die beiden Komponenten später im Formular von außen nicht direkt ansprechbar und bleiben so voll unter der Kontrolle der neuen Komponente (je nach Einsatzzweck bedeutet diese Kapselung entweder einen Vorteil oder einen Nachteil).

Das folgende Listing zeigt das Grundgerüst der neuen Komponente. Es besteht aus den intern verwendeten Instanzen von *TLabel* und *TEdit*, aus einem überschriebenen Konstruktor und aus den zwei Properties *Text* und *Caption*, über die die beiden wichtigsten Properties von Beschriftungs- und Editierfeld nach außen hin zugänglich gemacht werden:

```
TLabelEditCombi = class(TCustomControl)
private
  FLabel: TLabel;
  FEdit: TEdit;
  procedure SetCaption(const Value: String);
  procedure SetText(const Value: String);
  function GetCaption: String;
  function GetText: String;
protected
  constructor Create(AOwner: TComponent); override;
published
  property Text: String read GetText write SetText;
  property Caption: String read GetCaption write SetCaption;
end;
```

Der Konstruktor hat die Aufgabe, die beiden »Unterkomponenten« dynamisch zu erzeugen und ihre Properties einzustellen:

```
constructor TLabelEditCombi.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);

  FLabel := TLabel.Create(self);
  with FLabel do begin
    Parent := self;
    AutoSize := True;
    Visible := True;
    Caption := '* Beschriftung *';
    SetBounds(0, 0, 100, Height);
  end;

  FEdit := TEdit.Create(Self);
  with FEdit do begin
    Parent := self;
    Visible := True;
    SetBounds(110, 0, 200, Height);
  end;
end;
```

In den Property-Zugriffsmethoden werden sowohl Lese- als auch Schreibzugriffe einfach auf die entsprechenden Properties der Unterkomponenten weitergeleitet, zum Beispiel auf das *Caption*-Property:

```
function TLabelEditCombi.GetCaption: String;
begin
  Result := FLabel.Caption;
end;

procedure TLabelEditCombi.SetCaption(const Value: String);
begin
  FLabel.Caption := Value;
end;
```

Entsprechend funktioniert dies für beliebig viele weitere Properties. Im vorliegenden Fall ist sicher am wichtigsten, dass Position und Größe der Unterkomponenten angepasst werden können. *TLabelEditCombi* deklariert zu diesem Zweck keine weiteren Properties, sondern überschreibt die VCL-Methode *WMSize*, die bei jeder Größenänderung (der *TLabelEditCombi*-Komponente) aufgerufen wird. Da das *TLabel* wegen der *AutoSize*-Einstellung selbst für seine Größe sorgt, muss *TLabelEditCombi* nur noch für die Positionierung des Editierfelds sorgen. Diese findet in Abhängigkeit von der Größe des Labels statt. Nimmt das Label zu viel von der Breite der Kombi-Komponente weg, werden die beiden Komponenten untereinander positioniert.

```
procedure TLabelEditCombi.WMSize(var Message: TMessage);
  // in der VCL deklariert als:
    // procedure WMSize(var Message: TMessage); message WM_SIZE;
begin
  inherited; // geerbte Methode aufrufen
  if FLabel.Width > Width - 10 then
    FEdit.SetBounds(0, FLabel.Height + 2, Width, FEdit.Height)
  else
    FEdit.SetBounds(FLabel.Width, 0, Width - FLabel.Width, FEdit.Height);
end;
```

Die Position der Label-Komponente wird niemals geändert, da sie mit der Koordinate (0, 0) relativ zur oberen Ecke der Kombi-Komponente immer einigermaßen verwendbar ist. Diese automatische Positionierung könnte natürlich noch erheblich verbessert und flexibler gestaltet werden, aber die obige Version dürfte das Prinzip der Kombination zweier Komponenten zu einer Komponente genügend verdeutlichen.

6.6 Entwicklung neuer Steuerelemente

Dieses Kapitel beschreibt die umfangreichste der in diesem Buch behandelten Komponenten, die Farbpalette *TColorPalette*. Diese stimmt zwar in ihrem grundsätzlichen Ziel, den Benutzer Farben wählen zu lassen, mit Delphis Beispielkomponente *TColorGrid* überein, ist jedoch ansonsten erheblich flexibler und weist mehr Funktionen auf als *TColorGrid*. Der Haupteinsatzzweck von *TColorPalette* ist die Auswahl von Farben in Zusammenhang mit einer Grafik, während *TColorGrid* durch die Beschränkung auf die 16 Standardfarben nur für einfachere Zwecke dienen kann, z.B. zur Wahl der Textfarben in Delphis Quelltexteditor.

TColorPalette ist ein interaktives Steuerelement, das auf Mausaktionen reagieren muss und Events im Objektinspektor zur Verfügung stellt. Im Unterschied zur History-Kombobox aus dem letzten Kapitel erbt die Farbpalette jedoch dieses interaktive Verhalten und das äußere Erscheinungsbild nicht von einer Basisklasse, sondern muss es selbst implementieren.

Anhand der Farbpalette zeigt dieses Kapitel auch,

- wie Sie Property- und Komponenteneditoren schreiben,
- wie Sie Daten speichern, die nicht in Properties vorliegen, und
- wie Sie Popup-Menüs fest in Komponenten verankern und dem Benutzer gleichzeitig erlauben, sie zu erweitern.

6.6.1 Eine Testumgebung aufbauen

Die Farbpalette gehört zu den Komponenten, die so umfangreich sind, dass Sie sie wahrscheinlich nicht an einem Stück entwickeln wollen, sondern sie schon bei der Entwicklung hin und wieder testen wollen. Dank der Package-Technologie ist es sehr schnell möglich, eine bereits in der Komponentenpalette installierte Komponente zu aktualisieren und zu testen, indem Sie die Testanwendung als Projekt und den Quelltext der Komponente als zusätzliche Datei laden und dann die Komponente über einen Package-Editor neu übersetzen lassen, wenn Sie ihren Quelltext verändert haben.

Testen durch dynamische Erzeugung *R190*

Sie können eine Komponente jedoch auch schon testen, bevor Sie sie in der Komponentenpalette installieren, indem Sie die Unit der Komponente in Ihr Projekt einbinden und ein Exemplar der Komponente dynamisch zur Laufzeit anlegen.

So könnte die *OnCreate*-Methode eines solchen Testformulars beispielsweise wie folgt aussehen:

```
uses
  ..., ColorPal { muss von Hand eingebunden werden }
...

procedure TForm1.FormCreate(Sender: TObject);
begin
  with TColorPalette.Create(self) do begin
    Parent := self;
    { Properties können noch nicht im Objektinspektor
      eingestellt werden: }
    Align := alClient;
    Columns := 4;
    Lines := 4;
  end;
end;
```

Diese Methode funktioniert schon dann, wenn die Komponentenklasse noch aus dem Rumpf besteht, der vom »Komponenten-Experten« erstellten wurde. Mit einem solchen Testprogramm lassen sich die bei Delphi gewohnten schnellen Zykluszeiten von Testlauf zu Testlauf beibehalten.

Erst später, wenn zum Laufzeitverhalten der Komponente noch Funktionen hinzukommen, die schon zur Entwurfszeit funktionieren sollen, muss die Komponente in der Delphi-IDE installiert werden.

Versionsunterschiede zwischen Laufzeit und Entwurfzeit

Wie schon erwähnt können Sie bei einem geöffneten Package-Editor neue Versionen der Komponenten schnell in die Komponentenpalette installieren. Bis zu dieser manuellen Aktualisierung der Komponentenpalette verwendet Delphi im Formular-Editor weiter die alte Version einer Komponente – im Gegensatz zu einer Anwendung, bei der die Komponenten-Unit in die EXE-Datei eingebunden wird (also bei einer Anwendung, die nicht für die Benutzung von Laufzeit-Packages kompiliert wird). In einer solchen Anwendung wird die neue Komponentenversion verwendet, sobald Sie die Anwendung neu übersetzen (und dies geschieht beim nächsten Start der Anwendung aus der IDE automatisch, sofern der Pfad der Komponenten-Unit im Suchpfad des Projekts enthalten ist).

Während daher ein Testprogramm, das keine Laufzeit-Packages verwendet, immer die aktuelle Komponentenversion verwendet, enthält die Komponentenpalette der Delphi-IDE ohne manuelle Aktualisierung noch die alte Version. Solange Sie in der Komponente keine speziellen Dinge wie das Schreiben und Lesen von Daten aus der Formulardatei ändern, macht es nichts, wenn Sie Delphi mit einer leicht veralteten Komponentenversion arbeiten lassen und das Package nur ab und zu neu übersetzen. Jedoch können unterschiedliche Versionen einer Komponente zur Laufzeit und zur Entwurfszeit auch zu einer erheblichen Verwirrung des Entwicklers führen.

Debuggen von Entwurfszeit-Funktionen

Das schwierigste grundsätzliche Problem beim Testen einer Komponente dürfte wohl das Debuggen von Code sein, der von der Delphi-IDE aufgerufen wird. Falls ohne Umstände möglich, bietet es sich hier zunächst an, die Aufrufe, die Delphi durchführt, in einem eigenen Programm zu simulieren; im Fall von Property- und Komponenten-Editoren ist das beispielsweise sehr einfach. In anderen Fällen wie in einer Schreibmethode für Property-Daten ist es jedoch kaum möglich, den Code in dem Zusammenhang aufzurufen, in dem er auch von Delphi aufgerufen wird (wird dieser Code zur Laufzeit aufgerufen, fehlt die Schnittstelle zur Delphi-IDE).

Für solche komplizierten Fälle bietet Delphi die Möglichkeit, die Komponente tatsächlich im Delphi-Kontext zu debuggen. Wenn nämlich eine zweite Instanz der Delphi-IDE gestartet wird, kann der integrierte Debugger der ersten Instanz auch die laufenden Packages debuggen. Dazu geben Sie die EXE-Datei der IDE, also `delphi32.exe` im BIN-Verzeichnis der Delphi-Installation, unter START | PARAMETER im Feld HOST-ANWENDUNG ein. Auf den weiteren Vorgang treffen dann die allgemein für das Debuggen von DLLs gültigen Regeln zu (siehe das Ende von Kapitel 8.5.3).

6.6.2 Die Schnittstelle der neuen Farbpalette

Der abgeschlossene Charakter einer Komponente bringt es mit sich, dass die Schnittstelle der Komponente – sowohl zum Komponentenbenutzer als auch zum Anwender – schon zu Beginn einigermaßen klar umrissen sein sollte. Da die Farbpaletten-Komponente bereits auf CD-ROM vorliegt, können wir hier gleich mit der endgültigen Schnittstellen-Definition beginnen.

TColorPalette-Referenz

Das folgende Listing enthält die vollständige Klassendeklaration von *TColorPalette*, deren Elemente im weiteren Verlauf des Kapitels nach und nach zur Sprache kommen:

```
const
  MaxColors = 256; { mit 256 Farben verbraucht die Tabelle 1 kByte }

type
  TColorMark = (cmPen, cmBrush, cmFont);
  TColumns = 1..255;
  TRGBSteps = 1..5;

(** Methodenzeigertypen für die Events **)
  TColorChangeEvent =
    procedure(Sender: TObject; Color: TColor) of object;
  TPopupExpandEvent =
    procedure (Sender: TObject; var ExpandPoints: TPopupMenu)
       of object;
  TColorDefinitionEvent =
    procedure(Sender: TObject; Index: Integer;
              var ColorValue: TColor) of object;

(** Die Komponentenklasse **)
  TColorPalette = class(TCustomControl)
  private
  (* Events *)
    FOnPenColorChange,
    FOnBrushColorChange,
    FOnFontColorChange: TColorChangeEvent;
    FOnDefineColor: TColorDefinitionEvent;
    FOnExpandPopup: TPopupExpandEvent;
    FOnMouseDblClk: TMouseEvent;
  (* Variablen für Properties *)
    FColors: array[0..MaxColors-1] of TColor;
    SelColors: array[TColorMark] of Integer; { Indizes in FColors }
    FColumns,   { Zahl der im Fenster angezeigten Spalten }
    FLines: TColumns;  { Zahl der Zeilen }
    FSwapAlign: Boolean;   { dreht Spalten/Zeilen um}
    FNoSelection: Boolean; { schaltet die Selektierungsfunktion
                             der Maus aus  }
```

```
    FRGBSteps: TRGBSteps;
  (* eine Hilfsvariable *)
    RightClicked: Integer;
  (* Variable für die Entwurfszeit *)
    FSaveColors: Boolean;
  (* eine "Unterkomponente" *)
    ColorDialog: TColorDialog;
  private
  (* private Hilfsroutinen *)
    procedure InitColors;
    function GetColorOn(x, y: Integer): TColor;
    procedure CalcSides(var XSeite, YSeite: Integer);
    procedure DoSwapAlign(var X, Y: Integer);
    procedure InvalidateCell(i: Integer);
  (* Stream-Methoden *)
    procedure ReadColors(Reader: TReader); virtual;
    procedure WriteColors(Writer: TWriter); virtual;
  (* Property-Zugriffsmethoden *)
    function GetSelColor(Index: Integer): TColor;
    procedure SetSelColor(Index: Integer; NewValue: TColor);
    procedure SetSwapAlignFlag(Value: Boolean);
    procedure SetColumnNumber(Number: TColumns);
    procedure SetLineNumber(Number: TColumns);
    procedure SetRGBSteps(Val: TRGBSteps);
    function DummyFunc: Boolean;
  public
    (* GetColor und SetColor könnten normalerweise private
       sein (Propertylesemethoden!). Von Kapitel 6.8 (ActiveX)
       werden sie jedoch als public gewünscht: *)
    function GetColor(i: Byte): TColor;
    procedure SetColor(i: Byte; val: TColor);
  (* öffentliche Hilfsroutinen *)
    function PosToCell(x,y: Integer): Integer;
  (* überschriebene virtuelle Methoden *)
    constructor Create(AOwner: TComponent); override;
    procedure Paint; override;
    procedure MouseDown(Button: TMouseButton;
                        Shift: TShiftState; X, Y: Integer); override;
    procedure ChangeColorClick(Sender: TObject);
    procedure InitColorsClick(Sender: TObject);
    procedure DefineProperties(Filer: TFiler); override;
  (* Bearbeitung des Doppelklicks *)
    { zur Laufzeit }
    procedure WMLButtonDblClk(var Message: TWMLButtonDown);
       message WM_LBUTTONDBLCLK;
    { kann einen Doppelklick auch zur Entwurfszeit abfangen: }
    procedure WndProc(var Message: TMessage); override;
  (* geerbtes PopupMenu-Property darf nicht veröffentlicht sein: *)
    property PopupMenu stored False;
  (* ein unveröffentlichtes Property *)
    property Colors[i: Byte]: TColor read GetColor write SetColor;
```

```
published
(* Properties *)
  property SwapAlign: Boolean
     read FSwapAlign write SetSwapAlignFlag default False;
  property Columns: TColumns
     read FColumns write SetColumnNumber default 25;
  property Lines: TColumns
     read FLines write SetLineNumber default 5;
  property NoSelection: Boolean
     read FNoSelection write FNoSelection default False;
  property RGBSteps: TRGBSteps
     read FRGBSteps write SetRGBSteps default 5;
  property SaveColors: Boolean
     read FSaveColors write FSaveColors stored False;

  property PenColor: TColor index 0
     read GetSelColor write SetSelColor;
  property BrushColor: TColor index 1
     read GetSelColor write SetSelColor;
  property FontColor: TColor index 2
     read GetSelColor write SetSelColor;
  { ermöglicht den Zugriff auf den Property-Editor
    im Objektinspektor: }
  property ColorDummy: Boolean read DummyFunc stored False;

 (* Events *)
  { Benutzer setzt Farbmarke cmPen/cmBrush/cmFont neu: }
  property OnPenColorChange: TColorChangeEvent
     read FOnPenColorChange write FOnPenColorChange;
  property OnBrushColorChange: TColorChangeEvent
     read FOnBrushColorChange write FOnBrushColorChange;
  property OnFontColorChange: TColorChangeEvent
     read FOnFontColorChange write FOnFontColorChange;
  { Benutzer ändert einen Farbwert}
  property OnColorEdit: TColorEditEvent (* nur verwendet in Kapitel *)
     read FOnColorEdit write FOnColorEdit; (* 6.8.3 (ActiveX) *)
  { zu den folgenden beiden Events siehe zugehöriges Kapitel: }
  property OnDefineColor: TColorDefinitionEvent
     read FOnDefineColor write FOnDefineColor;
  property OnExpandPopup: TPopupExpandEvent
     read FOnExpandPopup write FOnExpandPopup;
  property OnMouseDblClk: TMouseEvent
     read FOnMouseDblClk write FOnMouseDblClk;

 (* zusätzliche Veröffentlichung von geerbten
    und geschützten Properties *)
  property Align;
  property OnEndDrag;
  property OnDragOver;
  property OnDragDrop;
  property DragMode;
```

```
    property OnMouseDown;
    property OnMouseUp;
    property OnMouseMove;
  end;
```

Bedienung der Komponente

Das Ziel der Farbpalette, Objekte mit Farben zu versehen, können Sie auf zwei verschiedene Weisen erreichen:

- Per Drag&Drop: Das Ziehen einer Farbe von der Palette auf ein grafisches Objekt liegt in der Verantwortung des Formulars, das die *TColorPalette*-Komponente benutzt. *TColorPalette* stellt zu diesem Zweck nur die von der VCL geerbten Drag&Drop-Events zur Verfügung. Kapitel 5.8.3 zeigt die Implementation dieses Drag&Drop im TreeDesigner.
- Per Markierung von Farben in der Palette. Drei dieser Farbmarkierungen sind vorgesehen, und zwar eine Vordergrund-, eine Hintergrund- und eine Schriftfarbe (anwählbar mit linker Maustaste, rechter Maustaste, sowie Shift+linker Maustaste). Wofür die gewählten Farben tatsächlich verwendet werden, liegt am Entwickler, der die Komponente benutzt. Im TreeDesigner werden sie beispielsweise dazu verwendet, die Farben der neu eingezeichneten und der markierten Objekte einzustellen.

Die Unit *ColorPal* definiert die drei Farbmarken in einem Aufzählungstyp, dessen Bezeichner sich im Namen an den Zeichenwerkzeugen der *TCanvas*-Klasse orientieren:

```
TColorMark = (cmPen, cmBrush, cmFont);
```

Damit eine Farbe gleichzeitig über alle drei Markierungen verfügen kann und alle drei Marken gleichermaßen lesbar bleiben, schreibt *TColorPalette* nicht die Kürzel »HG« und »VG« auf die gewählte Farbe (wie *TColorGrid*), sondern zeichnet eine Linie für die Stiftfarbe, ein leeres Rechteck für die Füllfarbe und ein »T« für die Schriftfarbe.

Die Properties Lines, Columns und die Farben

Die wichtigsten Properties von *TColorPalette* sind *Lines* und *Columns*. Sie zeigen an, aus wie vielen Zeilen und Spalten die Palette aufgebaut sein soll.

Intern organisiert *TColorPalette* die Farben im Array *FColors*, das bis zu 256 *TColor*-Werte aufnehmen kann. Dabei enthält das Array an den Indizes 0 bis *Columns-1* die Farben der ersten Zeile, gefolgt von den Farben der zweiten Zeile usw. Die Farben des Arrays werden also von links nach rechts am Bildschirm ausgegeben.

Die automatische Initialisierung der Farben können Sie durch das Property *RGBSteps* steuern. Dieses zeigt an, aus wie vielen verschiedenen Blau-, Grün- und Rot-Werten die Farben kombiniert werden. Wählen Sie mit *RGBSteps*-Werten zwischen 2 und 5, ob

die Farbpalette 8, 27, 64 oder 125 verschiedene Farbwerte enthalten soll. Bei jeder Änderung von *RGBSteps* initialisiert die Palette sämtliche Farben neu, Sie können diese Funktion also gut zur Entwurfszeit ausprobieren. Zur Berechnung der Farben siehe den Quelltext zu *TColorPalette.InitColors* auf der CD.

SwapAlign

Mit Hilfe des Flags *SwapAlign* können Sie die beschriebene Richtung der Farbanordnung umdrehen. Ist *SwapAlign* eingeschaltet, so bilden die ersten Felder von *FColors* nicht die erste Zeile, sondern die erste Spalte der am Bildschirm gezeichneten Palette. Darüber hinaus werden Spalten und Zeilen vertauscht, die Farben der ersten echten Spalte erhalten also die Indizes *0..Columns-1* (wobei hier das Property *Columns* durch die Vertauschung hier die Zeilen meint).

Für die Praxis bedeutet das, dass Sie die Palette mit *SwapAlign* umdrehen können und dass dabei eine vollständige Zeile zu einer vollständigen Spalte wird (anstatt dass die Zeile quasi mit einem neuen »Zeilenumbruch« auf mehrere Spalten zerstückelt wird, wie es der Fall wäre, wenn Sie einfach die Werte von *Columns* und *Lines* vertauschen).

Im Quelltext beschränkt sich der Einfluss des Properties *SwapAlign* auf die Methode *DoSwapAlign*. Diese wird von verschiedenen der in diesem Kapitel noch besprochenen Methoden zur möglichen Vertauschung von Zeilen- und Spaltenangaben verwendet, daher sei sie hier kurz aufgeführt:

```
procedure TColorPalette.DoSwapAlign(var X, Y: Integer);
var  Swap: Integer;
begin
  if SwapAlign then begin
    Swap:=X;   X:=Y;   Y:=Swap;
  end
end;
```

Die ausgewählten Farben

Das zweite Array-Property von *TColorPalette* ist *SelColors*. Es speichert die Farbindizes der Farben, die gerade ausgewählt sind. Um den *TColor*-Farbwert für die Stiftfarbe zu erhalten, benötigt die Palette also den folgenden Ausdruck:

```
FColors[SelColors[cmPen]];
```

Für den Benutzer der Komponente sollte natürlich eine einfachere Möglichkeit bestehen, die gewünschten Farbwerte zu erhalten. Zu diesem Zweck definiert *TColorPalette* die Properties *BrushColor*, *FontColor* und *PenColor*. Sie liefern direkt die *TColor*-Werte für die drei Farbmarken zurück.

Alle drei Properties sind auch beschreibbar. Das Schreiben übernimmt die Methode *SetSelColor*, die noch mehr zu tun hat als die Methode *GetSelColor*, die nur den oben gezeigten Ausdruck berechnen muss. Im einfachen Fall erhält *SetSelColor* einen *TColor*-

Wert als Parameter, der sich bereits unter den bis zu 256 aktuellen Farben von *FColors* befindet. Dann muss *SetSelColor* lediglich den Index *SelColors* auf die entsprechende Position von *FColors* setzen. Ist der Farbparameter noch nicht im Array *FColors*, ersetzt die Methode einfach die gerade gewählte Farbe durch die neue Farbe.

GetSelColor und *SetSelColor* sind weitere Beispiele für Methoden, die mehrere über einen Index unterschiedene Properties gleichzeitig behandeln, wobei sie den Property-Index als zusätzlichen Parameter erwarten. *GetSelColor* beispielsweise wandelt diesen Index in einen *TColorMark*-Wert um, um damit im Array *SelColors* den Index der gewünschten Farbe zu finden:

```
function TColorPalette.GetSelColor(Index: Integer): TColor;
begin
  Result := FColors[SelColors[TColorMark(Index)]];
end;
```

Mausfunktion

TColorPalette unterstützt neben dem Markieren von Farben noch zwei weitere Mausaktionen: Sie können mit Shift und der rechten Maustaste ein Popup-Menü aufrufen, mit dem Sie die Farben ändern und zurücksetzen können. Mit einem Doppelklick auf eine Farbe können Sie diese direkt editieren (für das Editieren ist der Standardfarbendialog zuständig).

Allerdings wird beim Doppelklick auch die Pinselfarbe gesetzt, denn der erste Teil des Doppelklicks erzeugt ein *MouseDown*-Ereignis. Außerdem wäre es wünschenswert, mit der rechten Maustaste das Popup-Menü direkt aufrufen zu können. Zu diesem Zweck definiert *TColorPalette* als vorletztes Property *NoSelection*; es ist hauptsächlich für den Paletteneditor in Kapitel 6.6.5 vorgesehen. Standardmäßig ist es auf *False* eingestellt, was bedeutet, dass Sie mit der Maus die drei verschiedenen Farbmarken setzen können. Wenn Sie dieses Property auf *True* setzen, werden die Farbmarken nicht mehr verändert. Das bedeutet, dass ein Doppelklick nun keine Nebeneffekte mehr hat und dass Sie das Popup-Menü mit der rechten Maustaste ohne Shift aufrufen können.

Die folgende Tabelle fasst die von *TColorPalette* bearbeiteten Mausaktionen bei ein- und ausgeschaltetem *NoSelection* zusammen:

NoSelection	Tastatur	Maus	Funktion
False		Links	Stiftfarbe wechseln
False		Rechts	Pinselfarbe wechseln
False	Shift	Links	Schriftfarbe wechseln
False	Shift	Rechts	Popup-Menü aufrufen
True		Rechts	Popup-Menü aufrufen
True/False		Doppelklick	Farbe editieren

Das letzte Property ist *SaveColors*, es ist hauptsächlich zu Demonstrationszwecken veröffentlicht und wird in Kapitel 6.6.6 besprochen.

Überschriebene Properties

In *TCustomControl* sind noch sehr viele häufig verwendbare Properties als *protected* deklariert. Es gehört daher zu den Standardaufgaben einer von *TCustomControl* abgeleiteten Klasse, sich daraus diejenigen Properties auszusuchen, die für ihre Zwecke am besten geeignet sind. Bei *TColorPalette* fällt die Wahl auf die Mausereignisse, die Drag&Drop-Ereignisse und die Properties *Align* und *DragMode*.

Events

Den letzten Teil der Komponentenschnittstelle für den Objektinspektor bilden die Events. Ein Anlass für Events liegt bereits auf der Hand: Wie die meisten anderen Steuerelemente auch, definiert *TColorPalette* Events, die bei Veränderungen durch den Anwender ausgelöst werden, hier also jedes Mal, wenn sich eine der drei Farbmarken ändert. *TColorPalette* definiert für jede Farbmarke ein eigenes Event: *OnPenColorChange*, *OnBrushColorChange* und *OnFontColorChange*

Dazu kommen noch zwei etwas exotischere Events: *OnDefineColor* und *OnExpandPopup*. Ersteres Event ermöglicht dem Benutzer der Komponente, die Standardfarbeinstellung der Palette zu überschreiben.

Auch das Event *OnMouseDblClk* ist selbst definiert, denn das geerbte geschützte Event *OnDblClick* gibt ein Beispiel für die Unterschlagung von Windows-Parametern: Es enthält nicht die Mausposition als Parameter, obwohl Windows diese mit dem Ereignis *WM_MouseDblClk* liefert. *TColorPalette* ruft *OnMouseDblClk* in seiner Methode *WMLButtonDblClk* auf, die auch den Farbauswahldialog zum Ändern einer Palettenfarbe aufruft.

6.6.3 Implementierung der Komponente

In diesem Abschnitt werfen wir einen Blick auf die Implementierung der Hauptfunktionen der Farbpalette. Dabei geht es vorwiegend um Quelltextteile, die spezifisch für Komponenten sind.

Zeichnen der Komponente *R87*

Beim Zeichnen der Farben muss *TColorPalette* auf ein Ereignis wie *OnPaint* verzichten, kann aber statt dessen die geerbte virtuelle Methode *Paint* überschreiben. Innerhalb einer solchen *Paint*-Methode haben Sie ein *Canvas*-Objekt zur Verfügung, so wie beim direkten Zeichnen in das Formular innerhalb einer *OnPaint*-Bearbeitungsmethode (für dieses *Canvas*-Objekt muss Ihre Komponentenklasse allerdings von *TCustomControl* oder *TGraphicControl* abgeleitet sein).

TColorPalette macht sich keine Mühe, einzelne Farben mit einer dreidimensionalen Umrandung zu versehen, so dass mehr Platz für die Farben bleibt und diese auch bei einer sehr kleinen Palette nicht zu Farbtupfern verkommen:

```
procedure TColorPalette.Paint;
var
  ColumnIndex, LineIndex, { Zählvariablen }
  XSeite, YSeite: Integer;   { Seitenlängen }
  x, y: Integer;      { Position des gerade gezeichneten Eintrags }
  Farbe: TColorRef;   { Farbe und ...}
  FarbIndex: Integer; { ...Index der Farbe dieses Eintrags }
  THeight, TWidth: Integer;

  procedure DoPenMark; { Setzen der Stiftfarbenmarkierung }
  begin
    if FarbIndex = SelColors[cmPen] then begin
      Canvas.MoveTo(x*XSeite, y*YSeite+YSeite div 2);
      Canvas.LineTo((x+1)*XSeite, y*YSeite+YSeite div 2);
    end;
  end;
  procedure DoBrushMark;
  procedure DoFontMark;
  ... { prinzipiell wie DoPenMark }
begin
  CalcSides(XSeite, YSeite);
  for LineIndex := 0 to Lines-1 do
    for ColumnIndex := 0 to Columns-1 do begin
      FarbIndex := LineIndex*Columns+ColumnIndex;
      Farbe := FColors[FarbIndex];
  (* Berechnung der x/y-Werte und der Farbe
        des gerade gezeichneten Eintrages, Einstellung
           der Attribute für das Farbrechteck *)
      y := LineIndex;
      x := ColumnIndex;
      DoSwapAlign(x, y);
      Canvas.Pen.Color := clBlack;
      Canvas.Pen.Mode := pmCopy;
      Canvas.Brush.Color := Farbe;
  (* Rechteck zeichnen *)
      Canvas.Rectangle(x*XSeite, y*YSeite,
                       (x+1)*XSeite+1, (y+1)*YSeite+1);
  (* Nun wird der Eintrag mit der Farbmarkierung
        gekennzeichnet, falls eine solche vorhanden ist: *)
      Canvas.Pen.Color := clWhite;
      { Der Stift soll auf jedem Hintergrund sichtbar sein: }
      Canvas.Pen.Mode := pmXor;
      DoBrushMark;  //
      DoPenMark;   // (siehe CD-ROM)
      DoFontMark; //
    end;
end;
```

Ungültigerklärung

Wie bei der Zeichenfläche des TreeDesigners ändern sich auch bei der Farbpalette oft nur kleine Teile der Bildschirmausgabe. Wenn Sie mit der Maus eine andere Farbe markieren, ändert sich die Darstellung der bisher markierten und der neu markierten Farbe. Es gibt auch hier wieder drei Möglichkeiten, diese Änderung sichtbar zu machen:

- Ungültigerklären der gesamten Palette durch einen *Invalidate*-Aufruf. Dies kommt einem Neuzeichnen der Palette gleich.
- Direktes Zeichnen der veränderten Farbrechtecke, beispielsweise in der Methode *OnMouseDown*.
- Ungültigerklären nur der veränderten Rechtecke.

Wie im TreeDesigner, so wurde auch hier die dritte Möglichkeit gewählt und in einer eigenen Ungültigerklärungs-Methode, die *InvalidateRect* aufruft, gekapselt (weitere Details zum Ungültigerklären mit *InvalidateRect* finden Sie in Kapitel 5.5.3):

```
procedure TColorPalette.InvalidateCell(i: Integer);
var
  XSeite, YSeite: Integer;   { Seitenlängen }
  x, y: Integer;
  R: TRect;
begin
  y := i div Columns;
  x := i mod Columns;
  CalcSides(XSeite, YSeite);
  DoSwapAlign(x, y);
  R := Rect(x*XSeite, y*YSeite, (x+1)*XSeite+1, (y+1)*YSeite+1);
  InvalidateRect(Handle, @R, False);
end;
```

Wichtigstes Einsatzgebiet für die neue Methode *InvalidateCell* ist das Aktualisieren zweier Farbzellen, wenn der Benutzer eine andere Farbe auswählt. Dies findet in der als Nächstes besprochenen Methode statt.

Verarbeitung der Mausereignisse

Auch die Mausereignisse stehen, abgesehen vom Doppelklick, in virtuellen Methoden zur Verfügung. Die Hauptaufgabe der neuen *MouseDown*-Methode von *TColorPalette* ist, die angeklickte Farbe herauszufinden und je nach Maustaste und dem Status von Shift eines der drei Farbänderungs-Ereignisse aufzurufen:

```
procedure TColorPalette.MouseDown(Button: TMouseButton;
                                  Shift: TShiftState; X, Y: Integer);
var
  ColorSelection: TColorMark;
```

```
begin
  if not NoSelection then begin
    if Button = mbRight then ColorSelection := cmBrush
      else if ssShift in Shift then ColorSelection := cmFont
        else ColorSelection := cmPen;

    InvalidateCell(SelColors[ColorSelection]);
    SelColors[ColorSelection] := PosToCell(X, Y);
    InvalidateCell(SelColors[ColorSelection]);

    case ColorSelection of
      cmPen: if Assigned(FOnPenColorChange) then
        FOnPenColorChange(self, FColors[SelColors[ColorSelection]]);
      cmBrush: if Assigned(FOnBrushColorChange) then
        FOnBrushColorChange(self,
           FColors[SelColors[ColorSelection]]);
      cmFont: if Assigned(FOnFontColorChange) then
        FOnFontColorChange(self,
           FColors[SelColors[ColorSelection]]);
    end;
  end;
  inherited MouseDown(Button, Shift, X, Y);
end;
```

Komponenten in Komponenten

Nachdem wir nun verschiedene Ereignisse durch virtuelle Methoden bearbeitet haben, kommen wir nun zum zweiten großen Unterschied zwischen der Komponenten- und der Formularprogrammierung, der *Verwendung* von Komponenten.

Wie in Kapitel 6.6.2 beschrieben, können Sie mit Shift+linker Maustaste ein lokales Popup-Menü für die Farbpalette aufrufen und mit einem Doppelklick die aktuelle Farbe in einem Standard-Farbdialog editieren. Wäre die Farbpalette ein Formular, hätte sie zur Entwurfszeit eine *TPopupMenu*- und eine *TColorDialog*-Komponente erhalten.

Als Komponente muss *TColorPalette* diese beiden Komponenten selbst erzeugen und auch die zugehörigen Variablen selbst deklarieren. Die Klassendeklaration enthält für den Dialog die Variable *ColorDialog*, das Popup-Menü wird nicht in einer neuen Variablen, sondern im geerbten Property *PopupMenu* gespeichert.

Der Konstruktor einer Komponente ist normalerweise immer der richtige Ort, Unterkomponenten zu erstellen und deren Properties zu initialisieren. Während die Erstellung eines Standarddialogs noch sehr einfach ist, besteht die Initialisierung eines

Popup-Menüs von Hand aus geschachtelten Aufrufen von Routinen der Unit *Menus*, die in Kapitel 4.6.1 beschrieben sind:

```
constructor TColorPalette.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
{ geerbte Properties anpassen: }
  Height := 100;
  Width := 100;
  PopupMenu := NewPopupMenu(self, 'internal', paRight, False,
    [NewItem('Farbe ändern', 0, False, True,
             ChangeColorClick, 0, 'intrn2'),
     NewItem('Farben zurücksetzen', 0, False, True,
             InitColorsClick, 0, 'intrn3')]);
{ eigene Properties anpassen: }
  RGBSteps := 5; { setzt auch Columns/Lines }
{ eine Komponente einfügen  }
  ColorDialog := TColorDialog.Create(self);
  ColorDialog.Options := [cdFullOpen];
{ den Rest erledigt: }
  InitColors;
end;
```

Nach der Ausführung des Konstruktors sind Popup-Menü und Farbdialog Teil der Komponente, wie sie auch Teil eines Formulars wären, wenn sie dort zur Entwurfszeit eingefügt worden wären. Die Programmierung dieser Komponenten unterscheidet sich also nicht mehr von der Programmierung der Komponenten eines Formulars.

Wichtig ist sowohl bei *TPopupMenu.Create* als auch bei *TColorDialog.Create*, dass die Komponente sich selbst als Elternkomponente und damit als ersten Parameter für diese Konstruktoren angibt. Das führt dazu, dass die VCL diese Komponenten automatisch freigibt, wenn die Farbpalette freigegeben wird (was wiederum automatisch geschieht, wenn das Formular freigegeben wird, in dem die Palette enthalten ist). Wenn Sie die Unterkomponenten nicht in den Besitz Ihrer Komponente überführen (indem Sie beispielsweise *nil* als ersten *NewPopupMenu*- oder *Create*-Parameter übergeben), müssen Sie diese Komponenten im Destruktor selbst freigeben.

Der Aufruf des Farbdialogs findet statt, wenn der Anwender den Popup-Menüpunkt *Farbe editieren...* auswählt oder wenn er auf die Farbe doppelklickt. Der Menüpunkt ist mit der Methode *ChangeColorClick* verknüpft, deren Dialogaufruf im folgenden Quelltextauszug gezeigt ist:

```
ColorDialog.CustomColors.Clear;
{ RightClicked enthält den Index der zu verändernden Farbe }
ColorDialog.CustomColors.
  Add('ColorA='+IntToHex(FColors[RightClicked], 6));
ColorDialog.Color := FColors[RightClicked];
if ColorDialog.Execute then begin
   ...
```

Interessant ist hier weniger der immer gleiche Aufruf von *Execute*, als die Initialisierung des Farbdialog-Properties *CustomColors*, bei dem es sich um eine Stringliste handelt (!). Um die erste Farbe des Dialogabschnitts *Selbst definierte Farben* zu setzen, fügen Sie der leeren Stringliste mit *Add* beispielsweise den String »ColorA=0A0B0C« hinzu, wobei der Farbwert hexadezimal angegeben werden muss und Sie bis »ColorP« gehen können.

Hinweis für Benutzer älterer Delphi-Versionen: Sollte dies mit Ihrer Delphi-Version nicht funktionieren, haben Sie vielleicht noch eine Version, in der der String 'Color' aus Versehen ins Deutsche übersetzt wurde. Verwenden Sie dann statt *Color* das Präfix *Farbe*.

6.6.4 Events mit Eingriffsmöglichkeiten

Neben den einfachen Farbänderungs-Events, deren Aufruf bereits im letzten Kapitel gezeigt wurde, definiert *TColorPalette* noch zwei weitere Events, die einen besonderen Unterschied zu den bisher beschriebenen Events aufweisen: Sie benachrichtigen das Formular weniger von einem Ereignis, als dass sie es ihm erlauben, das weitere Verhalten der Komponente durch Verändern eines Variablenparameters aktiv mitzugestalten.

OnExpandPopup *R192*

Da die Farbpalette ein eigenes Popup-Menü definiert, damit das *PopupMenu*-Property belegt und dieses sogar noch vor dem Komponentenbenutzer versteckt, könnte der Benutzer der Komponente kein Popup-Menü für die Farbpalette mehr definieren, wenn die Farbpalette keinen Ersatz für das Property *PopupMenu* bereitstellen würde.

Der Ersatz ist das Event *OnExpandPopup*, das über einen Variablen-Parameter verfügt, in dem der Komponentenbenutzer ein Objekt der Klasse *TPopupMenu* angeben kann. *TColorPalette* hängt dieses an das vordefinierte Popup-Menü an, sobald es aufgerufen wird. Die Voreinstellung des Parameters liegt bei *nil*, so dass der Benutzer nicht dazu verpflichtet ist, eine sinnvolle Methode zu schreiben. Der entsprechende Ausschnitt der Methode *MouseDown* (von der ein größerer Ausschnitt noch folgt) ist:

```
SecondMenu := nil;
if Assigned(FOnExpandPopup) then begin
  FOnExpandPopup(self, SecondMenu);
  if Assigned(SecondMenu) then...
```

Das Hauptformular des TreeDesigners, das die Farbpalette in einer Toolbar verwendet, nutzt dieses Ereignis und erweitert das Popup-Menü der Palette um die Punkte des Popup-Menüs, das auch für alle anderen Toolbars angezeigt wird (und über das die einzelnen Leisten angezeigt und versteckt werden können):

```
procedure TColorPaletteForm.PaletteExpandPopup(Sender: TObject;
  var ExpandPoints: TPopupMenu);
begin
  ExpandPoints := ControlBarPopup;
end;
```

Während die Benutzung dieses Events für den Benutzer äußerst einfach ist – er kann das Menüobjekt ja bequem mit dem Menü-Designer erstellen und muss dann nur eine einzeilige Methode wie *PaletteExpandPopup* schreiben –, muss *TColorPalette* das Menü in Handarbeit (mit erneuter Hilfe der in Kapitel 4.6.1 besprochenen Menüfunktionen) in ihr eigenes Menü verschieben und nach dem Aufruf wieder zurückbefördern:

```
procedure TColorPalette.MouseDown(Button: TMouseButton;
                                  Shift: TShiftState; X, Y: Integer);
var
  P: TPoint;
  SecondMenu: TPopupMenu;
  i, StartExpand: Integer;
  MovedItem: TMenuItem;
begin
  if (NoSelection and (Button = mbRight)) or
    ((Button = mbRight) and (ssShift in Shift)) then begin
  (* Position für das Menü feststellen *)
    RightClicked := PosToCell(X, Y); { Formularvariable }
    P.X := X; P.Y := Y;
    P := ClientToScreen(P);
  (* Erfragen eines Menüs durch den Event-Aufruf *)
    StartExpand := PopupMenu.Items.Count;
    SecondMenu := nil;
    if Assigned(FOnExpandPopup) then begin
      FOnExpandPopup(self, SecondMenu);
      if Assigned(SecondMenu) then begin
        (* Für den Fall, dass das erweiternde Popup-Menü irgendwelche
           Vorbereitungen treffen muss, wird nun dessen OnPopup-Ereignis
           aufgerufen. (Dies wurde für den TreeDesigner 2.5 notwendig.) *)
        if Assigned(SecondMenu.OnPopup) then
          SecondMenu.OnPopup(SecondMenu);
        (* Menüverschmelzung *)
        while SecondMenu.Items.Count > 0 do begin
          MovedItem := SecondMenu.Items[0];
          SecondMenu.Items.Delete(0);
          PopupMenu.Items.Add(MovedItem);
        end;
      end;
    end;
```

```
(* Menüaufruf *)
  PopupMenu.Popup(P.X, P.Y);
(* Eingefügte Punkte wieder zurückkopieren *)
  while PopupMenu.Items.Count > StartExpand do begin
    MovedItem := PopupMenu.Items[StartExpand];
    PopupMenu.Items.Delete(StartExpand);
    SecondMenu.Items.Add(MovedItem);
  end;
end;
... nun folgt die Behandlung der
      linken Maustaste (siehe Kapitel 6.6.3) ...
```

OnDefineColor

Eine weitere, ähnliche Eingriffsmöglichkeit erhält der Benutzer der Komponente mit dem Event *OnDefineColor*. Es wird jedes Mal aufgerufen, wenn eine Farbe auf ihren Standardwert zurückgesetzt wird, und erlaubt somit die Änderung der Standardfarben. Normalerweise geschieht dies immer mit allen Farben auf einmal – und zwar entweder bei der Konstruktion der Farbpalette oder wenn der Anwender aus dem Popup-Menü der Palette den Menüpunkt FARBEN ZURÜCKSETZEN auswählt.

Die mit *OnDefineColor* verknüpfte Methode erhält außer dem üblichen Absender-Parameter den Index der gefragten Farbe und einen Variablenparameter mit dem voreingestellten Farbwert. Wenn der Komponentenbenutzer diesen Wert ändern will, schreibt er ihn in diesen dritten Parameter.

Der Aufruf des Events findet in der Methode *InitColors* statt, nachdem das Array *FColors* bereits komplett initialisiert wurde:

```
if (Assigned(FOnColorDefine)) then
  for i := 0 to Columns*Lines-1 do begin
    UserDefinedColor := FColors[i];
    FOnColorDefine(self, i, UserDefinedColor);
    FColors[i] := UserDefinedColor;
  end;
```

Mit dieser Vorgehensweise hält sich *TColorPalette* an einen wichtigen Grundsatz der Komponentenentwicklung: Sie folgert aus der Tatsache, dass der Benutzer das *OnDefineColor*-Event bearbeitet, nicht, dass er in diesem den Wert von *ColorValue* setzt. Falls er das nicht tut, speichert der obige Code denselben Farbwert, der vorher in *UserDefinedColor* gespeichert wurde, wieder an Ort und Stelle zurück. Die vorgegebene Initialisierung der Farben wird also nicht durch zufällige Rückgabewerte der Event-Bearbeitungsmethode gestört.

Vorteil der Rückmeldungs-Events

Eine nicht gleichwertige Alternative zu *OnDefineColor* wäre es, wenn *TColorPalette* alle aktuell gültigen Farbwerte in einem Property zugänglich machen würde. Der Komponenten-Benutzer könnte dann zwar anfangs seine eigenen Werte in dieses Property schreiben, beim nächsten Farb-Reset mit dem Popup-Menüpunkt FARBEN ZURÜCKSETZEN wären diese Vorgaben allerdings außer Kraft gesetzt – wenn die Palette den Benutzer nicht wieder über ein Event von der Initialisierung informieren würde.

Das Event *OnDefineColor* erlaubt es dem Benutzer, auch nur eine einzelne aus 256 Farben mit einem neuen Wert zu definieren und sich in allen anderen Farben auf die Vorbelegung zu verlassen. Auch eine Berechnung der Standard-Farbwerte zur Laufzeit, in Abhängigkeit der Spalten- und Zeilenzahl der Farbpalette, wäre mit diesem Event sehr leicht möglich.

Auch wenn zumindest letztere Möglichkeit eher selten genutzt werden sollte, so gibt *OnColorDefine* doch ein gutes Beispiel dafür, wie sich die Schnittstelle einer Komponente erheblich flexibler gestalten lässt als bei ausschließlicher Verwendung von Properties. Das vorher besprochene Event *OnPopupExpand* hätte dagegen auch durch ein zweites *TPopupMenu*-Property ersetzt werden können.

6.6.5 Komponenten- und Property-Editoren

Die Farbpalette bietet eine gute Gelegenheit, Property- und Komponenteneditoren zu implementieren, mit denen sich eine Komponente noch weiter beeinflussen lässt als nur mit den Properties im Objektinspektor. Ebenfalls in der Unit *ColorPal* definiert sind:

- ein Property-Editor für das Farb-Property, in dem Sie die Farben der Palette einzeln editieren und Farbverläufe definieren können (Abbildung 6.8),
- ein Komponenteneditor, der eigentlich kein richtiger Editor ist, sondern lediglich aus zwei Menüpunkten besteht, die von der Delphi-IDE im Popup-Menü einer Farbpaletten-Komponente angezeigt werden (mit dem ersten wird der gerade erwähnte Editor für die Farben aufgerufen, mit dem zweiten werden die Farben auf die Voreinstellung zurückgesetzt).

Natürlich wäre es für den Anwender, der es letztlich mit der Palette zu tun bekommt, wichtiger, wenn die Palette die Farben zur Laufzeit ändern und in eigenen Palettendateien speichern könnte. Zum Lösen dieser Aufgabe benötigt jedoch das Formular, in dem die Palette später verwendet wird, lediglich zwei Speedbuttons, zwei Standard-Dateiwahldialoge und zwei *OnClick*-Methoden, weshalb wir uns hier sofort der größeren Herausforderung der Editoren zuwenden können. Die allgemeinen Grundlagen zu beiden Editortypen sind in Kapitel 6.4.4 beschrieben.

Entwurfszeit-Packages

Property- und Komponenteneditoren benötigen spezielle Units, die nicht in normalen Anwendungen, sondern nur in Packages enthalten sein können. Hierzu gehört beispielsweise die Unit *DesignEditors*, die einige weitere Units benötigt, die in Delphi nur innerhalb des Packages *designide* vorliegen (zu finden im *lib*-Verzeichnis der Delphi-Installation). Dieses Package müssen Sie der *requires*-Klausel des Packages hinzufügen, in dem sich Ihre Komponente befindet, für die Sie Editoren bereitstellen wollen.

Aus diesen Bedingungen folgt, dass Sie die Editoren in einer eigenen Unit definieren müssen, die nicht in normale Anwendungen eingebunden wird, wie im Abschnitt Auslagern der Editoren in eigene Units noch genauer erläutert werden wird.

Der Property-Editor *R194*

Ein Property-Editor ist ein Objekt einer von *TPropertyEditor* abgeleiteten Klasse. *TPropertyEditor* ist in der Unit *DesignEditors* (bis Delphi 5: *DsgnIntf*) deklariert und definiert eine Reihe von virtuellen Methoden, von denen Sie keine einzige überschreiben müssen, da keine davon als *abstract* deklariert ist.

Eine Beschreibung aller virtuellen Methoden würde den Rahmen dieses Kapitels bei weitem sprengen, daher müssen hier die im Beispiel-Editor verwendeten Methoden genügen:

- Die Methode *GetAttributes* liefert allgemeine Informationen über den Editor, beispielsweise, ob das Property aus einer Werteliste besteht (die der Objektinspektor in einer aufklappbaren Liste darstellt), ob es Unter-Properties hat oder ob es über einen Dialog editiert werden kann (in diesem Fall bringt Delphi einen entsprechenden Schalter im Objektinspektor an).
- Die Methode *Edit* wird dann aufgerufen, wenn der zuletzt genannte Schalter gedrückt wird. Die Methode kann dann das Formular des Editors aufrufen.
- Die Methode *GetValue* teilt Delphi mit, mit welchem Text das Property im Objektinspektor dargestellt werden soll.

Die Verwendung der anderen Methoden ist in der Online-Hilfe sowie im ToolsApi-Quelltext (`vcl\ToolsApi`) beschrieben und ist normalerweise sehr einfach bzw. ergibt sich in den konkreten Anwendungsfällen.

Der Editor für das Farb-Property der Palette ist in der folgenden Klasse *TPaletteEditor* definiert, deren Methode *GetAttributes* den Editor als Dialog ausgibt (der Editor wird also geöffnet, wenn Sie im Objektinspektor beim Farb-Property den Schalter mit den drei Punkten drücken) und das Textfeld im Objektinspektor vor Schreibzugriffen schützt (Attribut *paReadOnly*). Da nicht alle Farben in das Textfeld passen, kann die Funktion *GetValue* entweder einen festen Hinweistext wie »(Farben)« anzeigen oder

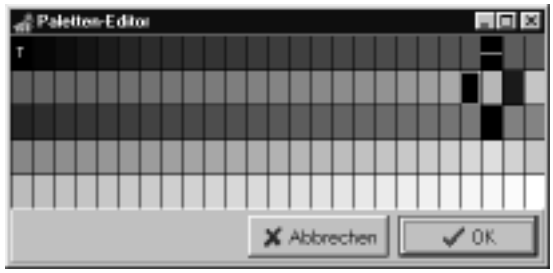


Abbildung 6.8: Der Editor der Palette besteht zum größten Teil selbst aus einer solchen.

sich irgendeinen Aspekt der Palette aussuchen, den sie anzeigen möchte. *TPaletteEditor* wählt letztere Möglichkeit und zeigt zur Demonstration den Mittelwert der Blauwerte aller Farben an (zwischen 0 und 255):

```
type
  TPaletteEditor = class (TPropertyEditor)
    procedure Edit; override;
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
  end;

function TPaletteEditor.GetAttributes: TPropertyAttributes;
begin  { teilt Delphi die Art des Property-Editors mit }
  GetAttributes := [paDialog, paReadOnly];
end;

function TPaletteEditor.GetValue: string;
var  { liefert den Text, der im Objektinspektor angezeigt wird }
  i, b: Word;
begin
  b := 0;
  with GetComponent(0) as TColorPalette do begin
    for i := 0 to Columns*Lines-1 do
      inc(b, GetBValue(FColors[i]));
    Result := 'Blau-Mittel = '+IntToStr(b div (Columns*Lines-1));
  end;
end;

procedure TPaletteEditor.Edit;
begin  { Aufrufen des Dialogs }
  EditPalette(GetComponent(0) as TColorPalette, Designer);
end;
```

Der eigentliche Editor wird in der Prozedur *EditPalette* aufgerufen, die dynamisch ein Editorformular erstellt, das als wichtigstes Element eine Farbpalette *Palette* enthält (dieses Formular ist hier nicht abgedruckt). *EditPalette* setzt die wesentlichen Properties des Paletteneditors auf die der Palette und lädt sie nach erfolgter Editierung wieder von dort zurück.

```
procedure EditPalette(EditedPal: TColorPalette; Designer: IDesigner);
var
  Dlg: TPalEditForm;
begin
  Dlg := TPalEditForm.Create(Application);
  with Dlg.Palette do begin
    { Die Editor-Palette soll aussehen wie die editierte Palette }
    NoSelection := True;
    FColors := EditedPal.FColors;
    Columns := EditedPal.Columns;
    Lines := EditedPal.Lines;
    Columns := EditedPal.Columns;
    SwapAlign := EditedPal.SwapAlign;
  end;
  { Aufruf des Editordialogs }
  if Dlg.ShowModal = idOK then begin
    { OK wurde betätigt -> Änderungen zurücklesen }
    EditedPal.FColors := Dlg.Palette.FColors;
    if Dlg.Palette.SaveColors then begin
      EditedPal.SaveColors := True;
      { Designer ist ein von TPropertyEditior geerbtes
        Element (und sollte eigentlich nie "nil" sein). }
      if Designer <> nil then
        Designer.Modified;
    end;
    EditedPal.Invalidate;
  end;
  Dlg.Free;
end;
```

Die wesentliche Funktion des Editorformulars ist eine Farbverlaufsfunktion: Das Formular verwendet die Drag&Drop-Ereignisse der Palette, um beim Ziehen einer Farbe auf eine andere Farbe einen Farbverlauf zwischen diesen beiden Farben zu erzeugen (siehe CD-ROM). Außerdem gestattet es der Editor natürlich, jede einzelne Farbe mit einem Doppelklick zu editieren. (Allerdings bedürfte es zu dieser Möglichkeit keines eigenen Property-Editors, denn die Farbpalette könnte den Doppelklick auch zur Entwurfszeit abfangen (in der Methode *WndProc*) und den Farbdialog aufrufen.)

Der eigene Property-Editor wird im Fall der Farbpalette allerdings durch den Komponenteneditor unnötig gemacht, denn dieser bietet ebenfalls eine Möglichkeit, das Editorformular aufzurufen, und verwendet dabei die obige Methode *EditPalette* wieder.

Der Komponenteneditor *R193*

Die Komponenteneditoren sind den Property-Editoren sehr ähnlich und in *DesignEditors* (bis Delphi 5: *DsgnIntf*) durch die Klasse *TComponentEditor* beschrieben. Wie schon beim Property-Editor folgt hier nur eine Beschreibung der in unserem Beispiel benötigten, also der von *TPaletteComponentEditor* überschriebenen virtuellen Methoden:

- In der Methode *GetVerbCount* teilen Sie Delphi mit, über wie viele verschiedene Verben / Befehle Ihr Editor verfügt.
- Sobald Delphi die Zahl der Verben kennt, kann es jeden einzelnen Befehl mit der Methode *GetVerb* erfragen. Überschreiben Sie diese Methode und geben Sie in Abhängigkeit vom als Parameter übergebenen Verb-Index den String zurück, der das Verb beschreibt. Delphi listet diese Strings im Popup-Menü der Komponente über den vordefinierten Punkten als Erstes auf.
- Nachdem Ihre Komponente die Verben mit dieser Methode aufgezählt hat, braucht sie nur noch darauf zu warten, das Delphi sie auffordert, eines dieser Verben auszuführen, weil der Benutzer den entsprechenden Menüpunkt ausgewählt hat. Hierfür ist die Methode *ExecuteVerb* zuständig, auch sie erhält wieder einen Verb-Index.
- Optional können Sie auch die Methode *Edit* überschreiben und darin definieren, wie die Komponente auf einen Doppelklick reagieren soll.

Der Komponenteneditor der Farbpalette bietet die Menüpunkte *Zurücksetzen* und *Farben editieren* an. Der erste Punkt setzt die Farben auf ihre Standardwerte zurück, die vom Property *RGBSteps* abhängig sind, während der zweite nur das eben definierte Property-Editorformular aufruft. Der Doppelklick ist eine alternative Möglichkeit, den Palletteneditor aufzurufen (Methode *Edit*):

```
type
  TPaletteComponentEditor = class (TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure Edit; override;
    procedure ExecuteVerb(Index: Integer); override;
  end;

implementation

function TPaletteComponentEditor.GetVerbCount: Integer;
begin
  Result := 2;
end;

function TPaletteComponentEditor.GetVerb(Index: Integer): string;
begin
  case index of
```

```
    0: Result := 'Zurücksetzen';
    1: Result := 'Farben editieren...';
  end;
end;

procedure TPaletteComponentEditor.Edit;
begin
  EditPalette(Component as TColorPalette, Designer);
end;

procedure TPaletteComponentEditor.ExecuteVerb(Index: Integer);
begin
  case Index of
    0: with (Component as TColorPalette) do begin
         InitColors;
         Invalidate;
       end;
    1: EditPalette(Component as TColorPalette, Designer);
  end;
end;
```

Registrieren der Editoren

Bei der Registrierung der Editoren kommt es wieder zu einem intensiven Einsatz von Typinformationen. Für einen Property-Editor übergeben Sie als letzten Parameter die von *TPropertyEditor* abgeleitete Klasse des Property-Editors. In den ersten drei Parametern geben Sie an, bei welcher Gelegenheit der Editor eingesetzt werden soll:

- Im ersten Parameter geben Sie Informationen über den Typ der Properties an, für die der Editor verwendet werden soll. Sie erhalten diese Informationen von der Funktion *TypeInfo*, der Sie den Typenbezeichner übergeben (beispielsweise *TypeInfo(Boolean)*).
- Optional können Sie im zweiten Parameter eine Klasse angeben, auf die Delphi den Einsatz des Editors beschränken soll.
- Aus dieser Klasse können Sie im dritten Parameter ein einzelnes Property auswählen, für das der Editor als Einziges anzuwenden ist. Sie geben dieses Property als String an.

Für den Property-Editor der Farbpalette stellt sich die Frage, für welches Property er überhaupt gelten soll, denn das Farben-Array *FColors* ist im Property-Editor wie alle anderen Array-Properties nicht zugänglich. *TColorPalette* definiert dazu das boolesche Property *ColorDummy*, das nur dazu dient, ein Feld im Objektinspektor für den Aufruf des Editors zu reservieren. (Eleganter wäre es, die Farben in einer eigenen Klasse zu definieren, so dass *FColors* zu einem Objekt würde und auch im Objektinspektor ange-

zeigt werden könnte.) Die Registrierung des Property-Editors findet in der Prozedur *Register* statt, die bereits für die Registrierung der Komponente und der Property-Kategorien zuständig ist:

```
procedure Register;
begin
  RegisterComponents('Zum Buch', [TColorPalette]);
   { (in diesem Beispiel spielt der Typ des Properties keine Rolle) }
  RegisterPropertyEditor(TypeInfo(Boolean), TColorPalette,
                         'ColorDummy', TPaletteEditor);
  RegisterPropertiesInCategory(...); // siehe Kapitel 6.3.1
```

Für den Komponenteneditor ist der Registrierungsvorgang sehr einfach, da Sie nur die Komponenten- und die Editorklasse miteinander in Beziehung bringen müssen. Die Registrierungsprozedur der Farbpaletten-Unit endet damit wie folgt:

```
  RegisterComponentEditor(TColorPalette, TPaletteComponentEditor);
end;
```

Auslagern der Editoren in eigene Units

Während in der oben gezeigten *Register*-Prozedur die Komponente und ihre Editoren und Kategorien gemeinsam registriert werden, finden Sie auf der CD eine Trennung zwischen Komponenten- und Editoren-Unit. *TPaletteEditor* und *TPaletteComponentEditor* sind in einer eigenen Unit namens *ColorPalDesignTime* untergebracht. Während diese die Unit *ColorPal* natürlich in ihrer *uses*-Klausel aufführt, ist das umgekehrt nicht der Fall, damit *ColorPal* völlig unabhängig von *ColorPalDesignTime* ist. Die oben gezeigten Aufrufe von *RegisterPropertyEditor*, *RegisterComponentEditor* und *RegisterPropertiesInCategory* finden in einer zweiten *Register*-Prozedur innerhalb von *ColorPalDesignTime* statt.

Diese Trennung hat zweierlei Vorteile:

- Es ist einfach möglich, getrennte Laufzeit- und Entwurfszeit-Packages von der Komponentenbibliothek des Buches herzustellen. Dazu muss im Laufzeit-Package gegenüber dem Entwurfszeitpackage lediglich die Unit *ColorPalDesignTime* entfernt werden. Da dies aber noch keine wesentliche Platzersparnis bringt und die Buchkomponenten möglicherweise nicht als Laufzeit-Package verwendet, sondern eher statisch zur EXE-Datei hinzugelinkt werden, ist dieser Fall nicht in die Praxis umgesetzt worden, das Package auf der CD ist also *nur* als *Entwurfszeit*-Package vorgesehen.
- Die Units des OpenTools API wie die schon erwähnte Unit *DesignEditors* können nicht in normale Anwendungen eingebunden werden und ihr Einsatz ist ja auch nur in Entwurfszeit-Packages sinnvoll. Wenn die *ColorPal*-Unit nun auch die Editorklassen enthalten würde, müsste sie solche ToolsApi-Units einbinden und könnte nun selbst nicht mehr in normalen Anwendungen genutzt werden, was ja ziemlich dem Zweck der Komponente widersprechen würde.

6.6.6 Speichern

Der Property-Editor für die Farben hat alleine jedoch noch nicht viel Nutzen, da sich die Farbwerte der Palette in einem Array-Property befinden, das die VCL grundsätzlich nicht automatisch speichern kann. Damit die geänderten Farben auch zur Laufzeit wieder sichtbar werden, benötigt *TColorPalette* weitere Methoden.

Zu einer an den Komponenten-Prinzipien orientierten Speicherungslogik gehören zwei Teile:

- Die Komponente muss eine Methode zur Verfügung stellen, mit der die VCL auch dieses nicht standardmäßige Property speichern kann. In diesem Fall muss die Methode 256 Farben, also theoretisch maximal ein KByte speichern.
- Vergleicht man den Platzbedarf von einem KByte mit den wenigen Bytes, die für gewöhnliche Properties genügen, erkennt man schnell, dass die Komponentenpalette die Farben nur speichern sollte, wenn diese auch geändert wurden. Die Komponente muss also eine Überprüfung durchführen, ob eine Speicherung notwendig ist.

Die Notwendigkeit des Speicherns

Wenn es sich bei den Farben um ein normales Property handelte, so könnten wir hinter diesem entweder einen *default*-Wert oder eine *stored*-Direktive angeben, anhand derer die VCL ebenfalls überprüfen könnte, ob die Speicherung notwendig ist,

```
property Colors: TImaginaryType read p1 write p2 stored AreColorsChanged;
```

wobei *AreColorsChanged* entweder eine Funktion oder eine Variable der Komponentenklasse ist. Zwar ist die oben gezeigte *stored*-Direktive bei den Farben der Palette nicht möglich, da die VCL nicht an der Speicherung beteiligt ist, trotzdem bleibt die Definition einer Variable oder Methode wie *AreColorsChanged* auch bei *TColorPalette* sinnvoll, wie sich gleich herausstellen wird (*TColorPalette* verwendet das Property *SaveColors*).

Nicht-Properties im Property-Tarnanzug *R199*

Grundsätzlich kann eine Komponente beliebige Daten in den Speicherungsprozess der VCL und der Delphi-IDE aufnehmen, indem sie die schon in *TPersistent* definierte virtuelle Methode *DefineProperties* überschreibt (siehe auch Kapitel 6.4.2). In dieser Methode kann eine Komponente ihre noch zu speichernden Daten nach Belieben aufteilen und gegenüber der VCL als Properties deklarieren. Eine solche Property-Deklaration besteht aus nahezu denselben Bestandteilen wie eine richtige Property-Deklaration (mit dem Unterschied, dass sie zur Laufzeit über einen Methoden-Aufruf stattfindet): Aus einem Property-Namen, aus einer Lese- und Schreibmethode und aus einem *Stored*-ähnlichen Flag.

Als Beispiel sehen wir uns zuerst die Schritte an, die im Fall der Farbpalette durchzuführen sind. Hierzu gehört zunächst die Definition der Lese- und Schreibmethode. Letztere erhält ein *TWriter*-Objekt als Parameter und tut nichts anderes, als die zu speichernden Farbwerte einzeln an dieses weiterzugeben:

```
procedure TColorPalette.WriteColors(Writer: TWriter);
var
  i: Integer;
begin
  for i := 0 to Columns*Lines-1 do
    Writer.WriteInteger(FColors[i]);
end;
```

Mit einer ähnlichen Methode liest die Komponente die Daten wieder aus einem *TReader*-Objekt:

```
procedure TColorPalette.ReadColors(Reader: TReader);
var
  i: Integer;
begin
  SaveColors := True; { siehe Abschnitt unten }
  for i := 0 to Columns*Lines-1 do
    FColors[i] := Reader.ReadInteger;
end;
```

Als Nächstes wird die virtuelle Methode *DefineProperties* überschrieben. *DefineProperties* gehört zu den wenigen von *TPersistent* eingeführten Methoden und erhält ein abstraktes *TFiler*-Objekt als Parameter. Hinter diesem verbirgt sich ein konkretes *TWriter*- oder *TReader*-Objekt.

Für die Methode *DefineProperties* spielt es keine Rolle, mit welchem dieser beiden Objekte sie es zu tun hat, sie ruft in beiden Fällen die *TFiler*-Methode *DefineProperty* auf, um das Schein-Property zu deklarieren. Wie angekündigt, werden hier ein Name für das Property, die Lese- und die Schreibmethode und schließlich ein Flag angegeben, das besagt, ob die Daten gespeichert werden müssen:

```
procedure TColorPalette.DefineProperties(Filer: TFiler);
begin
  inherited DefineProperties(Filer);
  Filer.DefineProperty('Colors', ReadColors, WriteColors, FSaveColors);
end;
```

Vielleicht wird Ihnen diese Vorgehensweise etwas merkwürdig vorkommen – warum können Sie z.B. nicht schon den Aufruf von *DefineProperty* davon abhängig machen, ob die Daten gespeichert werden müssen (*if FSaveColors then Filer.DefineProperties(...)*)? Diese Frage und die Notwendigkeit des Namensparameters lässt sich durch die Funktionsweise der Property-Speicherung erklären.

Funktionsweise

Alles in allem kommt es beim Speichern und Laden der als Properties verkleideten Daten zu einem beispielhaften Wechselspiel zwischen einem polymorphen *TFiler*-Objekt und einem polymorphen *TPersistent*-Objekt (hinter dem sich Ihre Komponente verbirgt). Dieses Wechselspiel wird hier am Beispiel des Schreibens einer erdachten Komponente *TNewComponent* erläutert:

- Nachdem *TWriter.WriteProperties* die Properties geschrieben hat, die die VCL automatisch speichern kann, sollen die anderen Daten gespeichert werden. Dazu ruft *TWriter* nun die Methode *DefineProperties* des zu speichernden Objekts auf (*TWriter* weiß an dieser Stelle nur, dass das zu speichernde Objekt eine von *TPersistent* abgeleitete Klasse hat). *TWriter* übergibt sich dieser Methode selbst als Parameter.
- In der Methode *TNewComponent.DefineProperties* wird das aufrufende *TWriter*-Objekt nun als abstraktes *TFiler*-Objekt angesehen. Ohne weiter darüber nachzudenken, zählt *TNewComponent* mit einer Reihe von *DefineProperty*-Aufrufen die zusätzlichen Properties auf.
- *DefineProperty* ist wieder an das aufrufende *TWriter*-Objekt gerichtet, das den Namen des Properties (angegeben im ersten Parameter) in den Stream schreibt und danach sofort wieder die Komponente aufruft, und zwar über die im dritten *DefineProperty* aufgerufene Schreibmethode (dies alles findet natürlich nur statt, falls der vierte Parameter *True* ist).
- So kommt es nun endlich zum Aufruf der *WriteData*-Methode von *NewComponent*. Diese schreibt die Daten, indem sie wieder Methoden von *TWriter* aufruft.

Beim Lesen der Komponente kann die VCL nun feststellen, ob das mit *DefineProperty* definierte Property gespeichert wurde, indem sie prüft, ob der im ersten Parameter angegebene Name im Stream steht. Falls dies nicht der Fall ist, ruft *TReader.DefineProperty*, die ebenfalls von *TNewComponent.DefineProperties* aufgerufen wird, die Methode *TNewComponent.ReadData* nicht auf.

Hinweis: Der vierte *DefineProperty*-Parameter hat beim Lesen also keinen Einfluss mehr. Die oben erwähnte Abfrage *if FSaveColors then Filer.DefineProperties(...)* in *TColorPalette.DefineProperties* würde also nicht funktionieren, da *DefineProperties* auch beim Lesen des Properties aufgerufen wird und erst die Methode *Filer.DefineProperties* erkennen kann, ob das Property gespeichert wurde oder nicht.

Das Property SaveColors

Es bleibt also noch die Aufgabe, die Variable zu definieren, die darüber entscheidet, ob gespeichert wird. *TColorPalette* initialisiert dazu im Konstruktor die Variable *FSaveColors* und ändert diese bei jeder Änderung einer Farbe.

Insbesondere dann, wenn die Farbwerte mit *ReadColors* gelesen werden, muss *FSaveColors* auf *True* gesetzt werden, damit die Daten bei der nächsten Speicherung wieder in den Stream geschrieben werden (*ReadColors* wird schließlich nicht nur beim Wiederherstellen des Formulars zur Laufzeit, sondern auch beim Laden des Formulars zur Entwurfszeit aufgerufen). Insofern ist das Flag *FSaveColors* besonders von normalen Veränderungsflags zu unterscheiden, die angeben, ob die Datei seit dem letzten Speichern geändert wurde. (Daher heißt *FSaveColors* auch nicht *FColorsChanged*.)

Kommunikation mit der Delphi-IDE

Da *FSaveColors* über das Property *SaveColors* veröffentlicht ist (zum Zwecke der Demonstration), können Sie das Umschalten des Flags bei Änderung einer Farbe auch im Objektinspektor sehen. Voraussetzung hierfür ist, dass *TColorPalette* Delphi darüber informiert, dass sich der Inhalt der Komponente und möglicherweise die Properties geändert haben. Dies geschieht mit der bereits in Kapitel 6.4.4 erwähnten Methode *IDesignerHook.Modified*, wobei *TColorPalette* vorher feststellen muss, ob ein solcher Designer überhaupt vorhanden ist. Es folgt der Teil der Methode *TColorPalette.ChangeColorClick*, der ausgeführt wird, wenn der Benutzer eine Farbe geändert hat:

```
FColors[RightClicked] := ColorDialog.Color; { Farbwert wird geändert }
InvalidateCell(RightClicked); { Farbzelle für ungültig erklären}
SaveColors := True; { Farben für das Speichern vormerken }
{ "Ungültig-Erklären des aktuellen Objektinspektor-Inhalts" }
if GetParentForm(self).Designer <> nil then
  GetParentForm(self).Designer.Modified;
```

Alle anderen Property-Änderungen finden im Objektinspektor selbst statt und müssen Delphi daher nicht extra über einen *Modified*-Aufruf mitgeteilt werden.

Hinweis: Wenn Sie ein Property im Objektinspektor ändern, liest Delphi auch die anderen Properties neu ein. Wenn ein Property andere Properties beeinflusst, werden diese Nebenwirkungen daher sofort sichtbar. Bei *TColorPalette* werden beispielsweise *Columns* und *Lines* durch *RGBSteps* beeinflusst, und in vielen Komponenten der VCL, die Text anzeigen, ändern Sie mit dem Property *Font.Height* auch das Property *Font.Size* und umgekehrt.

Binäre Properties

Ein besonderer Nachteil der oben abgedruckten Methoden zum Schreiben des Farb-Arrays ist ihre sehr geringe Effizienz. *TWriter* schreibt für jeden geschriebenen Integer-Wert zuerst eine Kennung (ob der Wert *Byte*, *Word* oder gar *LongInt*-Größe hat) und dann den eigentlichen Zahlenwert.

Um ähnliche Arrays effizienter schreiben und lesen zu können, sollten Sie ein datenintensives Property statt mit *TFiler.DefineProperty* mit der Methode *DefineBinaryProperty* definieren. Der Unterschied liegt in den Schreib- und Lese-Methoden, die Sie an diese Methode übergeben: Diese erhalten kein *TReader*- bzw. *TWriter*-Objekt als Parameter, sondern ein *TStream*-Objekt, das über effiziente Methoden zum Speichern von beliebig langen Speicherbereichen verfügt (siehe Kapitel 4.3). Da die Editiermöglichkeit der Palette zur Entwurfszeit sowieso eher demonstrativen Charakter hat, kann eine Optimierung hier jedoch ausbleiben.

6.6.7 Vordefinierte Aktionen

Zu guter Letzt eignet sich *TColorPalette* auch noch dazu, die Registrierung von selbst definierten Standardaktionen zu demonstrieren, was erst seit Delphi 4 möglich ist. Wenn Sie das Package mit *TColorPalette* in der IDE installiert haben, sollen Sie im Komponenteneditor von *TActionList* (siehe Kapitel 4.6.4) die Möglichkeit haben, mit NEUE STANDARD-AKTION... zwei Aktionen der Kategorie »Farbpalette« zu erzeugen, und zwar die beiden Aktionen, die Sie schon aus dem Popup-Menü der Palette aufrufen können: FARBE EDITIEREN... und FARBPALETTE ZURÜCKSETZEN. Im Dialogfenster *Standardaktionen* werden diese Aktionen unter den Namen *TColorPalChangeColor* und *TColorPalReset* auftauchen.

Die Erben der Klasse TAction *R195*

Beide Aktionen müssen von der Klasse *TAction* abgeleitet werden, da sie aber über eine wichtige Gemeinsamkeit verfügen, definiert die Unit *ColorPal* zuerst eine gemeinsame Basisklasse:

```
TColorPalAction = class(TAction)
public
  function HandlesTarget(Target: TObject): Boolean; override;
end;
```

Wie in Kapitel 4.6.4 beschrieben, testet die VCL auf der Suche nach einer Zielkomponente (*Target*) für eine Aktion verschiedene Komponenten daraufhin, ob sie vom Aktionsobjekt verarbeitet werden können. Dies geschieht durch einen Aufruf der *HandlesTarget*-Methode. *TColorPalAction* antwortet genau dann mit einem *True*, wenn es sich bei der getesteten Komponente um eine *TColorPalette* handelt:

```
function TColorPalAction.HandlesTarget(Target: TObject): Boolean;
begin
  Result:=Target is TColorPalette;
end;
```

Die beiden speziellen Klassen, in denen nun die beiden Standardaktionen definiert werden, können sich die Methode *HandlesTarget* schon sparen, denn sie erben sie direkt von *TColorPalAction*. Beide benötigen noch eine Methode *ExecuteTarget*, um die Aktion auszuführen. Außerdem erhalten sie noch einen Konstruktor, der das *Caption*-Property mit einer Vorbelegung versieht, so dass man, wenn man diese Standardaktion verwendet, den Menüpunkt-Text nicht mehr extra anzugeben braucht.

Da beide Klassen gleich aufgebaut sind, soll hier der Abdruck einer der Klassen zur Erläuterung genügen:

```
TColorPalChangeColor = class(TColorPalAction)
public
  constructor Create(AOwner: TComponent); override;
  procedure ExecuteTarget(Target: TObject); override;
end;

implementation

constructor TColorPalChangeColor.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  Caption:='Farbe editieren...';
end;

procedure TColorPalChangeColor.ExecuteTarget(Target: TObject);
begin
  (Target as TColorPalette).ChangeColorClick(Target);
end;
```

Die Methode *ExecuteTarget* erhält als Parameter immer ein Objekt, das vorher schon mit *HandlesTarget* als *TColorPalette* identifiziert wurde, und muss lediglich die schon bestehende Methode für den FARBE ÄNDERN...-Menüpunkt des Popup-Menüs aufrufen.

Schließlich muss die Delphi-IDE noch über die neuen Standardaktionen informiert werden, denn sie soll sie ja in einem ihrer Dialoge auflisten. Auch hier gibt es wieder eine Registrierungsfunktion, *RegisterActions*, welche in der schon bekannten Prozedur *Register* aufgerufen wird:

```
procedure Register; // Register-Prozedur der Komponenten-Unit
begin
  RegisterComponents('Zum Buch', [TColorPalette]);
  RegisterActions('Farbpalette', // << Aktions-Kategorie
                  [TColorPalChangeColor, TColorPalReset], nil);
end;
```

Da die Aktionen eine Laufzeitfunktion darstellen, werden sie in der Unit der Komponente selbst definiert, und nicht in einer eventuellen Laufzeit-Unit, die z.B. Property-Editoren und -Kategorien enthält (konkret in diesem Fall also in der Unit *ColorPal*, nicht in *ColorPalDesignTime*).

Dies ist quasi ein Minimalablauf zur Definition eigener Standardereignisse. Zu den zusätzlichen Möglichkeiten gehören:

- Die Anpassung des *Enabled*- oder *Checked*-Properties im Rahmen der *OnIdle*-Behandlung (entsprechend dem *OnUpdate*-Event eines *TAction*-Objekts wird in Standardaktions-Klassen die virtuelle Methode *UpdateTarget* verwendet).
- Die Erweiterung von *TAction* um neue Properties, die z.B. eine feste Zielkomponente vorgeben – Beispiele hierfür geben Delphis Datenbank-Standardaktionen, in denen Sie im *DataSource*-Property eine feste Datenquelle angeben können.

Wenn Sie die Professional oder eine höhere Version von Delphi haben, können Sie sich in den VCL-Quelltexten `DbActns.pas` und `StdActns.pas` den Code der Standardaktionen ansehen.

6.7 Formulare als Dialogkomponenten

Dieses Kapitel geht noch einen Schritt weiter als die Standard-Dialogkomponenten der VCL. Schon diese zeigen einige Vorteile gegenüber normalen Formularen wie z.B. die leichte Aufrufbarkeit über die Methode *Execute* und die bei allen Komponenten zu findenden Vorteile der Property-Einstellung im Objektinspektor. Zwar ist die *Execute*-Methode ein Vorteil gegenüber normalen Formularen, gegenüber anderen Komponenten ist sie jedoch eher ein Nachteil: die üblichen Steuerelementkomponenten funktionieren bereits, ohne dass sie zuerst durch eine Extramethode des Formulars aufgerufen werden müssen.

Die in diesem Kapitel vorgestellte Komponente ist eine Dialogkomponente, die diesen Aufruf von *Execute* unnötig macht und selbst für ihre Aktivierung sorgt, wenn Sie die entsprechende Option im Objektinspektor anschalten – ansonsten können Sie sie auch wie die Standarddialoge mit der *Execute*-Methode aufrufen. Das Ergebnis wird die Komponente *THotkeyManager* sein, die durch ihr alleiniges Vorhandensein in einem Formular mit Hauptmenü dafür sorgt, dass der Anwender die Tastenkürzel für die Menüpunkte beliebig anpassen kann, wobei die Änderung *persistent* sind, also zwischen zwei Programmläufen automatisch gespeichert werden.

6.7.1 Eine Hülle für das Formular

Wie jeder andere selbst erzeugte Dialog auch kann der Dialog für eine Komponente ein normales Formular sein. Dieses *ist* zwar eine Komponente (da von *TComponent* abgeleitet), kann aber nicht als solche in die Komponentenpalette eingebunden werden. Dafür bedarf es einer zusätzlichen nicht-visuellen Komponente als Verpackung, die dann im Formular so erscheint wie die Icons der Standarddialog-Komponenten.

THotkeyManager *R191*

THotkeyManager ist eine solche Hüllkomponente, die dazu dient, das eigentliche Dialogformular aufzurufen. Sie können ein solches Dialogformular als eigenständiges Projekt entwickeln, das Sie auch anderweitig verwenden können. *THotkeyManager* beispielsweise verwendet das Formular *THotkeyEditor*, das in Kapitel 4.6.3 beschrieben wurde (Abbildung 4.8) und bereits als normales Formular in den TreeDesigner eingebunden ist. Dieses Formular bleibt weiterhin in seiner eigenen Unit und kann dort innerhalb eines anderen Projekts visuell weiterentwickelt und verändert werden. Von der Komponenten-Unit aus lässt sich das Formular nur nicht-visuell über Programmcode ansprechen.

Für unsere Komponente stellt sich als Erstes die Frage, welche Klasse als Basisklasse dienen soll. Da es sich um eine nicht-visuelle Komponente handelt, kommen weder *TWinControl* noch *TControl* in Frage, auch *TCommonDialog* scheidet aus, weil diese nicht für Delphi-Formulare gedacht ist, sondern für Dialoge, die schon in Windows eingebaut sind. Letztendlich bleibt nur die Klasse *TComponent* übrig, von der *THotkeyManager* dann das Verhalten, sich zur Entwurfszeit als Icon darzustellen, sowie die veröffentlichten Eigenschaften *Name* und *Tag* erbt.

Das Verpacken des Formulars in der nicht-visuellen Komponente besteht nun lediglich darin, in *THotkeyManager* eine Instanz des Formulars zu deklarieren (Variable *HotkeyEditor*). Wie alle anderen Komponenten, die Sie nicht zur Entwurfszeit einfügen, müssen Sie auch solche eingepackten Formulare manuell initialisieren, indem Sie ihren Konstruktor aufrufen. Die Initialisierung, der Aufruf des Formulars und seine ab-

schließende Freigabe finden allesamt in der *Execute*-Methode der Hüllkomponente statt:

```
function THotkeyManager.Execute: Boolean;
var
  HotkeyEditor: THotkeyEditor;
begin
  Result := False;
  HotkeyEditor := THotkeyEditor.Create(Owner);
  if Assigned(ActionList) then
    HotkeyEditor.ActionList := ActionList
  else HotkeyEditor.Menu := Menu;
  if HotkeyEditor.ShowModal = mrOK then begin
    if FileName <> '' then
      SaveShortCutsToFile(FMenu, FileName);
    Result := True;
  end;
  HotkeyEditor.Free;
end;
```

Falls der Benutzer das Formular mit *OK* schließt, nimmt die Methode an, dass sich Tastenkürzeleinstellungen geändert haben, und speichert sie mit der hier nicht weiter interessanten Prozedur *SaveShortCutsToFile*, die in der Unit von *THotkeyEditor* enthalten ist.

6.7.2 Die automatische Aktivierung

Die automatische Aktivierung von *THotkeyManager* besteht darin, dass diese Komponente von selbst einen Menüpunkt an das Hauptmenü des Formulars anhängt und diesen mit einer eigenen Methode verknüpft, die den Tastenkürzel-Dialog aufruft. Die Erweiterung des Menüs findet mit Hilfe von einigen aus Kapitel 4.6.1 bekannten Methoden wie folgt statt:

```
Menu.Items[AutoMenuIndex].Add(
  NewItem(AutoMenuText, 0, False, True, AutoActivate, 0, ''));
```

Um den neuen Punkt gleich mit dem Aufruf des Tastenkürzel-Dialogs zu verknüpfen, wird in den obigen Zeilen die folgende Methode *AutoActivate* als *OnClick*-Methode für den neuen Menüpunkt angegeben:

```
procedure THotkeyManager.AutoActivate(Sender: TObject);
begin
  Execute;
end;
```

Verwendung der Komponente *R64*

Die folgenden Properties von *THotkeyManager* erlauben Ihnen, auf die automatische Aktivierung Einfluss zu nehmen:

- *AutoMenuIndex*: Hier geben Sie den Index des Hauptmenüpunkts an, hinter dessen Popup-Menü der Menüpunkt für den Tastenkürzeldialog eingefügt werden soll. Wenn Ihr fünfter Hauptmenüpunkt beispielsweise *Optionen* heißt, erreichen Sie mit einem *AutoMenuIndex* von vier, dass der automatische Menüeintrag am Ende des Optionsmenüs angehängt wird. Wenn Sie *Execute* manuell aufrufen wollen, müssen Sie *AutoMenuIndex* bei seinem voreingestellten Wert -1 belassen.
- In *AutoMenuText* können Sie den vorgegebenen Menütext »Tastenkürzel-Editor...« ändern.
- Wenn Sie in *FileName* einen Dateinamen angeben, lädt die Komponente die Tastenkürzel beim Programmstart automatisch und speichert sie nach jedem neuen Aufruf des Dialogs (in der bereits gezeigten Methode *Execute*).
- Auch das Property *Menu* ist optional, es gibt die Hauptmenükomponente an, in deren Popup-Menü sich die Komponente eintragen und dessen Tastenkürzel editiert werden sollen. Sie können es setzen, indem Sie aus der aufklappbaren Liste eines der im Formular vorhandenen *TMainMenu*-Objekte auswählen. Der letzte Dienst, den *THotkeyManager* anbietet, ist, das zu editierende Menü automatisch festzustellen. Wenn Sie im Property *Menu* keine Angabe machen, verwendet er statt dessen einfach das Hauptmenü des Formulars.
- Das Property *ActionList* wurde eingeführt, um auch die Tastenkürzel von Aktionslisten editieren zu können. Auch das Formular *THotkeyEditor* bietet dafür ein entsprechendes Property. In *THotkeyManager* ist es (anders als in *THotkeyEditor*) möglich, *ActionList* und *Menu* gleichzeitig zu setzen. In diesem Fall werden die Tastenkürzel aus *ActionList* editiert, und im *Menu* wird nur der automatisch erzeugte Menüpunkt zum Aufruf des Editors eingetragen. Wenn Sie die Tastenkürzel des Menüs editieren wollen, müssen Sie *ActionList* auf *nil* setzen (bzw. das Feld im Objektinspektor leer lassen).

Die automatische Menü-Erweiterung, die Auswahl des Menüs aus einer Dropdown-Liste und die automatische Feststellung des Menüs sind nur aus einer Komponente heraus möglich, die sich als Element in dem Formular befindet, dessen Menü verändert werden soll. Ohne die Komponente *THotkeyManager* als Verpackung könnte das Formular *THotkeyEditor* diese Funktionen nicht in dieser angenehmen Form anbieten.

6.7.3 Implementierung

Wir untersuchen zum Schluss zwei Aspekte von *THotkeyManager*, bei denen der Status der Komponente (Property *TComponent.ComponentState*) eine wichtige Rolle spielt und bei denen sich eine Gelegenheit ergibt, die virtuelle Methode *Loaded* zu überschreiben.

Das folgende Listing zeigt zunächst die Property-Schreibmethoden für *Menu* und *FileName*:

```
procedure THotkeyManager.SetMenu(NewMenu: TMainMenu);
begin
  if NewMenu <> FMenu then begin
    FMenu := NewMenu;
    if Assigned(FMenu) then begin
      ConnectToMenu;
      if FFileName <> '' then ReloadShortcuts;
    end;
  end;
end;

procedure THotkeyManager.SetFileName(NewName: string);
begin
  if FFileName <> NewName then  begin
    FFileName := NewName;
    if Assigned(FMenu) and (FFileName <> '')
       then ReloadShortCuts;
  end;
end;
```

SetMenu trägt den neuen Menüpunkt mit der Methode *ConnectToMenu* gleich in das Hauptmenü ein und lädt auch noch die Tastenkürzel, falls das Property *FileName* bereits gesetzt ist. Falls dies nicht der Fall ist, holt die Prozedur *SetFileName* dies nach, sobald *FileName* gesetzt wird. Beide Methoden sind also so geschrieben, dass Sie diese beiden Properties in beliebiger Reihenfolge setzen können und dass danach in jedem Fall die Initialisierung mit *ConnectToMenu* und *ReloadShortCuts* stattgefunden hat.

Hinweis: Allerdings setzt *THotkeyManager* voraus, dass die anderen beiden Properties, *AutoMenuIndex* und *AutoMenuText* bereits vorher gesetzt wurden. Um auch diese Bedingung zu beseitigen, müssten weitere Property-Schreibmethoden hinzugefügt und die oben gezeigten Methoden erweitert werden. Wenn Sie eine *THotkeyManager*-Komponente zur Entwurfszeit in das Formular einfügen, müssen Sie sich sowieso nicht um die Reihenfolge der Property-Initialisierung kümmern.

Der Unterschied zwischen Entwurfs- und Laufzeit *R200*

Das automatische Einbinden eines Menüpunkts zum Aufruf des Tastenkürzel-Editors ist zur Laufzeit sehr nützlich; da die Komponente jedoch auch von der Delphi-IDE ganz normal geladen wird, findet die Menüpunkt-Einbindung auch zur Entwurfszeit statt, falls keine zusätzlichen Vorkehrungen getroffen werden. Der neue Menüpunkt wäre im Entwurfszeit-Formular nicht nur sichtbar, sondern er würde auch noch in der Formulardatei als Teil des Menüs gespeichert werden. Beim nächsten Laden dieser Datei in die IDE würde die VCL diesen Punkt als Teil des Menüs laden, und *THotkeyManager* würde ihn schließlich ein zweites Mal hinzufügen. Bei jedem Laden des Formulars würde so ein weiteres Exemplar des Menüpunkts zum Menü hinzukommen.

Die einfachste Lösung, um dies zu vermeiden, besteht in der Abfrage, ob die Komponente sich im Entwurfsmodus der Delphi-IDE befindet (*csDesigning in ComponentState*), und das Menü nur dann zu erweitern, wenn das nicht der Fall ist. Diese Abfrage ist Teil der Methode *ConnectToMenu*, so dass die oben gezeigten Property-Schreibmethoden nicht geändert werden müssen:

```
procedure THotkeyManager.ConnectToMenu;
begin
  if not (csDesigning in ComponentState)
     and (AutoMenuIndex <> -1) then
  begin
    if not Assigned(FMenu) then
      FMenu := (Owner as TForm).Menu;
    if Assigned(FMenu) then FMenu.Items[AutoMenuIndex].
      Add(NewItem(AutoMenuText, 0, False, True, AutoActivate, 0, ''));
  end;
end;
```

Verwendung der Methode Loaded

Als Zweites soll nun der Ladevorgang der Komponente so angepasst werden, dass *ConnectToMenu* und *ReloadShortCuts* erst dann aufgerufen werden, wenn die Komponente fertig initialisiert ist und alle Properties gesetzt sind. Diese weitere Einschränkung ist hier zwar nicht zwingend erforderlich, stellt aber in der derzeitigen Version von *THotkeyManager* sicher, dass auch die Properties *AutoMenuIndex* und *AutoMenuText* richtig gesetzt sind, wenn der neue Menüpunkt zum Menü hinzugefügt wird.

Wie bei der ausführlichen Beschreibung des Ladeprozesses in Kapitel 6.4.2 erwähnt, ruft die VCL die virtuelle Methode *Loaded* auf, wenn sie die Komponente fertig aus der Formular- bzw. kompilierten Anwendungs-Datei geladen hat. Neben dem obligatorischen Aufruf der geerbten Methode holt *THotkeyManager* die Anpassung des Menüs in der überschriebenen *Loaded*-Methode nach:

```
procedure THotkeyManager.Loaded;
begin
  inherited Loaded;
```

```
    ConnectToMenu;
    ReloadShortCuts;
  end;
```

Um doppelte Aktionen zu vermeiden, darf nun während des Ladens jedoch keine Menüveränderung mehr stattfinden, wenn die VCL die Properties aus der Formulardatei lädt und die Methoden *SetFileName* und *SetMenu* aufruft. Daher fragen *ConnectToMenu* und *ReloadShortCuts* zusätzlich zum oben beschriebenen *ComponentState*-Flag *csDesigning* das Flag *csLoading* ab und führen ihre Aktion nur aus, wenn keines der Flags gesetzt ist. So wird beispielsweise die *if*-Bedingung der oben gezeigten Methode *ConnectToMenu* wie folgt erweitert:

```
if not (csDesigning in ComponentState)
   and not (csLoading in ComponentState)
   and (AutoMenuIndex <> -1) then
```

Es gibt also drei Möglichkeiten, wann *THotkeyManager* das Hauptmenü des Formulars anpasst:

- gar nicht, wenn sich das Formular im Entwurfsmodus befindet,
- in der Methode *Loaded*, wenn die Komponente zur Laufzeit aus den Formular-Ressourcen der EXE-Datei geladen wird, und
- sofort in den Schreibmethoden der Properties *FileName* und *Menu*, wenn Sie diese direkt im Quelltext setzen, beispielsweise nach einer dynamischen Erzeugung einer *THotkeyManager*-Komponente zur Laufzeit.

Die Klassendeklaration

Das Listing der Klassendeklaration fasst die besprochenen Teile von *THotkeyManager* zum Bilden der Hülle um das Formular, zum Definieren der Entwurfszeitschnittstelle, für die automatische Menüerweiterung und die automatischen Dateioperationen zusammen:

```
type
  THotkeyManager = class(TComponent)
  protected
    FMenu: TMainMenu;
    FActionList: TActionList;
    FAutoMenuIndex: Integer;
    FAutoMenuText: string;
    FFileName: string;
    procedure SetFileName(NewName: string);
    procedure SetMenu(NewMenu: TMainMenu);
    procedure SetActionList(NewList: TActionList);
    procedure ReloadShortCuts;
    procedure ConnectToMenu;
    procedure Loaded; override;
```

```
    procedure AutoActivate(Sender: TObject); { verknüpft mit OnClick }
    function MenuTextChanged: Boolean; { siehe stored von AutoMenuText }
  public
    constructor Create(AnOwner: TComponent); override;
    function Execute: Boolean;
  published
    property AutoMenuIndex: Integer read FAutoMenuIndex
                write FAutoMenuIndex default -1;
    property AutoMenuText: string read FAutoMenuText
                write FAutoMenuText stored MenuTextChanged;
    property Menu: TMainMenu read FMenu write SetMenu;
    property ActionList: TActionList read FActionList write SetActionList;
    property FileName: string read FFileName write SetFileName;
  end;
```

Einige Teile davon sind nicht besprochen worden, enthalten aber keine besonderen Neuigkeiten mehr, wie Sie im vollständigen Quelltext der CD überprüfen können.

6.8 ActiveX-Komponenten

Die Fähigkeit, aus *TWinControl*-Nachfolgerklassen automatisch ActiveX-Steuerelemente zu erzeugen, war 1997 eine der wichtigsten und meist beworbenen Neuerungen von Delphi 3 Professional. Da seither auf dem Gebiet der ActiveX-*Komponenten* keine großen Neuentwicklungen von Seiten Microsofts zu begrüßen waren, sind auch die entsprechenden Funktionen in den Delphi-Versionen 4 – 6 weitgehend unverändert geblieben.

Die Automatik der ActiveX-Komponenten-Erzeugung geht zwar nicht so weit, dass Sie überhaupt nichts mehr zu machen brauchen, aber immerhin schon so weit, dass wir nach sieben Unterkapiteln zu VCL-Komponenten nur noch ein Unterkapitel brauchen, um die Farbpaletten-Komponente aus Kapitel 6.6 zu einem ActiveX-Control zu machen, das über spezielle Entwurfszeit-Funktionen verfügt und das wir in einer Delphi-Anwendung testen werden.

Delphis Automatikfunktionen erlauben es zwar auch, ActiveX-Steuerelemente ohne Kenntnis des COM herzustellen, wenn Sie jedoch ein tieferes Verständnis für die Funktionsweise von ActiveX-Steuerelementen erlangen wollen, sollten Sie sich auch mit den Grundlagen der Interfaces in Kapitel 2.7 und mit der Erzeugung eigener COM-Objekte in Kapitel 8.5 beschäftigen.

Die VCL und ActiveX

Da gerade das Testen eines in Delphi erstellten ActiveX-Controls in einer Delphi-Anwendung erwähnt wurde, sei an dieser Stelle einmal auf die Gründe hingewiesen, warum sich die Erzeugung eines ActiveX-Steuerelements nur dann lohnt, wenn dieses

Element auch in *anderen* Entwicklungsumgebungen eingesetzt werden soll, die nicht über die VCL verfügen, und warum es selbst dann empfehlenswert ist, dieses Control in Delphi weiter als normale VCL-Klasse zu verwenden (und nicht in Form des ActiveX-Controls):

- Zur Entwurfszeit sind VCL-Komponenten besser in die Delphi-IDE integriert und können beispielsweise Property-Editoren zur Verfügung stellen, die flexibler sind als die Property-Einstellungsdialoge von ActiveX-Komponenten.
- Zur Laufzeit einer Anwendung arbeiten VCL-Komponenten effizienter als ActiveX-Komponenten, da sie nicht über eine komplizierte COM-Schnittstelle mit der Anwendung kommunizieren müssen.

Außerdem können Sie nur mit einer VCL-Komponente die Möglichkeiten der Sprache Object Pascal voll ausnutzen:

- VCL-Komponenten sind in eine Klassenhierarchie eingebunden, die als wirklich objektorientiert bezeichnet werden kann. Zwar können Sie auch von ActiveX-Steuerelementen neue Klassen ableiten, Sie können auf diese Weise jedoch keine virtuellen Methoden überschreiben, die von der ActiveX-Komponente definiert werden (überschreiben können Sie lediglich die virtuellen Methoden der von Delphi automatisch erzeugten Hüllkomponente; in die internen Abläufe der ActiveX-Komponente können Sie nur über Properties und Ereignisse eingreifen).
- Der vielleicht wichtigste Vorteil einer ausschließlichen Verwendung von Object Pascal ist, dass Sie keine Einschränkungen bei der Typenauswahl hinnehmen müssen. Einer VCL-Klasse können Sie sogar Object-Pascal-Objekte als Parameter übergeben, während ActiveX-Methoden höchstens *IDispatch*-Interface-Variablen »verstehen«.

ActiveX und das Web

Sowohl die ActiveX-Steuerelemente als auch die in Kapitel 6.8.5 behandelten ActiveForms können Sie auch im Web benutzen. Da diese Steuerelemente für den Web-Einsatz in keiner Weise modifiziert zu werden brauchen – einmal abgesehen davon, dass Sie sie zur Verkleinerung der ausführbaren Datei wahrscheinlich unter Verwendung von Packages kompilieren werden –, geht dieses Kapitel darauf nicht näher ein und verlässt sich auf Delphis Expertenfunktionen bzw. auf die Menüpunkte PROJEKT | DISTRIBUTION ÜBER DAS WEB und OPTIONEN FÜR WEB-DISTRIBUTION sowie auf Literatur zum HTML-Design.

6.8.1 Von der VCL-Hierarchie zu ActiveX

Bevor wir zu den ActiveX-Themen Typenbibliothek, Eigenschaftenseiten und Importieren (Testen) von ActiveX-Komponenten in Delphis Komponentenpalette kommen, erzeugen wir hier im ersten Schritt ein »Experimentalobjekt« für die nächsten Schritte: Die Komponente *TColorPalette* aus Kapitel 6.6 soll in ein ActiveX-Steuerelement umgewandelt werden. Delphi tritt hier mit dem Anspruch an, ein solches Steuerelement automatisch und in nur einem Schritt zu erzeugen, daher brauchen wir in diesem Abschnitt nichts zu programmieren.

Öffnen Sie zuerst mit DATEI | NEU (| WEITERE) den Katalog von Dingen, die Sie unter Delphi neu erzeugen können. Auf der Seite *ActiveX* wählen Sie *ActiveX-Element*. Sie erhalten dann den *ActiveX-Element-Experten*, in dem Sie im ersten Feld *VCL-Klassenname* die zu exportierende Komponente auswählen, in diesem Fall also *TColorPalette*. Die anderen Felder passt der Experte automatisch diesem Namen an, für die Dateien auf der CD-ROM wurden die Namen der Dateien auf maximal acht Zeichen verkürzt. Das ActiveX-Steuerelement heißt, wie vom Experten vorgeschlagen, *ColorPaletteX*; die Implementierungsunit wurde unter dem Namen *CPxImp* gespeichert. Außerdem wurde im Experten noch das Feld *Info-Fenster hinzufügen* markiert und als Threading-Model wurde die Voreinstellung *Apartment* übernommen. Das gesamte Projekt finden Sie auf der CD unter dem Namen `complib\cpx.dpr`.

Eine ActiveX-Bibliothek

Nach der Bestätigung der Eingaben mit *OK* fragt Delphi Sie, ob es ein neues *ActiveX-Bibliotheksprojekt* anlegen soll, falls das aktuelle Projekt noch kein solches ist (ein solches Projekt ist Voraussetzung für die automatische Erstellung einer ActiveX-Komponente). Sie können eine ActiveX-Bibliothek auch durch den gleichnamigen Eintrag auf der *ActiveX*-Seite unter DATEI | NEU (| WEITERE) erzeugen.

Eine ActiveX-Bibliothek wird vom Compiler zu einer DLL übersetzt (wobei diese DLL auch die Dateierweiterung `.ocx` aufweisen darf), daher beginnt der Projektquelltext mit dem Schlüsselwort *library*. Besonderes Kennzeichen einer solchen ActiveX-Bibliothek ist die Einbindung der Unit *ComServ*, die bereits alle Verwaltungsfunktionen für eine ActiveX-Bibliothek enthält, und eine *exports*-Klausel, über die die vier wichtigen Routinen dieser Unit exportiert werden.

Das Produkt des Experten

Nachdem Sie den ActiveX-Element-Experten auf die oben beschriebene Weise auf eine ActiveX-Bibliothek angewendet haben, finden Sie in Ihrem Projekt die folgenden Änderungen vor:

- Über die Option *{$R *.TLB}* im Projektquelltext wurde der Compiler damit beauftragt, die zum Projekt gehörende Typenbibliothek mit in die fertige DLL einzubinden. Natürlich hat der Experte die Typenbibliothek gleich mit erzeugt. Im Dateisystem finden Sie diese Bibliothek unter dem Namen des Projekts mit der Endung `.tlb` (allerdings erst, nachdem das Projekt zum ersten Mal gespeichert wurde) und können sie jederzeit mit DATEI | ÖFFNEN in den Typenbibliotheks-Editor laden.
- Die Typenbibliothek wurde außerdem als Pascal-Unit in die Projektdatei eingebunden; sie trägt einen fest vorgegebenen Namen, der sich aus dem Projektnamen und dem Suffix *_TLB* zusammensetzt (hier also `cpx_tlb.pas`).
- Ebenfalls neu im Projekt sind die beiden Units *CPxImp* und *CPxAbout*. Erstere enthält die Implementation der in der Typenbibliothek erzeugten COM-Schnittstellen, die andere das angeforderte Info-Fenster.
- Über die Option *{$E ocx}* im Projektquelltext wurde der Compiler schließlich angewiesen, der DLL die Endung .OCX zu geben, die per Konvention für ActiveX-Steuerelemente verwendet wird.

Das vorläufige Endergebnis

Damit hat der Experte das Versprechen von Borland eingelöst, denn das so erweiterte Projekt lässt sich ohne einen weiteren Eingriff sofort übersetzen und die daraus entstandene ActiveX-Datei `ColorPaletteX.ocx` kann sofort in andere ActiveX-Container oder in Delphi selbst eingebunden werden. Das Einzige, was unbedingt noch geändert werden sollte, ist der Info-Dialog, der wahrscheinlich noch nicht automatisch den richtigen Autor der Komponente angibt.

Wie Sie die Komponente in Delphis Komponentenpalette einbinden, beschreibt Kapitel 6.8.4, denn zunächst werden wir doch noch einige Verbesserungen an diesem automatisch erzeugten Control vornehmen.

6.8.2 Typenbibliothek und Implementation

Typenbibliotheken werden dazu verwendet, die Außenwelt über die Properties, Methoden und Ereignisse einer Komponente zu informieren. Grundsätzlich darf im Component Object Model jede DLL eine Typenbibliothek haben. Für einen Automationsserver wird sie beispielsweise empfohlen, ist aber nicht Pflicht. Für ActiveX-Steuerelemente ist sie dagegen eine unbedingte Voraussetzung.

Trotzdem ist der Typenbibliotheks-*Editor* (kurz TLB-Editor) für die Erstellung von ActiveX-Steuerelementen nicht so wichtig wie für die Automation, denn die ActiveX-Typenbibliothek sollte ja von Delphi automatisch erstellt werden, während Sie sie für einen Automationsserver selbst definieren müssen – falls Sie für Ihren Server eine Typenbibliothek einrichten wollen. Eine genauere Beschreibung des Editors und der Typenbibliotheken findet daher im Zusammenhang mit der COM-Automation erst in Kapitel 8.7 statt.

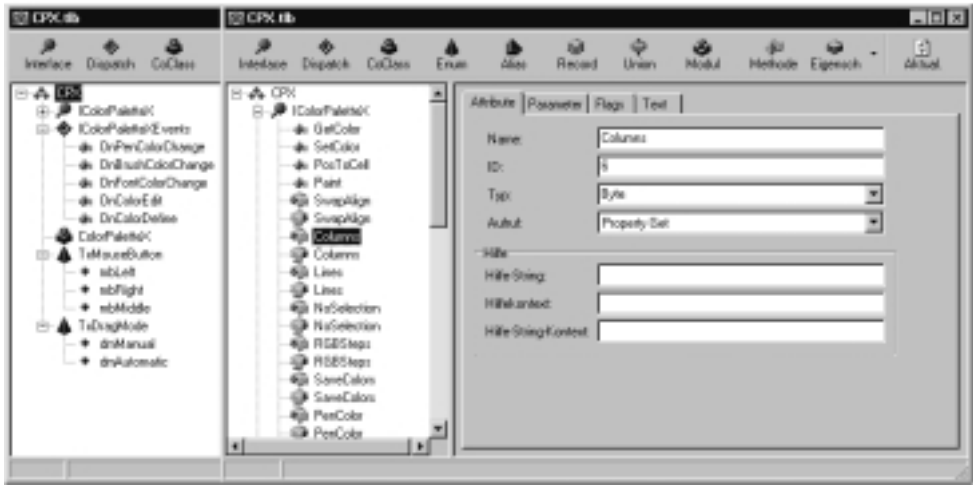


Abbildung 6.9: Die Typenbibliothek von ColorPaletteX.ocx

Die VCL-Komponente in der Typenbibliothek

Wie Sie aus der Baumdarstellung des TLB-Editors in Abbildung 6.9 erkennen können, enthält eine Typenbibliothek für alle wichtigen Arten von Elementen einer VCL-Komponente entsprechende Einträge:

- Auf der ersten Ebene befindet sich zunächst einmal der Eintrag *IColorPaletteX*, der der gesamten Komponente entspricht. *IColorPaletteX* ist eine Schnittstelle nach Definition des Component Object Models. Sie kann von anderen Anwendungen so aufgerufen werden wie die Schnittstelle eines COM-Automationsservers.
- Für Properties und Methoden werden dieser Schnittstelle weitere Einträge untergeordnet, die Sie einzeln auswählen und auf der rechten Seite des Editors editieren können. Seit Delphi 4 wird hier pro Property eine Lese- und eine Schreibmethode dargestellt (in Delphi 3 gab es pro Property nur einen Eintrag).

- Für die Ereignisse legt der Experte eine zweite Schnittstelle an, *IColorPaletteX-Events*, die die einzelnen Ereignisse der Komponente als Unterelemente zugeordnet bekommt (dieses Mal handelt es sich nicht um ein normales Interface, sondern um ein Dispatch-Interface, das durch ein anderes Icon symbolisiert wird).
- Schließlich finden Sie am Ende der Liste auch noch zwei oder drei (je nach Version der VCL) Einträge, die in der Legende des Editorfensters als *Enumeration* bezeichnet werden. In diesen Einträgen werden die Wertebezeichnungen von Aufzählungstypen deklariert, so dass diese auch in anderen Anwendungen verfügbar werden. Im Fall der Farbpalette sind das unter anderem die VCL-Typen *TMouseButton* und *TDragMode*, für welche die Typenbibliotheks-Einträge *TxMouseButton* und *TxDragMode* erzeugt werden.

Für die erfolgreiche Herstellung von ActiveX-Steuerelementen ist es nun weniger wichtig, dass Sie die internen Abläufe von Interfaces und Dispatch-Interfaces verstehen, als dass Sie wissen, dass der Nutzer Ihrer ActiveX-Komponente nur die Elemente Ihrer Komponente nutzen kann, die in der Baumdarstellung des Typenbibliotheks-Editors aufgelistet sind, womit wir an den Grenzen von Delphis Automatik angekommen wären ...

Grenzen der Typenbibliothek

Nicht immer enthält eine automatisch erzeugte Typenbibliothek alle Methoden, Properties oder Ereignisse, die von außen genutzt werden sollen. Für das Fehlen eines Elements kann es verschiedene Gründe geben:

- In der Deklaration des Elements kommt ein Typ vor, der nicht COM-kompatibel ist, also nicht *Byte*, *Currency*, *Real*, *Double*, *LongInt*, *Integer*, *Single*, *SmallInt*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool* und auch kein Interface-Typ ist. Nicht exportiert werden z.B. Methoden, die als Parameter normale Object-Pascal-Objekte erhalten, oder Array-Properties. Eine Übersicht über die erlaubten Typen finden Sie beispielsweise auch, wenn in den Umgebungsoptionen unter TYPBIBLIOTHEK die SPRACHE PASCAL eingestellt ist und Sie im rechten Teil des Typenbibliotheks-Editors eine der Auswahllisten für Typen aufklappen.
- Das Element ist in der Komponente nicht als *public* oder als *published* deklariert. In einem solchen Fall ist es logisch, dass der Experte das Element nicht in die Typenbibliothek übernimmt.
- Delphis Experte konvertiert ein Element aus anderen Gründen nicht in einen Typenbibliothekseintrag.

Falls der dritte Grund zutrifft, wird der Experte möglicherweise in der nächsten Delphi-Version erweitert oder neue Optionen bieten, welche Elemente konvertiert werden sollen. Für die Farbpalette wurden jedenfalls alle erwarteten Properties in das Interface übernommen.

Für die Eigenschaftsseite, die wir in Kapitel 6.8.3 schreiben werden, benötigen wir außer den automatisch übernommenen Properties noch einen Zugriff auf die einzelnen Farben der Farbpalette. Ist die Farbpalette als normale Object-Pascal-Klasse in ein Projekt eingebunden, sind ihre Farben über das *Colors*-Property ansprechbar. Bei diesem Array-Property besteht jedoch keine Aussicht auf eine automatische Konvertierung durch Delphis Experten, weshalb die einzelnen Farben über Methoden ansprechbar gemacht werden müssen. Hierzu wurden einfach die schon vorhandenen, ursprünglich privaten Methoden *GetColor* und *SetColor* in den *public*-Bereich der Klassendeklaration übernommen, damit Delphi sie automatisch mitkonvertiert (zur vollständigen Deklaration von *TColorPalette* siehe auch Kapitel 6.6.2).

Wenn Delphi einmal ein von Ihnen gewünschtes Property oder eine Methode nicht automatisch konvertieren sollte, können Sie die Typenbibliothek von Hand erweitern wie in Kapitel 8.7.1 beschrieben (in diesem Fall müssen Sie eventuell auch noch das Gerüst der Implementierungsunit erweitern, hierzu kommen wir gleich im nächsten Abschnitt).

Hinweis: Normalerweise ist es empfehlenswert, lieber die Schnittstelle der Object-Pascal-Komponentenklasse so zu ändern oder zu erweitern, dass alleine die von Delphi automatisch in die Typenbibliothek konvertierten Properties und Methoden ausreichend sind. Eine Veränderung der Typenbibliothek eines ActiveX-Steuerelements ist deshalb nicht so vorteilhaft, weil jede Änderung durch eine eventuelle Neuerzeugung der Bibliothek bei einem erneuten Aufruf des Experten überschrieben werden würde.

Die Implementation

Spätestens wenn Sie die Typenbibliothek wie oben beschrieben erweitern, ist es an der Zeit, sich um die Implementations-Unit zu kümmern, die der Experte ebenfalls automatisch erzeugt. Dieser Vorgang stimmt prinzipiell mit der Implementation einer der Automationsschnittstellen aus Kapitel 8.7 überein und braucht daher hier nicht im Detail beschrieben zu werden.

Interessant ist an dieser Stelle vor allem, wie Delphi eine solche Implementation wieder *automatisch* erzeugt. Das ActiveX-Element, mit dem die anderen Anwendungen kommunizieren, ist nämlich gar nicht Ihre Komponente, sondern eine von Delphis Experten automatisch erzeugte Co-Klasse, die die Schnittstelle aus der Typenbibliothek unterstützt. Ihre Grundfunktionalität erbt diese Co-Klasse von der VCL-Klasse

TActiveXControl, die eine der grundlegenden COM-Klassen neben *TInterfacedObject* darstellt. Für die Farbpalette sieht ihre Deklaration wie folgt aus:

```
type
  TColorPaletteX = class(TActiveXControl, IColorPaletteX)
  private
    { Private-Deklarationen }
    FDelphiControl: TColorPalette;
```

Wichtigstes Element dieser Klasse ist die Variable *FDelphiControl*, deren Typ offenbar die Klasse der Komponente ist, die als ActiveX-Element exportiert werden soll. Die automatisch erzeugte Co-Klasse funktioniert also als Vermittlerin zwischen der Außenwelt und Ihrer VCL-Komponente. Diese Vermittlungsfunktion ist an den automatisch erzeugten Methoden der Klasse deutlich erkennbar:

```
function TColorPaletteX.Get_FontColor: OLE_COLOR;
begin
  Result := OLE_COLOR(FDelphiControl.FontColor);
end;

function TColorPaletteX.Get_SwapAlign: WordBool;
begin
  Result := FDelphiControl.SwapAlign;
end;

procedure TColorPaletteX.Set_DragMode(Value: TxDragMode);
begin
  FDelphiControl.DragMode := TDragMode(Value);
end;
```

Jeder Zugriff auf eines der Properties (im Beispiel die Properties *FontColor*, *SwapAlign* und *DragMode*) über die automatisch erzeugten Schreib- und Lesemethoden wird direkt an die VCL-Komponente weitergegeben.

Der obige Codeausschnitt ist nicht nur zum Verständnis der internen Vorgänge einer Delphi-ActiveX-Komponente wichtig, sie ist auch ein anschauliches Muster für die Erweiterungen, die Sie an einem solchen automatisch erzeugten Gerüst vornehmen können – und vielleicht müssen, wenn Sie alle Fähigkeiten Ihrer Komponente nach außen hin zugänglich machen wollen, denn dieses Gerüst hat manchmal Lücken.

Sollten Sie darin beispielsweise eine leere Methode wie die folgende finden ...

```
procedure TColorPaletteX.SetColor(i: Byte; val: OLE_COLOR);
begin

end;
```

... müssen Sie diese durch einen Aufruf der entsprechenden Methode der Komponente ergänzen ...

```
FDelphiControl.SetColor(i, val);
```

Wenn Sie die Methode von einem ActiveX-Container gar nicht aufrufen wollen, können Sie sie natürlich auch leer lassen (sicherer wäre es allerdings, sie dann gleich aus der Liste des Typbibliothekseditors zu entfernen, so dass sie gar nicht mehr von außen aufgerufen werden kann).

Manueller Methoden-Export

Wenn Sie die von Delphi erzeugte ActiveX-Klasse (hier *TColorPaletteX*) um weitere Methoden erweitern wollen[16], können Sie so vorgehen wie beim Automatisieren von Methoden bei der COM-Automation: Im Beispiel der *SetColor*-Methode würden Sie etwa im TLB-Editor unter *IColorPaletteX* den Eintrag

```
SetColor(i: Byte; val: OLE_COLOR);
```

hinzufügen, die Implementierungsunit aktualisieren (über den Schalter im TLB-Editor) und dann in dieser Unit den Rumpf der Methode wie bereits oben gezeigt definieren. Eine ausführliche Erläuterung dieser Vorgehensweise finden Sie in Kapitel 8.7.1 für die Implementierung der COM-Automationsobjekte.

6.8.3 Eine Eigenschaftenseite

Eigenschaftenseiten von ActiveX-Elementen entsprechen den Komponenten- und Property-Editoren der Delphi-IDE, sind allerdings nicht so flexibel wie diese. In Delphi besteht eine Eigenschaftenseite zur Entwurfszeit aus einem normalen Formular, das jedoch nicht von *TForm*, sondern von *TPropertyPage* abgeleitet wird. Auf diesem Formular bringen Sie Komponenten unter, mit dem der Verwender der Komponente in seiner Entwicklungsumgebung, die ja auch eine andere als Delphi sein kann, auf verschiedene Eigenschaften der Komponente zugreifen kann.

Da Sie in Umgebungen wie in Delphi bereits die meisten Eigenschaften über so etwas Ähnliches wie einen Objektinspektor einstellen können, ist eine solche Eigenschaftsseite nur für besondere Eigenschaften oder Funktionen wirklich notwendig. Im Farbpalettenbeispiel aus Kapitel 6.6 konnte beispielsweise ein spezieller Dialog zum Editieren der einzelnen Farben und zum Festlegen von Farbverläufen verwendet werden. Diesen Dialog werden wir im vorliegenden Kapitel zu einer Eigenschaftsseite machen.

16 Eventuell übertragen nicht alle älteren Delphi-Versionen automatisch alle Methoden der »VCL-Komponente« in das ActiveX-Komponenten-Gerüst, so dass sich allein hieraus die Notwendigkeit eines manuellen Exports ergeben kann.

Entwurf des Formulars

Auf einem weiteren Ausflug zur *ActiveX*-Seite des DATEI | NEU (| WEITERE)-Dialogs wählen wir nun das Icon *Eigenschaftenseite* aus. Hier gibt es ausnahmsweise keinen Expertendialog, sondern es erscheint sofort ein neues Formular (vorausgesetzt, dass Sie bereits ein ActiveX-Bibliotheks-Projekt geöffnet hatten).

Für die Farbpalette muss dieses Formular nicht so viele Steuerelemente enthalten wie der Property-Editor aus Kapitel 6.6.5, denn eine Eigenschaftsseite wird in einen Dialog eingebunden, der bereits die wichtigen Schalter *OK*, *Abbrechen* und *Übernehmen* enthält (siehe Abbildung 6.10). Die Eigenschaftsseite der Farbpalette enthält demnach auch nur ein mit *alClient* ausgerichtetes *TColorPalette*-Exemplar.

Wichtig für den Entwurf des Formulars ist auch seine Größe, denn diese beeinflusst später die Größe des gesamten Eigenschaften-Dialogs.

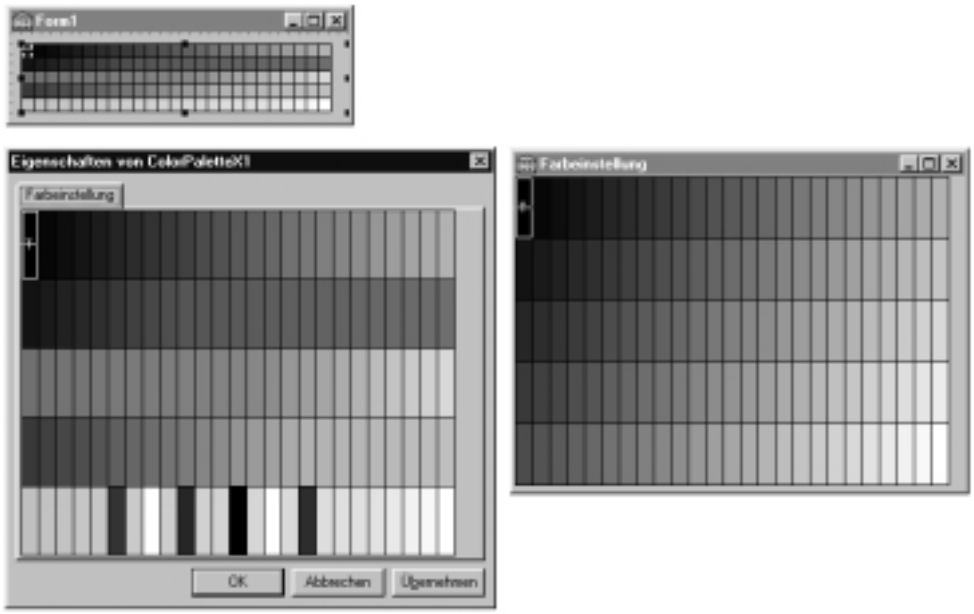


Abbildung 6.10: Ein ColorPaletteX-Control (oben) mit geöffneter Eigenschaftsseite, rechts das Formular der Eigenschaftsseite beim Entwurf in der Delphi-IDE

Der Code der Eigenschaftsseite

Der Code der Eigenschaftsseite wurde zum größten Teil einfach aus dem Property-Editor kopiert, den wir in Kapitel 6.6.5 entworfen haben, so z.B. die Routine *ColorRange* für die Berechnung eines Farbverlaufs und die Methoden für die Ereignisse *OnMouseDown*, *OnDragOver* und *OnDragDrop*. Hierbei funktioniert es sogar, die Farbpalette des

Property-Editors (Unit *ColorPalEdit*) zu einer Komponentenschablone zu machen und diese in das Formular der Eigenschaftsseite einzufügen.

Neben der Bedienungsoberfläche der Seite ist es besonders wichtig, dass die Seite auch die Werte der Komponenten laden kann, zu deren Editieren sie später einmal geöffnet wird. Sie muss also, ähnlich wie der in Kapitel 6.6.5 beschriebene Property-Editor, beispielsweise die Farbwerte aus der zu editierenden Komponente auslesen und in ihrer eigenen Farbpalette darstellen.

Auch hier wurde der Code für die Eigenschaftsseite aus dem bestehenden Code durch anfängliches Kopieren »abgeleitet« . In einer Eigenschaftsseite verteilt er sich auf zwei Methoden, deren Rümpfe Delphi automatisch erzeugt hat:

```
procedure UpdatePropertyPage; override;
procedure UpdateObject; override;
```

Beim Öffnen des Eigenschaftsdialogs wird die virtuelle Methode *UpdatePropertyPage* aufgerufen. Ihre Aufgabe ist es, die aktuellen Einstellungen der Komponente einzulesen und in den eigenen Steuerelementen darzustellen.

Beim Schließen des Dialogs und beim Drücken des Schalters *Übernehmen* wird die Methode *UpdateObject* aufgerufen, in der die Eigenschaftsseite die Werte zurück in die Komponente schreibt. Die »Komponente« liegt jedoch hier nicht in Form einer VCL-Komponente vor, sondern in Form der Variablen *OleObject*. Dies ist eine Variante, wie sie bei einer bestimmten Form der COM-Automation eingesetzt wird.

Als Beispiel sei hier die *UpdateObject*-Methode der Farbpaletten-Eigenschaftsseite abgedruckt:

```
procedure TPropertyPage1.UpdateObject;
var
  i: Integer;
begin
  { Update OleObject from your controls }
  with Palette do begin
    for i:=0 to Columns*Lines-1 do
      Oleobject.SetColor(i, GetColorInt(i));
    if Palette.SaveColors then
      OleObject.SaveColors:=True;
  end;
end;
```

Der Übernehmen-Schalter

Als weiteren wichtigen Punkt gilt es noch, die Funktion des bereits angesprochenen Übernehmen-Schalters sicherzustellen. Dieser ist nämlich standardmäßig grau dargestellt und wird nur dann aktiviert, wenn Ihre Seite dem Eigenschaftsdialog mitteilt, dass es überhaupt Änderungen gibt, die übernommen werden können. Diese Mittei-

lung ist besonders einfach, denn sie besteht lediglich in einem Aufruf der VCL-Methode *Modified*. Die Eigenschaftsseite der Farbpalette meldet sich in zwei Fällen als »modifiziert«: Wenn der Benutzer einen Farbverlauf definiert hat und wenn er eine einzelne Farbe per Dialog verändert hat. Im letzten Fall erzeugt die in der Eigenschaftsseite enthaltene Farbpalettenkomponente ein *OnEdit*-Ereignis, das im Seitenformular wie folgt bearbeitet werden kann:

```
procedure TPropertyPage1.PaletteColorEdit(Sender: TObject; Index: Integer;
  NewColor: TColor);
begin
  Modified;
end;
```

Registrieren der Seite

Als Letztes muss die Seite noch aktiviert werden. Hierzu suchen Sie noch einmal die Implementierungsunit zur automatisch erzeugten Co-Klasse auf und tragen die Seite(n) in die bereits mit einem Kommentar-Beispiel aufwartende Methode *DefinePropertyPages* ein:

```
procedure TColorPaletteX.DefinePropertyPages(
                            DefinePropertyPage: TDefinePropertyPage);
begin
  DefinePropertyPage(Class_PropertyPage1);
end;
```

Dabei ist *Class_PropertyPage1* die *TGUID*-Konstante, die der Seitenexperte kurz vor den Beginn des Implementationsteils der Formular-Unit der Seite geschrieben hat.

6.8.4 Testen der ActiveX-Komponente in Delphi

Wie in der Einleitung zu Kapitel 6.8 ausführlich erläutert wurde, empfiehlt es sich, Komponenten, die in Delphi entwickelt wurden, unter Delphi auch als VCL-Klassen und nicht als ActiveX-Controls zu verwenden. So nutzt der TreeDesigner aus Kapitel 5 die Farbpalette auch in der Urfassung, in der sie in Kapitel 6.6 entwickelt wurde.

Eine Einbindung eigener ActiveX-Komponenten in eine Delphi-Anwendung bietet sich jedoch erstens zum Test der Komponente an und zweitens zum Lernen, wie fremde ActiveX-Komponenten in Delphi genutzt werden können (äußerlich gibt es nämlich zwischen der OCX-Farbpalette und einer in einem völlig anderen Entwicklungssystem erstellten ActiveX-Komponente keinen Unterschied).

Einbinden eines ActiveX-Controls

Das Einbinden von ActiveX-Elementen läuft wie das Exportieren von VCL-Komponenten dialoggesteuert ab und führt wie dieses ebenfalls zu automatisch erzeugten Dateien. Auch ActiveX-Steuerelemente können über Packages installiert werden. Da die ActiveX-Version der Farbpalette nur zur Demonstration gedacht ist, ist sie nicht im Package der anderen Buch-Komponenten enthalten und muss getrennt installiert werden:

- Wählen Sie zuerst Komponente | ActiveX importieren.
- Falls es sich um ein Element handelt, das noch nicht in der Registry eingetragen ist, drücken Sie *Hinzufügen* und geben die (OCX-)Datei an, in der sich das Element befindet. Delphi holt die Registrierung dann nach und fügt das Element in die Liste der registrierten Steuerelemente innerhalb des Dialogs ein.
- Markieren Sie ein Element in dieser Liste und drücken Sie den Schalter *Installieren*.

An dieser Stelle sind Sie nun ungefähr an der Position angelangt, die Sie auch mit KOMPONENTE | KOMPONENTE INSTALLIEREN... erreichen: bei der Auswahl eines Packages, denn auch ActiveX-Komponenten werden in Delphi über Packages in der Komponentenpalette installiert. Allerdings ist es nicht die ActiveX-Komponente selbst (die OCX-Datei), die in das Package aufgenommen wird, sondern eine automatisch erzeugte Unit, die zwischen OCX und VCL vermittelt.

Nach der Installation wird die Komponente normalerweise auf der Seite *ActiveX* der Komponentenpalette eingefügt und steht nun grob gesehen wie eine normale Komponente zur Verfügung: Sie können die Komponente in das Formular einzeichnen, Eigenschaften im Objektinspektor editieren, per Doppelklick auf die Komponente einen Komponenten-Editor aufrufen (der bei ActiveX-Controls als Eigenschaftendialog bezeichnet wird) und die OCX-Datei wie eine normale DLL mit Ihrer Anwendung weitergeben. Genauer betrachtet gelten natürlich die schon angesprochenen Unterschiede zu einer echten VCL-Komponente.

Um eine ActiveX-Komponente nach einer Änderung ihrer Implementierung in der Komponentenpalette zu aktualisieren, können Sie grundsätzlich so vorgehen wie beim Aktualisieren von Komponenten und den Package-Editor sowie dessen Aktionsschalter verwenden. Bei ActiveX-Elementen stellt sich jedoch das Problem, dass Delphi sie nicht neu kompilieren kann, solange die ausführbare OCX-Datei sich in Benutzung befindet und daher gesperrt ist. Vor der Neukompilierung müssen Sie ein solches Steuerelement daher temporär aus der Komponentenpalette entfernen.

Ein Blick hinter die Kulissen

Zum Abschluss des Kapitels werfen wir noch einen Blick hinter die Kulissen, denn dort hat sich soeben bei der Installation der ActiveX-Komponente in der Delphi IDE ein interessanter Kreis geschlossen, und zwar bei der Erzeugung der von Delphi automatisch erzeugten »Hüll-Komponente«, die statt der eigentlichen OCX-Datei in der Komponentenpalette installiert wurde:

Da Delphi die Unit dieser Komponente auf Grundlage der Typenbibliothek erzeugt (eine bessere Möglichkeit gibt es ja nicht, um an verlässliche Informationen über ein ActiveX-Control zu gelangen), ist diese Unit völlig identisch mit der Unit, die Delphi schon bei der Erstellung der ActiveX-Komponente aus der Typenbibliothek erzeugt hat, in diesem Beispiel also der Unit *CPX_TLB.pas*. Diese Unit enthält nämlich nicht nur die im Typenbibliotheks-Editor sichtbaren Interface-Deklarationen, sondern auch noch eine Klasse, die in diesem Beispiel *TColorPaletteX* heißt. Eine solche Klasse kennen wir schon aus der Implementierungsunit *CPxImp*. Im Überblick haben wir also zwischen den folgenden Schnittstellen und Klassen zu unterscheiden:

- die ursprüngliche VCL-Komponente *TColorPalette*;
- die COM-Schnittstelle für die zugehörige ActiveX-Komponente *IColorPaletteX* und die ActiveX-Komponente selbst, *ColorPaletteX*;
- die Implementation der COM-Schnittstelle in der Unit *CPxImp* durch die Klasse *TColorPaletteX*;
- die Klasse *TColorPaletteX*, die als Hülle um die *ColorPaletteX*-Komponente erzeugt wird.

Es gibt also zwei verschiedene Deklarationen von *TColorPaletteX*:

```
TColorPaletteX = class(TActiveXControl, IColorPaletteX)
  ... und ...
TColorPaletteX = class(TOleControl)
```

Die erste davon wird in der ActiveX-Bibliothek verwendet, die in Kapitel 6.8.1 erzeugt wurde; die zweite stellt das dar, was im ActiveX-Container noch von der ersten *TColorPaletteX* bzw. von der Ursprungskomponente *TColorPalette* übrig geblieben ist.

6.8.5 ActiveForms

ActiveForms waren eine der spektakulärsten Neuerungen von Delphi 3. Intern handelt es sich bei einer ActiveForm um eine Ableitung der Klasse *TActiveForm*, die Sie in Delphis Formulareditor wie ein ganz normales Formular editieren können; nach außen hin ist eine ActiveForm ein ActiveX-Steuerelement. Die Besonderheit dieses Elements ist, dass es aus vielen weiteren Steuerelementen bestehen kann.

Als Einsatzziel für eine ActiveForm wird oft nur genannt, das Formular als aktives Dokument auf einer HTML-Seite einzusetzen, jedoch können Sie ein solches Formular auch wie andere ActiveX-Steuerelemente verwenden und sogar in Delphis Komponentenpalette einbinden. Dort hat es schon gewisse Ähnlichkeiten mit einer Komponentenschablone, allerdings können Sie nicht mehr auf die einzelnen Komponenten, geschweige denn auf deren Properties zugreifen (es sei denn, Sie erweitern die Schnittstelle der ActiveForm manuell). Die Komponenten sind damit sozusagen mit dem Formular zu einer untrennbaren Einheit »festgeschweißt«, bei der Sie nur noch die Umrandung dieser Einheit beeinflussen können (Property *AxBorderStyle*).

Da die »normalen« ActiveX-Steuerelemente in Delphi intern genauso funktionieren wie ActiveForms, können wir uns in diesem Kapitel ganz auf diese »normalen« ActiveX-Steuerelemente konzentrieren. Wie am Beispiel der Farbpaletten-Komponente können Sie auch bei ActiveForms die automatisch erzeugte Typenbibliothek und die Implementierungsunit erweitern sowie Eigenschaftsseiten entwerfen (wobei Letzteres dringend erforderlich ist, um die starre Oberfläche einer solchen Komponente »aufzuweichen«). Augenfälligster Unterschied beim Erstellen einer ActiveForm ist, dass Sie im DATEI | NEU (| WEITERE)-Dialog statt *ActiveX-Element* das Symbol *Active Form* auswählen.

Hinweis: Um Eigenschaftenseiten für eine ActiveForm zu registrieren, müssen Sie die *DefinePropertyPages*-Methode Ihrer Formularklasse selbst überschreiben, da Delphi nicht wie bei ActiveX-Elementen automatisch einen Rumpf für diese Methode schreibt. Diese Methode ist zwar für *TActiveForm* nicht in der Hilfedatei dokumentiert, aber trotzdem vorhanden.

7 Datenbankanwendungen

Dieses Kapitel befasst sich mit der Entwicklung von Datenbankanwendungen und verwendet dabei als alternative Zugriffsmethoden sowohl die BDE als auch dbExpress. Die Beispielprogramme benutzen drei verschiedene Arten von Datenbanken: dBase / Paradox (nur über BDE), Interbase (BDE und dbExpress) sowie MyBase (ohne BDE). Während die BDE-Komponenten bereits aus allen früheren Delphi-Versionen bekannt sind, hat Borland dbExpress mit Kylix und Delphi 6 anlässlich der plattformübergreifenden Programmentwicklung neu eingeführt.

- Kapitel 7.1 zeigt, wie Sie mit den Bordmitteln von Delphi Tabellen und Datenbanken im dBase / Paradox-, im Interbase- und im MyBase-Format erzeugen, mit welchen Komponenten Sie aus Delphi-Anwendungen auf diese Datenbanken zugreifen und wie bestehende Anwendungen auf die neue dbExpress-Architektur umgestellt werden können.
- Kapitel 7.2 steigt dann richtig in die Delphi-IDE ein und erläutert den grundlegenden Aufbau eines Datenbank-Formulars sowie die automatischen Abläufe innerhalb einer Datenbankanwendung. Als Beispielprogramme werden zwei Browser-Anwendungen vorgestellt, mit denen Sie lokale Datenbanken und Interbase-Datenbanken untersuchen können.
- Kapitel 7.3 befasst sich im Detail mit Feldern und den dafür von Delphi bereitgestellten Komponenten; zu den Themen gehören beispielsweise das programmgesteuerte Untersuchen der Spaltentypen und das Setzen der Feldwerte in einem Record. Als neue Beispielanwendung verwendet dieses Kapitel eine Dateidatenbank, die ganz programmgesteuert aufgebaut wird; der Benutzer muss nur das einzulesende Verzeichnis angeben.
- In Kapitel 7.4 wird die Dateidatenbank durch Verwendung von Indizes, die das Hauptthema dieses Kapitels sind, erweitert und vervollständigt.
- Ein praxisnäheres Beispiel für eine Datenbankanwendung mit mehreren Formularen und Datenmengen, in denen auch Benutzereingaben verarbeitet werden, finden Sie in Kapitel 7.5. Anhand einer Terminverwaltungs-Anwendung wird dort die Arbeit mit SQL-Abfragen und Datenbank-Updates demonstriert.

Einige der Beispielprogramme dieses Kapitels laufen übrigens auch noch mit der Standard-Version von Delphi 4, wo Datenbankunterstützung noch zu dieser kleinsten Ausgabe dazugehörte, während für alle Beispielprogramme unter Delphi 6 mindestens die Professional-Ausgabe erforderlich ist (teilweise finden Sie auf der CD auch spezielle Programmversionen mit kleineren Einschränkungen für die Delphi-Generationen noch vor Delphi 4).

Komponenten für BDE und dbExpress

Um Ihnen eine Unterscheidung der in diesem Kapitel teilweise parallel behandelten Datenzugriffsmethoden BDE und dbExpress zu erleichtern, fasst die folgende Tabelle die grundlegenden Komponenten der beiden Zugriffsmethoden sowie die Komponenten, die unabhängig von der verwendeten Zugriffsmethode sind, vorab zusammen:

	BDE-spezifisch	**dbExpress-spezifisch**	**allgemein**
Verbindungs-komponenten	TDatabase	TSQLConnection	–
Datenmengen-Komponenten	TTable, TQuery, TBDE-ClientDataSet	TSQLClientDataSet (unidirektional: TSQLTable, TSQLQuery)	TClientDataSet (einzeln verwendet kann diese Komponente nur auf MyBase-Dateien zugreifen)
weitere in diesem Kapitel verwendete Komponenten	TUpdateSQL (kurz erwähnt in Kapitel 7.5.6)	TSQLMonitor (Protokollfunktion für SQL-Anweisungen)	TDataSource (benötigt in allen Beispielprogrammen)
Palettenseite in D6	BDE	DBEXPRESS	DATENZUGRIFF
Palettenseite in D4/5	DATENZUGRIFF	–	MIDAS/DATENZUGRIFF

Die in Kapitel 7.2.2 genannten datensensitiven Steuerelemente und die in Kapitel 7.3 behandelten Feldkomponenten stehen unabhängig von der verwendeten Datenzugriffsmethode allen Datenbankanwendungen zur Verfügung.

7.1 Datenbank-Variationen und Datenzugriff

Auch wenn in diesem Buch sehr wenig Platz für die Erläuterung von allgemeinen Datenbankgrundlagen ist, können an dieser Stelle grundlegende Begriffe der Datenbankprogrammierung zusammengefasst werden.

Grundbegriffe und Klassen

Als *Datenbanken* sollen im Folgenden Sammlungen von einer oder mehreren *Tabellen* gelten. Jede Tabelle hat eine genau festgelegte Struktur: Sie besteht aus einer oder mehreren Spalten, die jeweils einen Namen und einen Datentyp besitzen, und beliebig vielen Datensätzen, den Records. Die einzelnen Spalten werden auch als *Felder* bezeichnet. Die Tabellen lassen sich als Spezialfall des abstrakten Konzepts der *Datenmengen* auffassen. Eine Datenmenge hat den gleichen Aufbau wie eine Tabelle, muss aber nicht Teil einer Datenbank sein, sondern kann auch dynamisch aufgebaut werden, z.B. als Ergebnis einer *Abfrage*, die z.B. lauten könnte: »Liste alle Datensätze auf, in denen das Feld Energiebedarf kleiner als 2 ist.«

Das wichtigste Verknüpfungsmerkmal von Tabellen in relationalen Datenbanken ist die *Relation* zwischen zwei Tabellen. Eine Relation verknüpft die Einträge aus einer Tabelle mit Zusatzinformationen aus einer anderen Tabelle. Dabei dient jeweils ein Feld als Schlüssel, der in beiden Tabellen übereinstimmen muss, um die »Gesamtinformation« zu ergeben. Zwei Beispiele dazu:

- Ein Beispiel, das wir später auch in einem Beispielprogramm verwenden werden: In der Dateidatenbank aus Kapitel 7.3.6 werden in einer Tabelle die Namen aller Dateien aufgeführt, die in einem bestimmten Verzeichnisbaum enthalten sind. Dabei soll neben dem Dateinamen auch der komplette Pfad abgespeichert werden. Nun wäre es jedoch nicht besonders platzsparend, wenn etwa für 100 Dateien eines Verzeichnisses 100-mal der komplette und immer gleiche Verzeichnispfad abgespeichert werden müsste. Daher speichert die Dateidatenbank die Verzeichnispfade in einer eigenen Tabelle, versieht sie mit Nummern und legt in der Dateitabelle nur noch die Nummer des Pfades ab anstatt den kompletten String. Die Verknüpfung der beiden Tabellen geschieht also über das Feld »Verzeichnisnummer«, das in beiden Tabellen enthalten sein muss.
- Das Standardbeispiel für Relationen aus Geschäftsanwendungen ist wohl das einer einfachen Kundendatenbank, in der die Adressen der Kunden in einer Tabelle aufbewahrt und mit den berühmten Kundennummern versehen werden. Zu vielen Kunden gibt es nun in einer anderen Tabelle eine oder mehrere Bestellungen. Wenn die Größe einer Bestellung nicht von vornherein abschätzbar ist, wird normalerweise noch eine Tabelle für die einzelnen Bestellpositionen angelegt. Diese Tabelle enthält alle Bestellpositionen aller Bestellungen aller Kunden, was auf den ersten Blick etwas unübersichtlich erscheinen mag. Doch über den Schlüssel der Bestellnummer können genau die Positionen gefunden werden, die zu einer bestimmten Bestellung eines bestimmten Kunden gehören.

Allgemein empfiehlt sich die Bildung einer Relation immer dann, wenn zu einer Menge von Informationen mehrere gleich aufgebaute Mengen von Zusatzinformationen gehören (mehrere Dateien zu einem Verzeichnis, mehrere Bestellungen pro Kunde,

mehrere Bestellpositionen pro Bestellung). Dieses Grundprinzip lässt sich auch in formalere Regeln fassen und wird in der Datenbank-Theorie bei der Definition der so genannten *Normalformen* verwendet, die jedoch in diesem Kaptiel nicht weiter erörtert werden können.

In der VCL finden Sie einige der genannten Konzepte durch grundlegende Klassen repräsentiert: *TDataSet* und davon abgeleitete *DataSet*-Klassen stehen für Datenmengen, die Objekte der *TField*-Klassen für einzelne Felder. Für die Datenbanken selbst gibt es keine Klasse, da diese von externen Datenbanksystemen oder Treibern verwaltet werden. Auf der Seite von Delphi treten Datenbanken statt dessen durch *Connection*-Objekte in Erscheinung, die eine von *TCustomConnection* abgeleitete Klasse haben. Sie sorgen für eine Verbindung zum externen Datenbank-Server. (Die explizite Verwendung solcher Verbindungsobjekte ist optional: Wenn Sie keine dieser Komponenten in Ihre Anwendung einfügen, wird evtl. automatisch eine solche Komponente erzeugt.) Die Relationen zwischen Tabellen begegnen uns in den Komponenten in den Properties *MasterSource* und *MasterFields* der Datenmengen-Komponenten wieder (siehe hierzu Kapitel 7.4.5).

Physikalische Struktur

Was die physikalische Struktur der Datenbank betrifft, so gibt es zwei grundlegende Arten, eine Datenbank auf Datenträger zu speichern. Die einfachere Art besteht darin, für jede Tabelle eine eigene Datei anzulegen. Dies können Sie in Delphi mit den Komponenten *TClientDataSet* und *TSQLClientDataSet* erreichen. Beide können sowohl binäre als auch XML-Dateien erzeugen (siehe Kapitel 7.1.2). Auch Desktop-Datenbanken im Paradox- und dBase-Format werden in der Form »eine Tabelle – eine Datei« abgelegt (siehe Kapitel 7.1.1).

Die andere Organisationsform besteht darin, dass die gesamte Datenbank in einer einzigen Datei abgelegt wird. Dies ist bei den von Interbase angelegten Datenbanken der Fall, die üblicherweise in Dateien mit der Endung `.gdb` gespeichert werden. Wie Sie eine solche Datenbank erzeugen können, erfahren Sie in den Kapiteln 7.1.3 und 7.1.5.

7.1.1 BDE-spezifische Desktop-Datenbanken

Während es für die meisten Datenbanken alternative Zugriffswege neben der BDE gibt (so können Sie Interbase-Datenbanken beispielsweise über die BDE, über spezielle, in diesem Buch nicht weiter behandelte Interbase-Express-Komponenten oder über dbExpress ansteuern), gibt es im Lieferumfang von Delphi keine Alternative zur BDE, wenn es um die Benutzung von Datenbanken im altbewährten dBase- oder Paradox-Format geht. Diese Datenbanken heißen Desktop-Datenbanken, weil ihre Daten immer auf dem PC gespeichert sind, auf dem auch die Anwendung läuft.

Tabellen

Dabei verteilen sich die Daten einer einzigen Datenbank auf mehrere Dateien (pro Tabelle werden z.B. eine Datendatei und ggf. eine oder mehrere Indexdateien angelegt). Die Dateinamen sind dabei auch gleichzeitig Tabellennamen, beispielsweise `files.db` (mit Paradox-Dateiendung) oder `customer.dbf` (dBase).

Um diese Dateien zu einer »Datenbank« zusammenzufassen, dient die Verzeichnisstruktur des Dateissystems: Das Verzeichnis selbst zählt als Datenbank. Für den Fall, dass es zu (wartungs)aufwändig ist, wenn für jeden Zugriff auf diese Datenbank immer der genaue Verzeichnispfad angeben werden muss, gibt es das Konzept der Aliase. Der Verzeichnispfad dabei durch einen Alias-Namen ersetzt.

Alias-Namen

Unter einem Alias-Namen können Sie in der BDE-Verwaltung eine beliebige Datenbank angeben, also sowohl das lokale Verzeichnis einer Desktop-Datenbank als auch eine externe Server-Datenbank. Die derzeit bei der BDE registrierten globalen Alias-Namen können Sie im BDE-Konfigurationsprogramm einsehen (siehe Delphi-Programmordner oder den Eintrag *BDE-Verwaltung* in der Systemsteuerung) oder im Datenbank-Explorer (Delphi-IDE: DATENBANK | EXPLORER).

Das Installationsprogramm von Delphi legt bereits mindestens einen solchen Alias-Namen (*DBDEMOS*) an, der das Verzeichnis angibt, in dem sich Delphis Beispieldatenbanken befinden. Wie Sie weitere globale Aliase vergeben, können Sie Online im BDE-Konfigurationsprogramm erfahren. Neben lokalen Alias-Namen können Sie mit Hilfe der Komponente *TDataBase* und deren Property *DatabaseName* auch lokale Aliase innerhalb einer Delphi-Anwendung definieren[17].

Datenzugriffskomponenten *R177*

Um eine Verbindung zu einer dBase- oder Paradox-Datenbank aufzunehmen, benötigen Sie den Namen der Datenbank (Alias-Name oder Verzeichnispfad) und die Namen der Tabellen. Die für die Verbindung notwendigen Komponenten finden Sie in Delphi 6 auf der Palettenseite *BDE*, bis Delphi 5 auf der Seite *Datenzugriff*. Im einfachsten Fall verwenden Sie für jede Tabelle der Datenbank eine *TTable*-Komponente, in deren *TableName*-Property Sie den Namen der Tabelle angeben. Damit Sie hierbei im Objektinspektor bequem aus den vorhandenen Tabellen wählen können, muss zuvor der Datenbankname eingestellt worden sein. Hierfür gibt es zwei Möglichkeiten:

17 *TDataBase* ist eine der oben erwähnten *TCustomConnection*-Komponenten, und zwar die speziell für den BDE-Zugriff zuständige, jedoch optionale Verbindungskomponente.

- Sie können ihn für jede Datenmengenkomponente (etwa *TTable* oder *TQuery*) separat im Property *DatabaseName* angeben. Sollte sich der Datenbankname ändern, müssen Sie auch die *DatabaseName*-Properties aller verwendeten Datenmengen-Komponenten ändern.
- Die zweite Möglichkeit funktioniert nur unter Verwendung eines Alias-Namens. Sie benötigen dann zusätzlich für die Datenmengen-Komponenten eine *TDatabase*-Komponente, in deren *AliasName*-Property Sie den Alias-Namen angeben. Im Property *DatabaseName* geben Sie der *Database*-Komponente einen *lokalen* Aliasnamen, der nur innerhalb der Delphi-Anwendung gültig ist. In den *DatabaseName*-Properties der *TTable*- oder *TQuery*-Komponenten geben Sie dann statt des globalen BDE-Aliases nur diesen lokalen Aliasnamen an. Wenn sich der (globale) Datenbankname ändert, müssen Sie dann nur noch den Aliasnamen einer einzigen *Database*-Komponente ändern.

Hinweis: *TDataBase* wird nicht nur zur Kapselung der Verbindungen benötigt, sondern auch für die Steuerung der Transaktionen bei einer Server-SQL-Datenbank. Eine gute Transaktionskontrolle stellt beispielsweise sicher, dass die Daten der Datenbank in verschiedenen Sonder- und Härtefällen konsistent bleiben, dass also etwa ein gleichzeitiger Datenzugriff zweier Benutzer nicht zu einer Vermischung der Daten und ein Stromausfall nicht zu einer nur halb ausgeführten Operation führen kann.

Die Datenbankoberfläche

Die Datenbankoberfläche ist ein leistungsfähiges Hilfsmittel zur Verwaltung von dBase- und Paradox-Tabellen. Sie eignet sich auch quasi als neutraler Schiedsrichter zum Überprüfen der Veränderungen, die eine Delphi-Datenbankanwendung an einer Tabelle vorgenommen hat. Wenn Ihre Delphi-Anwendung beispielsweise eine unerwartete Fehlermeldung ausgibt, können Sie die Tabelle, mit der dies stattfand, in der Datenbankoberfläche untersuchen. Diese erlaubt einen sichereren Einblick in die Tabelle als eine in der Entwicklung befindliche Anwendung, bei der es noch zu Fehlermeldungen kommt und die daher womöglich etwas falsch macht. Sie können die Datenbankoberfläche sowohl über Delphis Programmgruppe als auch über das TOOLS-Menü der IDE starten.

Hinweis: Wenn es um das Untersuchen von Tabellen geht, ist der Datenbank-Explorer möglicherweise ein geeigneteres Werkzeug. Mit ihm können Sie komfortabler als mit der Datenbankoberfläche von Tabelle zu Tabelle wechseln und deren Inhalt modifizieren. Gestartet wird der Explorer entweder aus dem DATENBANK-Menü der IDE oder aus dem lokalen Menü einer Datenbankkomponente wie *TTable* oder *TQuery*.

Erstellen einer Tabelle

Um mit der Datenbankoberfläche die Tabelle für das Beispielprogramm aus Kapitel 7.3.6 anzulegen, wählen Sie den Menüpunkt DATEI | NEU | TABELLE, geben im darauf folgenden Fenster den Dateityp *Paradox 7.0 für Windows* an und gelangen so in eine Dialogbox, wie sie in Abbildung 7.1 gezeigt ist.

Der wichtigste Bestandteil dieser Dialogbox ist die *Feldliste*, in der Sie die Namen und Typen der einzelnen Felder festlegen, sie entspricht damit ungefähr der Record-Deklaration in Object Pascal:

- In der Spalte *Typ* haben Sie je nach dem gewählten Dateiformat eine unterschiedliche Auswahl an Datentypen. Für Paradox gibt es beispielsweise 17 Typen inklusive verschiedener Integer-, Binär- und Spezialformate (z.B. Grafik und OLE). Eine Auswahl aller Typen erhalten Sie, wenn Sie im entsprechenden Feld der Dialogbox die Leertaste oder die rechte Maustaste drücken. Für die Beispieltabelle können Sie die in Abbildung 7.1 zu sehenden Zeichen direkt in die Spalte eingeben.
- Die Spalte *Größe* ist nur bei Feldtypen gültig, die eine weitere Größenangabe benötigen, so z.B. bei alphanumerischen Feldern.
- Schließlich markieren Sie in der Spalte *Schlüssel* die Felder, aus denen der Tabellenschlüssel gebildet werden soll. Der Schlüssel hat im Pascal-Record keine Entsprechung und ist in Kapitel 7.4.2 ausführlich besprochen. In der Beispieltabelle aus Abbildung 7.1 sind die ersten beiden Felder als Schlüssel markiert, was bewirkt, dass die Datensätze zuerst nach der Verzeichnisnummer und dann nach dem Dateinamen sortiert werden, und in dieser Reihenfolge in der Tabelle abgelegt werden. Andere Reihenfolgen werden über Indizes hergestellt, siehe ebenfalls Kapitel 7.4.2.

Entsprechend der Abbildung 7.1 wurde auch die zweite Tabelle der Dateidatenbank definiert, sie besteht nur aus zwei Feldern: Einem Integer-Feld für die Verzeichnisnummer und einem String-Feld für den Verzeichnisnamen.

Tabellen im Programm-Code erzeugen

Ein Beispiel dafür, wie Sie dBase- und Paradox-Tabellen auch im Programmcode erzeugen können, finden Sie ebenfalls im Beispielprogramm *FileDB*. Wählen Sie im Hauptfenster dieses Programms den Menüpunkt TABELLE | NEU, so erstellt die folgende Methode beide Tabellen mitsamt den benötigten Indizes neu:

```
procedure TMainForm.CreateDBClick(Sender: TObject);
begin
  with Files do begin
    Active := False;
    DataBaseName := DBName;
```

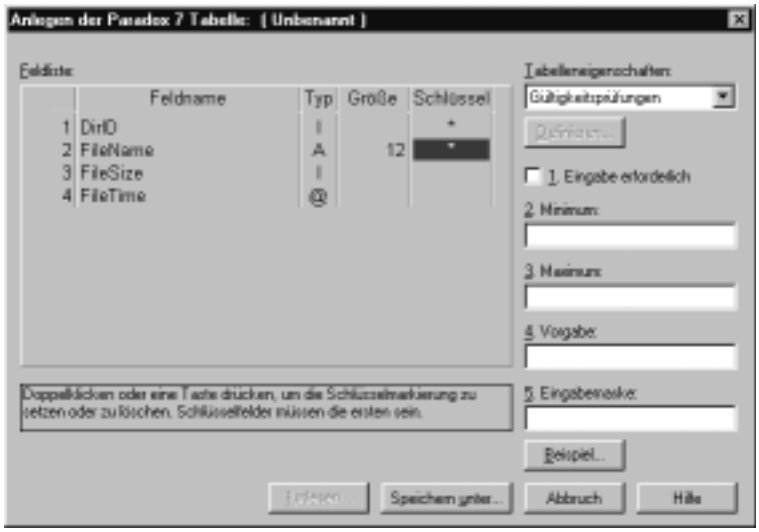


Abbildung 7.1: Die Struktur der ersten von zwei verknüpften Tabellen

```
      TableName := 'FILES.DB';
      TableType := TTDefault;
      with FieldDefs do begin
        Clear;
        Add('DirID', // Name des Feldes
            ftInteger, // Datentyp
            0,    // Größe (nur bei Datentypen variabler Größe benötigt)
            True);  // gibt an, ob diese Spalte einen Wert enthalten muss
        Add('FileName', ftString, 12, True);
        Add('FileSize', ftInteger, 0, True);
        Add('FileTime', ftDateTime, 0, True);
      end;
      with IndexDefs do begin
        Clear;
        Add('', 'DirID;FileName', [ixPrimary]);
        Add('BySize', 'DirID;Groesse', []);
        Add('ByDate', 'DirID;Datum', []);
      end;
      CreateTable;
      Active := True;
    end;
    with Dirs do begin
      ... und nun das gleiche mit der zweiten Tabelle, siehe CD ...
    end;
  end;
```

Die Methode baut sowohl die Feld- als auch die Indexdefinitionen der Tabellen neu auf (*FieldDefs* und *IndexDefs* werden also zu Beginn mit *Clear* gelöscht) und verwendet zum Abschluss die Methode *TTable.CreateTable*, um eine Tabelle nach der neuen Defini-

tion zu erzeugen. *FieldDefs* und *IndexDefs* werden in den Kapiteln 7.3.3 und 7.4.2 vorgestellt, für Details zur oben verwendeten *Add*-Methode und zu weiteren *TTable*-Methoden (etwa *TTable.AddIndex* für das nachträgliche Hinzufügen eines Indizes, *DeleteTable* und *EmptyTable* für das Löschen bzw. Leeren einer Tabelle) sei auf die Online-Hilfe verwiesen.

7.1.2 MyBase-Datenbanken

Zur Verwendung einer MyBase-Tabelle in einer Delphi-Anwendung müssen Sie weder die BDE, noch irgendein Datenbanksystem installieren oder konfigurieren, noch eine Verbindungskomponente für den Zugriff auf eine Datenbank einrichten, denn bei MyBase handelt es sich sozusagen um das »Privatformat« der *TClientDataSet*-Komponenten. Im Lieferumfang von Delphi befinden sich bereits eine ganze Reihe von MyBase-Tabellen, die sich gut zu Demonstrations- und Lernzwecken verwenden lassen, Sie finden diese zusammen mit den dBase- und Paradox-Beispieltabellen im Verzeichnis `Gemeinsame Dateien\Borland Shared\Data` jeweils im Binärformat (Dateiendung `.cds`) und im XML-Format (Dateiendung `.xml`).

Auch im Bereich der professionellen Datenbankanwendungen gibt es durchaus Verwendung für MyBase-Tabellen. Das Standardbeispiel dafür geben Anwendungen nach dem Aktenkoffermodell (briefcase model), bei denen eine Datenbank zwar auf einem Server gelagert ist, die Anwendung aber nicht immer mit diesem Server verbunden ist und daher für die Zeit der Trennung einen für den Anwender wichtigen Teilbereich der Daten auf dem lokalen PC verwaltet – eben in den MyBase-Dateien. Der »Anwender« kann z.B. ein Außendienstmitarbeiter sein, der auf seinen Reisen zusätzliche Daten sammelt, die zunächst in der MyBase-Datei aufbewahrt und später dann bei einem Wiedersehen des Servers zu diesem weitergeleitet werden.

Zwei wichtige Grenzen von MyBase-Dateien sollten Sie jedoch kennen:

- Die ClientDataSet-Komponenten halten immer die gesamte Datenmenge im Speicher und verwalten sie dort in einem eigenen Datenpaket namens *Data*. Zwar hat sich die Speicherausstattung der Computer seit der Erfindung der Datenbank um Zehnerpotenzen gesteigert, jedoch kann man auch den zu verwaltenden Daten nicht jegliches Wachstum absprechen und manche Datenbanken werden wohl immer zu groß sein, um sie komplett in den Speicher zu laden.
- Die ClientDataSet-Klassen sind zwar von *TDataSet* abgeleitet und erben damit zahlreiche nützliche Funktionen zum Bearbeiten, Filtern und Suchen von Daten, jedoch können Sie auf eine MyBase-Datei keine SQL-Abfrage anwenden. Die SQL-Funktionen der ClientDataSet-Klassen sind dafür vorgesehen, das *Ergebnis* einer SQL-Abfrage von einem externen Datenbank-Server aufzunehmen (dieses Ergebnis kann dann natürlich wieder in einer MyBase-Datei gespeichert, von dort aber nicht

mehr weiter per SQL bearbeitet werden, denn die SQL-Ausführung findet immer in einem externen Datenbank-System statt).

Wenn diese Grenzen nicht stören und Sie nicht die speziellen Fähigkeiten eines professionellen Datenbanksystems wie Transaktionskontrolle, Sicherheitsfunktionen etc. benötigen, dann geben Ihnen die MyBase-Datenbanken einen guten Ansatz zur Entwicklung einer Anwendung, die sich später auch leicht auf ein größeres System skalieren lässt. Bei der Verwendung von MyBase benötigen Sie neben der ausführbaren Datei Ihrer Anwendung übrigens nur die Datei `midas.dll`.

MyBase-Tabellen erzeugen *R178*

Um eine neue MyBase-Tabelle zu erzeugen, benötigen Sie zunächst eine Komponente der Klasse *TClientDataSet* oder einer davon abgeleiteten Klasse. Wenn Sie nur die MyBase-Funktionalität testen wollen, wird die Klasse *TClientDataSet* selbst schon genügen, ansonsten verwenden Sie statt dessen spezielle ClientDataSets wie *TSQLClientDataSet* (mit Anschluss an dbExpress) oder *TBDEClientDataSet* (mit Anschluss an die BDE). Fügen Sie also eine neue ClientDataSet-Komponente in ein Formular ein (mit bestehenden Komponenten aus Beispielprogrammen, die bereits mit irgendwelchen Tabellen verknüpft sind, funktioniert der folgende Ablauf nicht!) und starten Sie mit einem Doppelklick auf das Icon der Komponente (oder über ihr Popup-Menü) den Felder-Editor (Abbildung 7.2), der zunächst noch keinerlei Einträge enthält.

Aus seinem Popup-Menü können Sie nun den Punkt NEUE FELDER aufrufen, um in das ebenfalls in der Abbildung dargestellte Dialogfenster zu gelangen. Hier können Sie nun vier Eigenschaften eines neuen Feldes (d.h. einer neuen Tabellenspalte) festlegen: Den Namen des Feldes in der Datei, den Namen der Komponente, die dieses Feld auf Seiten der Delphi-Anwendung vertreten soll, den Datentyp, der in diesem Feld gespeichert werden soll, und je nach Datentyp noch die Größe des Feldes. Wichtig ist, dass der Radio-Schalter FELDTYP auf DATEN gestellt ist, wenn Sie das Feld in der Datei speichern wollen.

Auf diese Weise können Sie nun für jede Spalte der neuen Tabelle ein neues Feld definieren. Feldname, Komponentenname, Feldtyp (der im Dialog auf DATEN eingestellt sein sollte) und Datengröße sowie den Index des Feldes in der Felderliste können Sie nachträglich auch im Objektinspektor ändern, wenn Sie das Feld im Felder-Editor selektieren. Der Datentyp des Feldes lässt sich nachträglich nicht mehr ändern, weil er nicht durch ein Property, sondern durch die Klasse der Feldkomponente repräsentiert wird. Für ein String-Feld erhalten Sie beispielsweise eine Feldkomponente der Klasse *TStringField*. Um dies nachträglich zu ändern, müssen Sie also die Feldkomponente aus dem Felder-Editor löschen und ein neues Feld erzeugen.

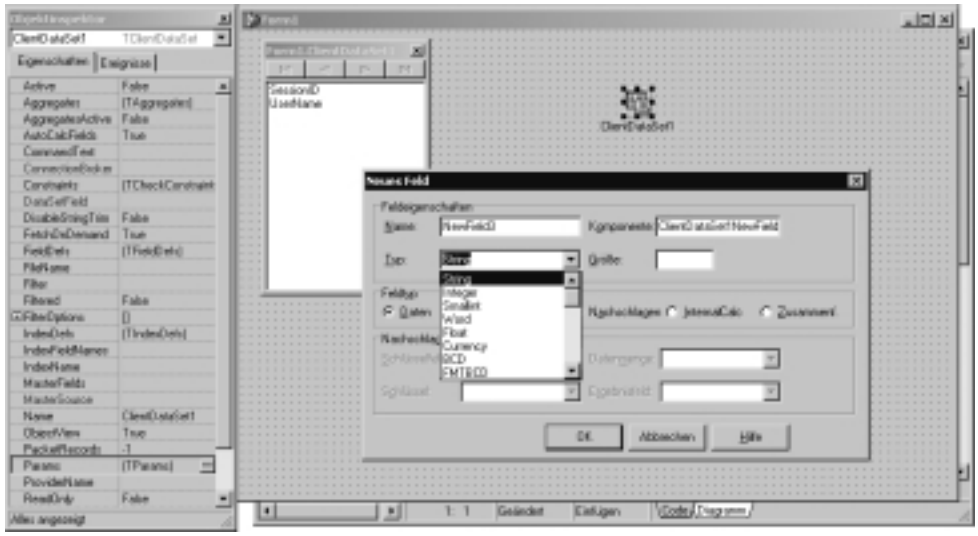


Abbildung 7.2: An der Erzeugung einer neuen MyBase-Tabelle sind der Feldereditor, ein Dialog und optional (z.B. für spätere Korrekturen) der Objektinspektor beteiligt.

Wenn Sie die Definition der Spalten abgeschlossen haben, führen Sie zwei Punkte des lokalen Menüs der ClientDataSet-Komponente aus:

- Mit DATASET ERSTELLEN bewirken Sie, dass die Komponente im Speicher eine Struktur aufbaut, die den von Ihnen definierten Tabellenspalten entspricht.
- Der Erfolg dieses ersten Schritts zeigt sich darin, dass das Popup-Menü danach einige zusätzliche Einträge enthält. Wählen Sie nun je nach dem gewünschten Dateiformat den Punkt IN MYBASE-XML-TABELLE SPEICHERN... oder IN BINÄRE MYBASE-DATEI SPEICHERN.

Den Namen der so erzeugten Datei können Sie nun optional, wenn Sie die ClientDataSet-Komponente weiterverwenden wollen, in deren *FileName*-Property angeben, so dass die Komponente zukünftig ihre Daten automatisch aus dieser Datei bezieht bzw. sie darin speichert.

Erzeugen der Tabellen zur Laufzeit

Der beschriebene Ablauf im Felder-Editor ähnelt stark dem Ablauf, mit dem Sie eine MyBase-Tabelle auch zur Laufzeit programmgesteuert erzeugen können. Sie nehmen eine leere Felderliste (*TClientDataSet.FieldDefs*) und fügen der Reihe nach die einzelnen Spalten hinzu (*Add*-Methode), wobei Sie den Datentyp und bei bestimmten Typen die Datengröße als Parameter angeben. Danach rufen Sie statt des Menüpunkts DATASET ERSTELLEN die Methode *CreateDataSet* auf.

Für die in Kapitel 7.3.6 verwendete Dateidatenbank würde dies etwa wie folgt aussehen:

```
with Files do begin
  Active := False;
  with FieldDefs do begin
    Clear;
    Add('DirID', ftInteger);
    Add('FileName', ftString, 12);
    Add('FileSize', ftInteger);
    Add('FileTime', ftDateTime);
  end;
  CreateDataSet;
end;
```

Der Code ähnelt also dem in Kapitel 7.1.1 gezeigten Code zur Erzeugung einer Paradox-Tabelle sehr, hier werden lediglich einige BDE-spezifische Properties wie *DataBaseName*, *TableName* und *TableType* nicht gesetzt und statt *CreateTable* wird *CreateDataSet* aufgerufen.

Beachten Sie, dass *TClientDataSet* nur dann eine Datei auf dem Datenträger erzeugt, wenn Sie explizit die *SaveToFile*-Methode aufrufen oder das *FileName*-Property gesetzt haben (die automatische Speicherung findet im letzteren Fall allerdings erst beim Schließen der Anwendung statt).

Hinweise: Im Vergleich zum Listing aus Kapitel 7.1.1 wurden oben die letzten Parameter der *Add*-Methoden weggelassen. Es gilt dann die Standardeinstellung von 0 für die Größenangabe und *False* für den *Required*-Parameter, der angibt, ob die Spalte immer einen Wert enthalten muss oder ob sie auch leer sein darf.

Das erwähnte Beispielprogramm aus Kapitel 7.3.6 verwendet übrigens keine MyBase-Tabellen, sondern liegt lediglich in Versionen für Paradox und Interbase vor. Daher kann der obige Code in diesem Programm *nicht* verwendet werden!

7.1.3 Interbase-Datenbanken

Mit Interbase steht ein leistungsfähiger Datenbank-Server unter Windows und Linux zur Verfügung, der in der Version 6 sogar als Open Source vorliegt. Alle Versionen von Delphi 6 mit Datenbankunterstützung enthalten auch eine 5-Benutzer-Lizenz für das von Borland mitgelieferte Interbase 6.0. Die auf der CD zu diesem Buch enthaltenen Beispieldatenbanken sind mit Interbase 5.6 erstellt worden, so dass sie auch mit früheren Delphi-Versionen funktionieren (Interbase 6 verwendet standardmäßig ein neues Dateiformat, das nicht mehr abwärtskompatibel ist; Dateien früherer Interbase-Versionen können jedoch ebenfalls gelesen werden).

Da es im Rahmen dieses Kapitels nicht möglich ist, einen Datenbankadministrations-Kurs durchzuführen, gehen die Beispielprogramme dieses Kapitels, die mit Interbase arbeiten, von einer lokalen Installation des Interbase-Servers aus und speichern ihre Daten in lokalen Interbase-Dateien. Nach der Installation von Interbase sollte der lokale Server bei jedem Systemstart automatisch »als Icon« oder als Systemdienst (nur unter Windows NT/2000/XP) gestartet werden. Sollte dies nicht der Fall sein, können Sie den Start beispielsweise über die Interbase-Programmgruppe im START-Menü nach- bzw. wiederholen (weitere Informationen dazu gibt der ebenfalls in dieser Programmgruppe installierte *Operations Guide* aus der Interbase-Dokumentation).

Für einen ersten Funktionstest genügen die von Borland mitgelieferten Beispieldatenbanken wie etwa `employee.gdb` im selben Verzeichnis wie die BDE- und MyBase-Beispieldateien.

In Kapitel 7.2.6 finden Sie eine Browser-Anwendung, mit der Sie den Inhalt dieser Datenbanken komfortabel untersuchen können, und in diesem Kapitel soll es zunächst um die Erzeugung einer Interbase-Datenbank gehen.

Interaktives SQL

Teil der Interbase-Installation unter Windows ist das Utility *wisql32.exe*, das Ihnen neben einigen dialogunterstützten Funktionen die Möglichkeit bietet, Kommandos an den Interbase-Server zu senden. Die Anfragen werden wie die Antworten von Interbase im Fenster des Tools angezeigt (siehe Abbildung 7.3).

Auf diese Weise können Sie sowohl normale SQL-Abfragen durchführen, die als Ergebnis Datenmengen aus bestehenden Tabellen haben, als auch die *Metadaten* einer Datenbank definieren (Daten, die Informationen über die Datenbank selbst liefern, werden als Metadaten bezeichnet). Zu diesen Metadaten gehört vor allem die Struktur der Datenbank, also etwa der Aufbau der Tabellen, jedoch gibt es aufgrund der umfangreichen Funktionalität von Interbase noch viele weitere Arten von Metadaten wie etwa Trigger, Generatoren und Stored Procedures. Dies alles zu dokumentieren ist Aufgabe spezieller Bücher; eine komplette Referenz finden Sie in der *Language Reference*, die Sie in der Interbase-Programmgruppe unter *Documentation* finden.

Während Sie im Dialog mit *wisql* gut Wartungsarbeiten, Tests und Experimente durchführen können, sollten Sie die eigentliche Definition von Datenbanken und ihren Strukturen über SQL-Skripte durchführen, die sich später zu beliebiger Zeit wiederholt ausführen lassen.

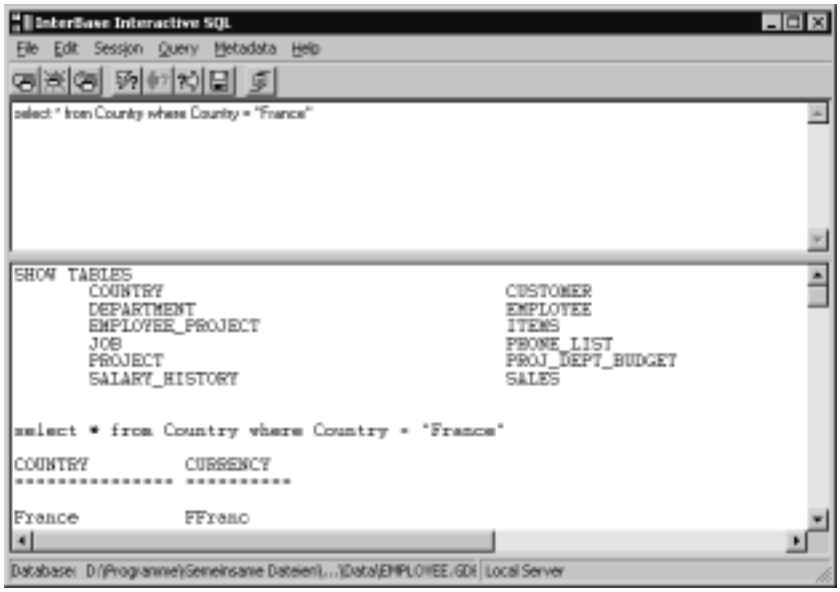


Abbildung 7.3: Im Dialog mit Interbase per wisql

Datenbanken erzeugen

Die Dateidatenbank aus Kapitel 7.3.6 liegt außer in der Version für Paradox-Tabellen (deren Erzeugung in Kapitel 7.1.1 kurz skizziert wurde) auch in einer Version für Interbase vor. Um Ihnen einen ersten Eindruck von der SQL-Data Definition Language (DDL) zu vermitteln, ist im Folgenden das Skript zur Erzeugung der Interbase-Dateidatenbank komplett abgedruckt.

```
CREATE DATABASE "FileDB.gdb" USER 'SYSDBA' PASSWORD 'masterkey'
  PAGE_SIZE 1024;

CREATE TABLE Directories(DirID INTEGER NOT NULL, Path VARCHAR(80)
  NOT NULL);
CREATE TABLE Files(DirID INTEGER NOT NULL, Filename VARCHAR(80)
  NOT NULL, FileSize INTEGER, FileTime DATE);

CREATE UNIQUE INDEX DIRECTORIESO ON Directories(DirID);
CREATE INDEX Path ON Directories(Path);

CREATE UNIQUE INDEX FILESO ON Files(DirID, FileName);
CREATE INDEX BySize ON Files(DirID, FileSize);
CREATE INDEX ByDate ON Files(DirID, FileTime);
```

Es besteht aus mehreren Anweisungen, die jeweils mit einem Semikolon abgeschlossen werden. Im Dialog mit *wisql* ist es übrigens nicht erforderlich, eine einzelne Anweisung mit einem Semikolon abzuschließen (im Gegensatz etwa zur Arbeit mit dem

reinen Befehlszeilen-Tool *isql*, das nicht in der Interbase-Programmgruppe installiert ist, aber von der Befehlszeile aus gestartet werden kann). Um eine Anweisung auszuführen, muss der entsprechende Schalter in der Toolbar von *wisql* bzw. Strg+Enter gedrückt werden.

In der ersten Anweisung wird eine Datenbank angelegt und eine neue Datei namens `FileDB.gdb` erzeugt. Als Benutzername und Passwort sind im obigen Listing die Standard-Zeichenketten angegeben, die auch in den mit Interbase mitgelieferten Beispieldatenbanken verwendet werden.

In den Anweisungen zwei und drei werden die beiden Tabellen der Dateidatenbank definiert; beide Anweisungen enthalten eine Liste von Feldern, für die jeweils Name, Datentyp und evtl. Datengröße angegeben werden. Für die einzelnen Felder können Sie bei Interbase auch einige zusätzliche Eigenschaften festlegen, die es bei MyBase-Dateien nicht gibt. Im obigen Listing wird lediglich die Eigenschaft »NOT NULL« angegeben, die besagt, dass dieses Feld niemals leer sein darf (alle Datensätze der Tabelle müssen in diesem Feld also einen Wert aufweisen).

Hinweis: Die »NOT NULL« -Eigenschaft lässt sich in MyBase-Tabellen nachbilden: Sie ist zwar im Dialog zum Erstellen eines neuen Feldes in ClientDataSet-Komponenten (siehe Kapitel 7.1.2) nicht berücksichtigt, kann dort aber nachträglich über das Property *Required* der Feldkomponente eingestellt werden.

Die restlichen Zeilen des Listings sorgen für die Erzeugung verschiedener Indizes, die in dieser Datenbank nützlich sein *könnten*, wenn das Beispielprogramm die Such- und Sortierfunktionen von Interbase nutzen und nicht immer die gesamten Tabellen in das ClientDataSet laden würde, wo Such- und Sortierfunktionen dann unabhängig von Interbase von der Delphi-Komponente durchgeführt werden (zur Verwendung von Indizes in den ClientDataSets des Programms siehe Kapitel 7.4.4 und 7.4.5).

Für die Syntax der im Listing verwendeten DDL-Anweisungen und die verfügbaren Optionen sei auf die Interbase-Sprachreferenz verwiesen.

Skripte ausführen

Während Sie mit *wisql* Skripte über das FILE-Menü ausführen können, wird häufig auch ein Weg benötigt, Skripte programmgesteuert auszuführen. Hierzu eignet sich die Befehlszeilenversion des Tools, *isql*, mit der das obige Skript wie folgt ausgeführt werden kann:

```
isql -i CreateFileDB.sql
```

Das Programm *IBFileDB* (die Interbase-Version der Beispielanwendung) besitzt einen Menüpunkt DATENBANK | DATENBANK UND TABELLEN PER SKRIPT ERZEUGEN, mit dem das Skript durch das Programm ausgeführt werden kann. Der Code des Menüpunkts sieht wie folgt aus:

```
procedure TMainForm.CreateDBClick(Sender: TObject);
var
  ReturnCode: Cardinal;
  isqlPath, ScriptPath: String;
  pInfo: PROCESS_INFORMATION;
  sInfo: STARTUPINFO;
begin
  // Pfade zusammensetzen:
  isqlPath := IncludeTrailingPathDelimiter(FileDBOptions.isqlPath.Text)
                    + 'isql';
  ScriptPath :=
    IncludeTrailingPathDelimiter(ExtractFilePath(Application.ExeName))
    + 'CreateFileDB.sql';

  // Datenstruktur für CreateProcess vorbereiten:
  fillchar(sInfo, sizeof(sInfo), 0);
  sInfo.cb := sizeof(STARTUPINFO);
  sInfo.wShowWindow := SW_SHOW; // isql-Fenster für den Benutzer sichtbar

  // Ausführung von ISQL im Verzeichnis, das im Optionsdialog
  // eingestellt ist:
  chdir(FileDBOptions.gdbPath.Text);
  if (not CreateProcess(nil, PChar(isqlPath+' -i '+ScriptPath),
                        nil, nil, FALSE, 0, nil, nil,
                        sInfo, pInfo)) then
     ShowMessage('isql konnte nicht ausgeführt werden. Sind die '+
                 'Pfadeinstellungen korrekt?')
  else begin
     WaitForSingleObject(pInfo.hProcess, INFINITE);
     GetExitCodeProcess(pInfo.hProcess, ReturnCode);

     if (ReturnCode <> 0) then
       ShowMessage('isql gab den Fehler-Code ' + IntToStr(ReturnCode) +
                   ' zurück. Möglicherweise konnte das Skript '+
                   'nicht fehlefrei ausgeführt werden.');
   end;
end;
```

Der Pfad von *isql* muss dabei vor dem Aufruf des Menüpunkts im Optionsdialog des Beispielprogramms (DATENBANK | OPTIONEN...) eingestellt werden; die obige Methode greift auf diese Einstellung zurück (*FileDBOptions.isqlPath*). Da das Skript im selben Verzeichnis gespeichert ist wie die EXE-Datei der Anwendung, setzt sich der Skript-Pfad (*ScriptPath*) aus dem Pfad der EXE-Datei (*Application.ExeName*) und dem Namen der Skript-Datei zusammen.

Nach der Einstellung der Pfade wird *isql* mit der Windows-API-Funktion *CreateProcess* gestartet, deren ausschweifige Parameterliste überwiegend mit Nullen gefüllt wird. Im Erfolgsfall (wenn *CreateProcess* keine Null zurückliefert), wartet die API-Funktion *WaitForSingleObject* darauf, dass das gestartete Skript beendet wurde. Mit einer dritten API-Funktion *GetExitCodeProcess* wird schließlich der Fehlercode abgefragt, der von *isql* zurückgeliefert wird (etwa, wenn das angegebene Skript nicht existiert; bei Fehlern während der Ausführung eines Skripts wird *kein* Fehlercode zurückgegeben).

Wenn *isql* seine Arbeit erfolgreich durchführen konnte, ist das Ergebnis immer eine Interbase-Datei *FileDB.gbd*. Da das Skript keine absolute Pfadangabe enthält, legt *isql* diese Datei im jeweils aktuellen Verzeichnis an. Das Beispielprogramm erlaubt es Ihnen im bereits erwähnten Dialog (DATENBANK | OPTIONEN...) ein Verzeichnis vorzugeben. Um *isql* in dieses Verzeichnis zu führen, muss es vor dem Start von *isql* zum aktuellen Verzeichnis gemacht werden, was in der obigen Methode durch die Standard-Funktion *chdir* erledigt wird.

Zugriff auf Interbase-Datenbanken mit der BDE *R176*

Während Kapitel 7.1.6 beschreibt, wie Sie mit dbExpress auf lokal gespeicherte Interbase-Datenbanken zugreifen, sei hier kurz die Zugriffsmethode mit der BDE zusammengefasst: Zunächst kommen dieselben Komponenten für den Datenzugriff in Frage wie für den Zugriff auf Paradox / dBase-Tabellen (die als Alternative verfügbaren speziellen Interbase-Express-Zugriffskomponenten können in diesem Buch nicht näher behandelt werden), und auch hier beschränken wir uns vorerst auf den einfachsten Fall der *TTable*-Komponenten, in deren *TableName*-Property Sie eine Tabelle der Datenbank auswählen können. Bei der Angabe des Datenbanknamens ist im Falle einer Interbase-Datenbank die Verwendung eines Alias-Namens zwingend erforderlich.

Um einen globalen Alias auf eine Interbase-Datenbank einzurichten, erzeugen Sie etwa im Datenbank-Explorer (Delphi-IDE: DATENBANK | EXPLORER) mit OBJEKT | NEU... einen neuen Alias, wählen aus der Treiberliste den Treiber »*Intrbase*« und geben daraufhin im rechten Teil des Datenbank-Explorer-Fensters unter SERVER NAME den Pfad der lokalen Interbase-Datei an. Im Datenbank-Explorer wird die Alias-Definition durch Drücken des ÜBERNEHMEN-Schalters abgeschlossen (gebogener blauer Pfeil).

Ansonsten gibt es die gleichen Möglichkeiten wie beim Zugriff auf dBase / Paradox-Tabellen: Entweder Sie geben den Alias-Namen direkt in allen *DatabaseName*-Properties der Datenmengen-Komponenten an oder Sie verwenden für alle Datenmengen eine gemeinsame *TDatabase*-Komponente (siehe Kapitel 7.1.1).

7.1.4 Das Konzept der Datenquelle

Schon seit der ersten Version unterscheidet Delphi zwischen Komponenten für den Datenzugriff (nicht-visuelle Komponenten der Palettenseite *Datenzugriff*) und Kompo-

nenten für die visuelle Präsentation und Manipulation der Daten (visuelle Komponenten auf der Palettenseite *Datensteuerung*). Zu Letzteren gehören Editierfelder, Markierungsfelder und Listen, in denen Daten angezeigt, ausgewählt und editiert werden können. Die Zugriffskomponenten dienen dazu, den visuellen Steuerelementen die Daten bereitzustellen.

Ein zentrales Konzept dabei ist das der *Datenquelle*, realisiert durch die Komponente *TDataSource*. Alle erwähnten visuellen Daten-Steuerelemente werden an eine solche DataSource angeschlossen, beziehen daraus ihre Daten und reichen sie an diese weiter, wenn sie editiert wurden und in die Datenbank bzw. das Änderungsprotokoll zurückgeschrieben werden sollen. Eine DataSource steht dabei normalerweise für eine einzelne Tabelle bzw. Abfrage (allgemein: eine Datenmenge).

Hinweis: Die Verknüpfung von Datenmenge über Datenquelle zu den visuellen Steuerelementen findet über die Properties *TDataSource.DataSet* sowie die *DataSource*-Properties der visuellen Steuerelemente statt, was aber in Kapitel 7.2.1 noch ausführlicher erläutert werden wird.

Der Grund, warum die Daten-Steuerelemente nicht direkt mit der Tabelle bzw. Abfrage verknüpft werden, sondern nur über den Umweg über eine DataSource, wird mit jedem Mal, wenn Borland eine neue Datenzugriffsmethode einführt, neu unterstrichen: Wird eine DataSource verwendet, können Sie den gesamten Datenzugriff neu zusammenstellen, ohne etwas an den Daten-Steuerelementen zu ändern. Sie können z.B. 10 verschiedene Steuerelemente an die DataSource anschließen und brauchen doch nur eine einzige DataSource-Komponente zu ändern, um etwa zwischen den folgenden Arten des Datenzugriffs zu wechseln:

- Datenzugriff über die BDE unter Windows, z.B. zur Nutzung lokaler dBase- oder Paradox-Tabellen: Die DataSource kann dazu direkt an eine *TTable* oder *TQuery*-Komponente angeschlossen werden.
- Datenzugriff über die Interbase-Express-Komponenten für direkten Zugriff auf Interbase ohne die BDE.
- Datenzugriff über ADO-Komponenten, falls unter Windows die Datenbankschnittstelle von Microsoft genutzt werden soll, etwa zur Verwendung von Microsoft-Datenbanken.
- Datenzugriff über eine *ClientDataSet*-Komponente, welche mit Hilfe der MIDAS-Bibliothek Daten von einer entfernten, mit Delphi erzeugten Server-Anwendung bezieht.
- Datenzugriff über die neue dbExpress-Bibliothek: Die DataSource wird an ein *ClientDataSet* angeschlossen, welches wiederum auf verschiedene Weise mit weiteren

Zugriffskomponenten verbunden werden kann. Einige der Variationsmöglichkeiten sind am Ende von Kapitel 7.1.6 aufgeführt.

Ein Vorhaben, bei dem es zwingend zu einer Änderung des Datenzugriffs und somit zu einem »Umstöpseln« der DataSource kommt, ist das Portieren einer dBase/Paradox-Datenbankanwendung auf die neue dbExpress-Architektur.

7.1.5 Portierung von Desktop-Datenbanken

Dieses Kapitel beschreibt den grundsätzlichen Ablauf, mit dem eine dBase/Paradox-Datenbankanwendung auf die neue plattformübergreifend verfügbare dbExpress-Architektur umgestellt werden kann. Da Borland keinen dBase/Paradox-Treiber für dbExpress mitliefert, muss die Datenbank zunächst in ein neues Format konvertiert werden (sofern nicht jemand anderes als Borland einen dBase- oder Paradox-Treiber entwickelt).

Als Ziel der Konvertierung kommen theoretisch alle auch unter Kylix unterstützten Formate in Frage, im Folgenden ist jedoch nur die Konvertierung in das Interbase-Format näher untersucht. Delphi bringt ab der Professional-Ausgabe bereits ein Utility zur Konvertierung von Desktop-Datenbanken mit, mit dem unter anderem auch Interbase-Datenbanken erzeugt werden können.

Hinweis: Für Konvertierungen, die sich mit Data Pump nicht durchführen lassen, können Sie auch selbst ein kleines Delphi-Programm schreiben. In den BDE-Komponenten von Delphi finden Sie als kleine Hilfe dabei die Komponente *TBatchMove*, die automatisch den Inhalt einer Quell- in eine Zieltabelle schreibt.

Das Data Pump-Tool

Sie finden das Data Pump-Tool bei einer vollständigen Installation von Delphi Professional in Delphis Programmgruppe unter dem Namen *Data Pump*. Das Tool ist in der Form eines Wizards gehalten, besteht also aus mehreren Dialogseiten, zwischen denen Sie mit den Schaltern WEITER und ZURÜCK wechseln können. Wenn Sie auf diesen Seiten alle erforderlichen Angaben gemacht haben, drücken Sie auf der letzten Seite (siehe Abbildung 7.4) den Button UPSIZE, um aus den gewählten Einzel-Tabellen eine einzige Datei im Interbase-Format (Endung .GDB) zu erzeugen.

Nachdem auf den ersten drei, in der Abbildung nicht gezeigten Seiten zunächst die zu konvertierende Datenbank, ein BDE-Alias für die Ziel-Datei und die zu konvertierenden Tabellen ausgewählt werden, geht es auf der vierten Seite des Dialogs darum, eventuelle Probleme beim Konvertieren aufzuzeigen und zu lösen. Für jede für die Konvertierung gewählte Tabelle werden die Felddefinitionen, die Indizes und die referenzielle Integrität überprüft:

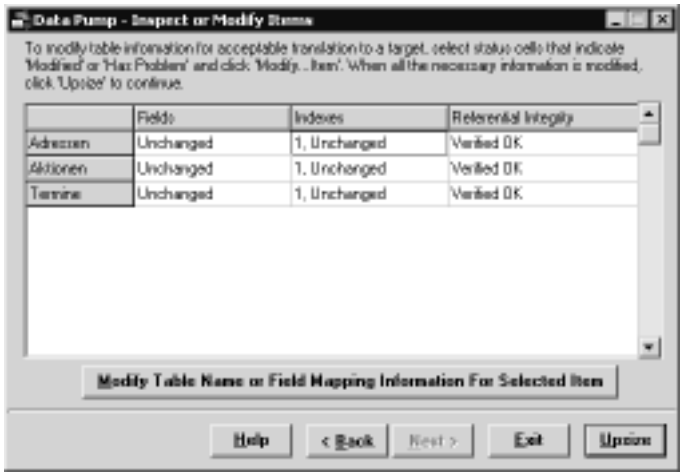


Abbildung 7.4: Konvertieren einer Desktop-Datenbank in das Interbase-Format

- Bei der Übersetzung der Felddefinitionen können Umbenennungen erforderlich werden, wenn in der Desktop-Datenbank ein Feldname verwendet wurde, der unter Interbase nicht erlaubt ist, weil es sich um ein reserviertes Wort handelt. Das Data Pump-Utility versieht diese Felder automatisch mit einem neuen Namen; so erhält etwa das Feld SIZE den Namen »SIZ_«. Diesen Namen sowie weitere Übersetzungsparameter (verwendeter Interbase-Datentyp, Minimal- und Maximalwert etc.) können Sie jedoch im Dialog von Data Pump einstellen.
- Auch die für die Tabelle geführten Indizes versucht Data Pump zu übertragen. Im Dialog können Sie die mit Interbase erzeugten Indizes anpassen, wenn auch auf eine etwas komplizierte Weise (zwischen den einzelnen Indizes kann nur mit WEITER und ZURÜCK hin- und hergeschaltet werden).
- Die Referentielle Integrität kann unter anderem verletzt werden, wenn Sie nicht alle zusammengehörigen Tabellen gewählt haben und so die Spalte einer gewählten Tabelle mit der Spalte einer nicht gewählten Tabelle verknüpft ist. Dieser Fehler lässt sich dadurch beheben, dass Sie im Dialog zurück zur Tabellenauswahl gehen und die fehlende Tabelle hinzufügen.

Hinweis: Da Data Pump für die Konvertierung auf eine Interbase-Datenbank zugreifen muss, muss der Interbase-Dienst gestartet sein. Falls Sie diesen nicht für einen automatischen Start konfiguriert haben, können Sie den Start manuell über das Interbase-Untermenü aus dem Start-Menü nachholen (die korrekte Installation von Interbase vorausgesetzt).

Test der neuen Interbase-Datei

Nachdem Sie einzelne Tabellen mit Data Pump in eine Interbase-Datenbank umgewandelt haben, müssen natürlich auch noch die Verweise innerhalb der Delphi-Anwendung von den lokalen Tabellen auf die neue Datenbank umgeleitet werden. Dies ist allerdings nur für einen ersten Test sinnvoll, denn die *TTable*- und *TQuery*-Komponenten stehen unter dbExpress nicht mehr zur Verfügung. Folgende Änderungen sind notwendig, um die alte Anwendung *vor* der Umstellung auf dbExpress mit der Interbase-Datenbank zusammenarbeiten zu lassen:

- Zunächst müssen die *DatabaseName*-Properties der *TTable*-, *TQuery*- oder *TDatabase*-Komponenten auf den Alias der Interbase-Datenbank weisen. Falls Sie auch für die alten Tabellen bereits ein Alias verwendet haben, können Sie dieses löschen (oder umbenennen) und für die neue Interbase-Datenbank ein Alias des alten Namens erzeugen, denn dann brauchen Sie die *DatabaseName*-Properties, in denen dieses Alias genannt wird, *nicht* zu ändern.
- Anders bei den *TableName*-Properties von *TTable*-Komponenten bzw. bei der Angabe von Tabellen in SQL-Anweisungen von *TQuery*: Bei lokalen Datenbanken enthalten die Tabellen-Namen auch eine Dateiendung wie etwa `.db`. Diese Endung muss jeweils entfernt werden, denn sie gehört nicht zum Namen der Tabelle in der Interbase-Datenbank.

Wenn Sie die Tabellen- oder Datenbanknamen noch an anderer Stelle in Ihrem Projekt verwendet haben (z.B. kann der Tabellenname in SQL-Queries vorkommen), müssen Sie natürlich auch diese anpassen. Häufig reichen diese Änderungen bereits aus, um Ihre Anwendung mit Interbase laufen zu lassen. Sie müssen dann nur noch – wie schon für die Benutzung von Data Pump – sicherstellen, dass der lokale Interbase-Server läuft.

Auch die Änderung von Feldnamen in Data Pump kann sich natürlich auf eine Delphi-Anwendung auswirken. Wenn Sie für ein geändertes Feld eine statische *TField*-Komponente erzeugt haben, passen Sie deren *FieldName*-Property an oder entfernen die ganze Feldkomponente aus dem Felder-Editor der zugehörigen Datenmengen-Komponente und fügen es dann mit FELDER HINZUFÜGEN wieder ein (dadurch ändert sich aber auch der Name der Komponente und Ihre speziellen Property-Einstellungen an der alten Komponente müssen wiederholt werden). Falls der Name des Feldes auch direkt im Object-Pascal-Quelltext genannt wird, sind eventuell einige weitere eher mechanische Umbenennungs-Handgriffe erforderlich.

Hinweis: Für die Autoinkrement-Felder von Paradox gibt es in Interbase keine entsprechenden Feldtypen, so dass diese Eigenschaft bei der Konvertierung verloren geht. Anwendungen, die sich daher beispielsweise auf automatisch erzeugte Datensatznummern verlassen, werden mit der so konvertierten Interbase-Datei nicht mehr korrekt funktionieren. In Server-Datenbanken wie Interbase werden Autoinkrement-Felder durch flexiblere Mechanismen nachgebildet wie etwa durch Trigger und Generatoren. Kapitel 7.5.1 gibt ein Beispiel für den Ersatz von Autoinkrement-Feldern durch Trigger.

Datenzugriffskomponenten für dbExpress und BDE

dbExpress bringt neue Datenzugriffskomponenten, arbeitet aber mit denselben Datenanzeigekomponenten (Palettenseite *Datensteuerung*) wie frühere Delphi-Versionen. Da BDE- und dbExpress-Datenmengen zudem mit dem bewährten *TDataSet* eine gemeinsame Vorfahr-Klasse haben, gestaltet sich die Portierung leichter, als man von einer neuen Architektur vielleicht erwarten würde.

Die Portabilität der Datenanzeigekomponenten bedeutet grob gesagt, dass Sie den Teil Ihrer Anwendung, der die *TDataSource*-Komponenten als Datenquelle verwendet, unverändert übernehmen können. Für dbExpress geändert werden müssen nur die Komponenten, die sich zwischen Interbase und der DataSource befinden.

Die dbExpress-Komponenten, die von Borland als Ersatz für die BDE-Komponenten *TTable*, *TQuery* und *TStoredProc* vorgeschlagen werden, sind nur sehr eingeschränkt verwendbar, da sie unidirektional sind und Sie mit diesen Komponenten beispielsweise immer nur einen Datensatz gleichzeitig anzeigen können. Um die volle Funktionalität der Anwendung zu erhalten, müssen *TTable*, *TQuery* und *TStoredProc* daher meistens durch *TSQLClientDataSet*-Komponenten ersetzt werden. In dessen Property *CommandType* wählen Sie dann aus *ctTable*, *ctQuery* und *ctStoredProc* den genauen Typ der Datenmenge. In *CommandType* machen Sie die restlichen Angaben, die etwa bei *ctTable* aus dem Namen der Tabelle, bei *ctQuery* aus der SQL-Anfrage, die in den BDE-Komponenten noch unter *TQuery.SQL* angegeben wurde, besteht.

Weder in *TSQLClientDataSet* noch in den anderen Datenmengen-Komponenten von dbExpress werden Sie das bekannte *DatabaseName*-Property wiederfinden, da die verwendete Datenbank in *dbExpress* auf eine der drei folgenden Arten angegeben wird:

- In *FileName* können Sie eine MyBase-Datei angeben (siehe Kapitel 7.2.5).
- Eine Verbindung zu einem Datenbank-Server im Allgemeinen und einer Interbase-Datei in unserem speziellen Fall kann über *ConnectionName* ausgewählt (vergleichbar mit einem Alias-Namen der BDE)
- oder über eine eigene *TSQLConnection*-Komponente definiert werden. Diese Komponente entspricht in etwa der *TDatabase*-Komponente einer BDE-Anwendung.

Der Verbindungsaufbau über *ConnectionName* oder eine *TSQLConnection*-Komponente wird in Kapitel 7.1.6 näher erläutert.

Nach dem Austausch der Datenzugriffskomponenten sind noch einige weitere Umstellungen zu erwarten. Diese können hier nicht im Detail beschrieben werden und werden auch in der Online-Hilfe von Delphi ausführlich dokumentiert (unter *CLX für die plattformübergreifende Entwicklung verwenden / Datenbankanwendungen auf Linux portieren*). Für einfache Programme wie etwa die Dateidatenbank aus Kapitel 7.3.6 genügt es fast schon, die statischen Felder für die ClientDataSets im Felder-Editor neu zu erzeugen, die DataSource-Komponenten der Anwendung mit den ClientDataSets zu verbinden und die Komponenten nach Bedarf zu aktivieren (*Active*-Property der ClientDataSets und das *Connected*-Property der Verbindungskomponente).

Eine Überprüfung ist in jedem Fall auch bei den Update-Mechanismen notwendig, da Änderungen des Benutzers in *TClientDataSet*-Komponenten nicht ebenso automatisch in die Datenbank übernommen werden wie unter der BDE (siehe hierzu Kapitel 7.2.7).

7.1.6 Datenzugriff mit dbExpress

Damit eine Delphi-Anwendung auf beliebige Datenbank-Server zugreifen kann, ist eine flexible Architektur gefragt. Der Zugriff auf den externen Server (»extern« meint hier lediglich: außerhalb der Delphi-Anwendung) wird in dbExpress durch Verbindungskomponenten gekapselt, die in *TCustomConnection* eine gemeinsame Basisklasse haben. Speziell dient etwa *TSQLConnection* (in der Komponentenpalette auf der Seite *dbExpress*) dazu, Verbindung zu allerlei SQL-Servern aufzunehmen. An diese Verbindungskomponenten können Sie dann andere Komponenten der *dbExpress*-Seite wie etwa *TSQLClientDataSet* anschließen.

Definition der Verbindungen *R175*

Alle auf dem System verfügbaren Verbindungen werden von DBExpress in einer Konfigurationsdatei definiert, die Sie je nach Sprachversion Ihres Betriebssystems und den Pfadeinstellungen, die Sie bei der Installation von Delphi gemacht haben, etwa unter `...\Programme\Gemeinsame Dateien\Borland Shared\DBExpress\dbxdrivers.ini` finden. Diese Datei ist zwar als Textdatei in einem einfachen Format (Ini-Format) leicht zu bearbeiten, jedoch können Sie über eine *TSQLConnection*-Komponente auch dialoggesteuert neue Verbindungen definieren oder bestehende Verbindungen ändern (Abbildung 7.5); die Komponente aktualisiert dann entsprechend die Konfigurationsdatei.

Um beispielsweise eine Verbindung für eine neu erzeugte Interbase-Datenbank zu definieren, fügen Sie zunächst eine *TSQLConnection*-Komponente in ein Formular ein, rufen dann aus dem Popup-Menü dieser Komponente oder per Doppelklick den Verbindungseditor auf, klicken auf den »+«-Schalter, wählen als Treiber »Interbase« und nehmen dann in der in Abbildung 7.5 zu sehenden Tabelle auf der rechten Seite des

Fensters die notwendigen Einstellungen vor. Im Minimalfall brauchen Sie hier nur im Feld DATABASE die Datenbank anzugeben; empfehlenswert ist hier eine absolute Pfadangabe, auch wenn relative Pfadangaben funktionieren, solange Sie die Verbindung nur aus einem festen Verzeichnis heraus nutzen.

Hinweis: Bei der Auswahl des Treibernamens werden automatisch zwei weitere wichtige Properties der Verbindungskomponente eingestellt, die die Namen der Treiberbibliotheken angeben. So weist beispielsweise im Falle des Interbase-Treibers das Property *LibraryName* auf *dbexpint.dll*, das Property *VendorLib* auf *gds32.dll*. Unter Linux werden andere Dateien verwendet (*libsqlib.so.1* und *libgds.so.0*), so dass Sie, wenn Sie eine unter Kylix entwickelte dbExpress-Anwendung unter Delphi kompilieren wollen oder umgekehrt, diese beiden Properties aktualisieren müssen, was automatisch geschieht, wenn Sie den Treibernamen neu zuweisen. Sie können dies im Objektinspektor über das Property *DriverName* tun.

Login-Daten

Wenn Sie die Möglichkeit des automatischen Logins nutzen wollen, geben Sie außerdem noch einen USER_NAME und ein PASSWORD an. Wenn eine Datenbank sowieso nur durch das Standard-Passwort *masterkey* für den Benutzernamen *SYSDBA* geschützt ist, ist dies sicher eine gute Lösung, um ständige Login-Dialoge zu vermeiden. Für kritischere Datenbanken ist die Angabe der Login-Daten in der Verbindungsdefinition jedoch nicht geeignet, da sie im Klartext innerhalb der Datei *dbxdrivers.ini* gespeichert wird.

Um das automatische Login zu aktivieren, müssen Sie außerdem noch das Property *TSQLConnection.LoginPrompt* auf *False* setzen, sonst erhält der Benutzer trotz einer Passwort-Angabe in den Verbindungsdaten einen Login-Dialog.

Hinweis: Alle Verbindungsdaten für eine *TSQLConnection* (*Params*-Property) werden in der Formulardatei des Datenmoduls gespeichert und daraus auch beim Programmstart wieder geladen. Die aktuellen Einstellungen aus *dbxconnections* werden *nicht* geladen, solange Sie das Property *TSQLConnection.LoadParamsOnConnect* bei seinem voreingestellten Wert *False* belassen. Dies bedeutet, dass Sie zur Installation einer Anwendung auf einem anderen PC nur die ausführbare Datei und den Datenbanktreiber installieren, sofern die Verbindungsparameter (wie etwa der Pfad zur Datenbank) sich ohne Änderung auf diesen anderen PC übertragen lassen. In Kapitel 7.2.6 wird eine Möglichkeit vorgestellt, wie Sie Verbindungsdaten programmgesteuert definieren oder ändern können, ebenfalls ohne Verwendung der Konfigurationsdatei.

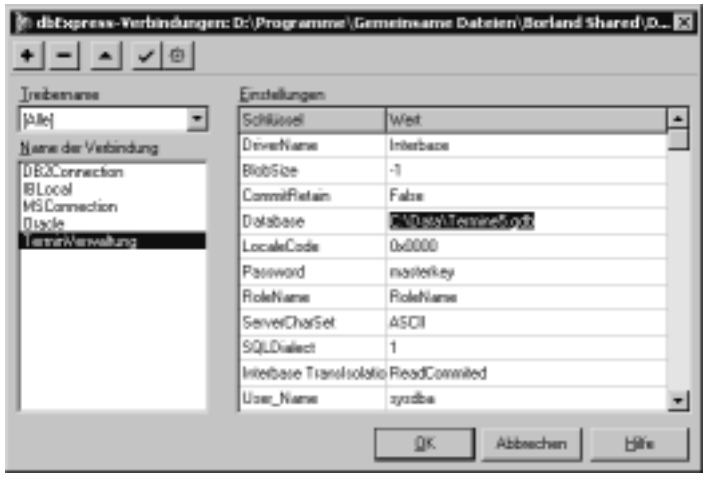


Abbildung 7.5: Definition einer neuen Datenbankverbindung im Einrichtungsdialog von TSQLConnection

Herstellen der Verbindung

Um nun eine der so definierten Verbindungen in einer Delphi-Anwendung zu nutzen, können Sie den Namen der Verbindung (in Abbildung 7.5 in der linken Listbox zu sehen) entweder direkt in einer Datenmengen-Komponente wie *TSQLClientDataSet* im *ConnectionName*-Property angeben, oder Sie kapseln die Verbindung über eine eigene Komponente der Klasse *TSQLConnection*, mit deren Verbindungseditor Sie die Verbindung ja vielleicht erst erzeugt haben. In diesem Fall setzen Sie das *ConnectionName*-Property der *TSQLConnection*-Komponente und wählen diese im Property *DBConnection* der Datenmengen-Komponente aus.

Der Vorteil der Kapselung in der eigenen Komponente liegt darin, dass Sie die Verbindung später leichter ändern können. Angenommen, Sie haben fünf *TClientDataSet*-Komponenten, die alle dieselbe Verbindung nutzen. Sie können diese Verbindung direkt im *ConnectionName*-Property dieser Komponenten angeben, müssen dann aber beim Wechsel der Verbindung alle fünf Properties ändern. Wenn Sie dagegen das Property *DBConnection* der fünf ClientDataSets mit einer Verbindungskomponente verknüpft haben, müssen Sie als Einziges das *ConnectionName*-Property dieser Komponente anpassen.

Variationen des Datenzugriffs

Für die an die Verbindungskomponenten angeschlossen Datenmengenkomponenten bietet dbExpress die folgenden Variationen:

- In den Beispielprogrammen dieses Buchs wird entweder ein *TClientDataSet* verwendet, das an eine MyBase-Datei angeschlossen ist (siehe Kapitel 7.2.5), oder ein

TSQLClientDataSet, welches über eine *TSQLConnection*-Komponente mit einer Interbase-Datenbank verbunden ist.

- Eine Neuerung von dbExpress gegenüber der BDE stellen die unidirektionalen Datenmengen dar, zu denen *TSQLDataSet*, *TSQLTable*, *TSQLQuery* und *TSQLStoredProc* gehören (dabei sind die letztgenannten drei nur spezielle Arten von *TSQLDataSet*, die eine gewisse Abwärtskompatibilität zu den BDE-Komponenten *TTable*, *TQuery* und *TStoredProc* herstellen sollen). Diese unidirektionalen Datenmengen können Sie anstelle eines ClientDataSets verwenden, wenn Sie eine besonders schnelle Verbindung zum Datenbank-Server benötigen und die Einschränkungen der unidirektionalen Datenmengen keine Rolle spielen (die Datenmenge kann nur in einer vorgegebenen Reihenfolge durchlaufen werden, eine Anzeige in *DBGrid*-Komponenten ist nicht möglich, ebenso wenig wie das Editieren der Daten).

- Das *SQLClientDataSet*, wie es in den Beispielen dieses Buchs verwendet wird, enthält intern ebenfalls eine unidirektionale Datenmenge sowie eine Provider-Komponente, über die der Zugriff auf die unidirektionale Datenmenge erfolgt. Wenn Ihnen die voreingestellte Weise, wie *SQLClientDataSet* mit diesen internen Komponenten umgeht, nicht ausreicht, können Sie das *SQLClientDataSet* selbst in drei Komponenten aufspalten: ein normales *ClientDataSet* (ohne »SQL«), einen Provider (*DataSetProvider* auf der Komponentenpalette *Datenzugriff*) und eine unidirektionale Datenmenge.

- Diese Aufspaltung ist insbesondere dann notwendig, wenn Sie eine mehrschichtige Anwendung schreiben. Die *TDataSetProvider*-Komponente kann nämlich in eine Bibliothek ausgelagert werden, die dann auf einem externen Rechner ausgeführt wird und dort über eine unidirektionale Datenmenge und eine *SQLConnection* auf einen dritten Rechner, den Datenbank-Server, zugreifen. Auf der Seite der Anwendung (auf dem ersten Rechner bzw. der ersten Schicht) befindet sich dann lediglich noch das ClientDataSet, das über eine spezielle Verbindungskomponente auf den externen Rechner zugreifen kann.

Auch wenn Sie zwischen diesen Variationen wechseln wollen, genügt es prinzipiell, die entsprechenden Datenzugriffskomponenten umzustellen und die DataSource-Komponente mit den unter Umständen neu eingeführten Zugriffskomponenten zu verbinden. Die visuellen Komponenten, an die die DataSource-Komponente ihre Daten weiterleitet, bekommen von dieser Änderung nichts mit.

Nachdem Ihnen dieses Kapitel nun die Grundbegriffe und -Konzepte vorgestellt hat, beginnen wir im nächsten Kapitel mit der Praxis und bauen zunächst ein ganz einfaches Datenbankformular auf.

7.2 Von der Tabelle zum Browser

Delphi lässt Ihnen bei der Datenbank-Programmierung, wie bei anderen Anwendungstypen auch, einen großen Spielraum darin, inwieweit Sie Ihre Anwendung bereits zur Entwurfszeit festlegen. So können Sie mit dem Felder-Editor und dem Objektinspektor Datenbankanwendungen konstruieren, die schon mit nur wenigen Zeilen Code sinnvoll einsetzbar sind; und mit Hilfe des Datenbankformular-Experten können Sie sogar den Entwurf der Formulare teilweise automatisieren. Das andere Extrem wäre, dass Sie im Formular außer ein paar Steuerelementen keinerlei Datenbankfunktionalität unterbringen und zur Laufzeit intensiv die Datenbankschnittstelle der VCL verwenden oder sogar direkt auf die BDE zugreifen.

In diesem Kapitel werden wir beim erstgenannten Extrem, also bei Anwendungen ohne selbst geschriebenen Code, beginnen, bevor nach und nach verschiedene Möglichkeiten untersucht werden, wie Sie die VCL zur Laufzeit nutzen können, um Ihre Datenbankanwendungen individuell zu gestalten.

7.2.1 Das grundlegende Datenbankformular

Die Datenbankkomponenten von Delphi sind so aufeinander abgestimmt, dass Sie nur einige davon zusammensetzen müssen, um eine komplette Datenbankanwendung zu erhalten, die sich auf eine vordefinierte Weise verhält und bereits eine komplette Datenbanktabelle anzeigt, durch die Sie blättern und die Sie editieren können. Um eine solche Anwendung zu erhalten, müssen Sie noch keinen Code schreiben, sondern lediglich einige wenige Komponenten einfügen und ein paar Properties einstellen.

Neben dieser tabellenartigen Anwendung können Sie einen zweiten Grundtyp von Datenbankanwendungen ebenfalls in kürzester Zeit generieren: Anwendungen, die mit einer Eingabemaske arbeiten. Eine solche Maske für eine vorgegebene Datenbank können Sie durch den Datenbankformular-Experten erzeugen lassen (siehe Kapitel 7.2.4).

Diese Anwendungsgerüste stellen bereits eine große Menge von Standard-Verhaltensweisen zur Verfügung und bilden damit eine solide Grundlage für die meisten Datenbankanwendungen. Da Sie nur für Ergänzungen und Änderungen des Standardverhaltens weitere Komponenten einfügen und neuen Code schreiben müssen, bleibt Ihnen viel Routinearbeit erspart.

Ein kleiner Rest an Routinearbeit ist jedoch unvermeidlich: das Anlegen der beschriebenen Anwendungsgerüste. Beide bauen auf dem im Folgenden beschriebenen Konzept auf.

Hinweis: Sie können die im Folgenden beschriebene Daten-Pipeline teilweise auch grafisch über das Objekthierarchie-Fenster (bzw. in Delphi 5 in der Baumansicht des Datenmodul-Designers) aufbauen. Die erste Daten-Pipeline ist jedoch so einfach, dass sich dies noch nicht besonders lohnen würde, daher werden diese weiter führenden Funktionen erst in Kapitel 7.2.8 vorgestellt.

Die grundlegende Komponenten-Pipeline

Noch grundlegender als die beiden oben beschriebenen Anwendungsgerüste ist der Datenfluss zwischen drei elementaren Komponenten: dem *DataSet*, der *DataSource* und den visuellen Datensteuerelementen. Dieses Dreigespann von Komponenten dürfte in jeder interaktiven Delphi-Datenbankanwendung zu finden sein.

Der Weg der Daten von der Datei zum Bildschirm hat – einschließlich Datei und Bildschirm – vier Stationen:

- Datenbank-Server: Die erste Station lässt sich für den Entwickler grob als »außerhalb der Delphi-Anwendung« beschreiben, dies kann eine lokale Datei oder die Datenbank auf einem Netzwerkserver sein, bei dreischichtigen Anwendungen gibt es zwischen dem Datenbank-Server und dem PC, auf dem die Anwendung läuft, noch eine Zwischenschicht, was aber für die Anwendung selbst keine Rolle spielt.
- Datenzugriff: Die zweite interessante Station befindet sich innerhalb der Delphi-Anwendung und wird von den Datenzugriffskomponenten gebildet. Bei Verwendung der BDE genügen hier *TTable*- oder *TQuery*-Komponenten, optional ergänzt durch eine *TDataBase*-Komponente (siehe Kapitel 7.1.1). Im Falle von My-Base-Dateien kommen statt dessen *ClientDataSet*-Komponenten zum Einsatz (siehe Kapitel 7.1.2), bei Verwendung von dbExpress ist es eine Komponente des Typs *TSQLConnection*, kombiniert mit weiteren Komponenten (zum Datenzugriff mit dbExpress siehe Kapitel 7.1.6). In allen Fällen ist zumindest eine Komponente erforderlich, die eine von *TDataSet* abgeleitete Klasse hat (wie etwa *TTable* für eine komplette BDE-Tabelle, *TClientDataSet* für eine MyBase-Tabelle oder *TSQLTable* für das Ergebnis einer SQL-Abfrage über dbExpress).
- Datenquelle: Für die dritte Station der Daten stellt Delphi die Klasse *TDataSource* zur Verfügung. Der Name von *TDataSource* gibt diesen Komponententyp bereits als »Datenquelle« aus. Abgesehen davon, dass in dieser Daten-Pipeline ja jede Station die Datenquelle für die jeweils nächste Station darstellt, ist *TDataSource* eine besonders wichtige Datenquelle, denn sie trennt die visuellen Komponenten von den Datenzugriffskomponenten ab und macht die visuellen Komponenten dadurch vor Änderungen im Datenzugriff unabhängig (siehe auch Kapitel 7.1.4). *TDataSource* ist also die Datenquelle für die visuellen Komponenten.

- Diese visuellen Komponenten bilden die letzte Station: Sie zeigen die aus einer *TDataSource* stammenden Daten im Formular an. Zum größten Teil handelt es sich um Spezialversionen der Standardkomponenten. So zeigt beispielsweise *TDBEdit* ein bestimmtes Feld aus dem aktuellen Datensatz der zugeordneten *DataSource* an und erlaubt dem Anwender, seinen Inhalt zu editieren.

Manueller Aufbau der Daten-Pipeline

Die hier beschriebene Daten-Pipeline kann vom Datenbankformular-Experten zwar automatisch hergestellt werden; um ihren Aufbau besser verstehen zu können, sollten Sie diese Pipeline jedoch auch einmal manuell aufbauen. Mit den folgenden Schritten können Sie in effektiver Weise eine Anwendung wie in Abbildung 7.6 zusammenbauen, die eine Tabelle komplett in einer scrollbaren *TDBGrid*-Komponente anzeigt:

- Fügen Sie eine *TTable*-Komponente von der Seite *BDE* (bis Delphi 5: *Datenzugriff*) in das Formular ein. Setzen Sie die Properties *DatabaseName* und *TableName*: In *DatabaseName* legen Sie die Datenbank fest (also den Verzeichnis- oder Aliasnamen), in *TableName* geben Sie die Tabelle der Datenbank an (im einfachsten Fall also den Dateinamen, siehe Kapitel 7.1.1). Wählen Sie zum Test eine der mit Delphi mitgelieferten Tabellen, indem Sie *DatabaseName* auf das vorinstallierte Alias *DBDEMOS* setzen und dann in *TableName* eine Tabelle aus der Liste auswählen.

- Als Nächstes fügen Sie eine Komponente des Typs *TDataSource* in das Formular ein. Das einzige Property, das Sie hier verändern müssen, ist *DataSet*, es gibt eine *DataSet*-Komponente an, aus der die *DataSource* ihre Daten bezieht. Geben Sie hier die im ersten Schritt eingefügte *TTable*-Komponente an, bzw. wählen Sie sie aus der aufklappbaren Liste aus. Solange Sie nur eine *DataSet*-Komponente im Formular haben, genügt auch schon ein Doppelklick auf das Editierfeld des *DataSource*-Properties, um die Tabelle dort einzutragen.

- Als letzte Komponente folgt nun eine, die die Daten zur Laufzeit sichtbar macht: Am informativsten ist hier *TDBGrid*. Wechseln Sie in der Komponentenpalette auf die Seite *Datensteuerung*, fügen Sie eine *TDBGrid*-Komponente in das Formular ein und setzen Sie ihr *DataSource*-Property per Doppelklick auf die im letzten Schritt eingefügte *DataSource*-Komponente.

  Das Formular in Abbildung 7.6 enthält zusätzlich zwei *TDBEdit*-Komponenten, deren *DataSource*-Property gleichermaßen angepasst wurde. Bei solchen Komponenten, die nur ein Feld der Tabelle anzeigen, müssen Sie außerdem das Property *DataField* setzen, siehe hierzu Kapitel 7.2.2.

- Wenn Sie vorerst keine Eingaben im Quelltexteditor machen und die Daten der Tabelle bereits zur Entwurfszeit sehen wollen, nehmen Sie die Pipeline nun in Betrieb, indem Sie das *Active*-Property der *TTable*-Komponente auf *True* setzen. Dies können Sie auch schon im ersten Schritt vor dem Einfügen der *TDBGrid*-Kompo-

nente tun, allerdings würden Sie dann den interessanten Einschalt-Effekt, durch den ein bisher leeres *TDBGrid* mit Daten gefüllt wird, verpassen. Zur Entwurfszeit können Sie bereits über die Bildlaufleisten durch die Tabelle blättern.

Wenn Sie nicht die BDE für den Datenzugriff verwenden wollen, wählen Sie im ersten Schritt statt *TTable* eine oder mehrere andere Datenzugriffskomponenten. In Kapitel 7.1 wurden bereits verschiedene alternative Datenzugriffsarten vorgestellt. Um etwa ein *ClientDataSet* zu verwenden und mit einer der von Borland mitgelieferten Demo-Tabellen zu testen, fügen Sie statt der *TTable* ein *TClientDataSet* in das Formular ein und geben in seinem *FileName*-Property den Pfad zur gewünschten MyBase-Datei an. Im Gegensatz zu *TTable* scheidet die Verwendung eines Alias-Namens hier allerdings aus, so dass Sie den Pfad direkt angeben müssen, etwa `...\Programme\Gemeinsame Dateien\Borland Shared\Data`.

Hinweis: Wenn Sie den Inhalt einer MyBase-Datei zur Entwurfszeit in ein ClientDataSet laden (indem Sie das *Active*-Property auf *True* setzen), gehört die gesamte Datenmenge zu den Eigenschaften der Komponente, die in der Formulardatei gespeichert werden, wodurch diese plötzlich auf eine nahezu beliebige Größe anwachsen kann. Dies können Sie vermeiden, indem Sie vor dem Speichern das *Active*-Property des ClientDataSets wieder abschalten.

Wenn Sie diese Anwendung starten, haben Sie die Möglichkeit, auch über die Tastatur durch die Datentabelle zu blättern, die Spaltenbreiten zu ändern und sogar, die Daten zu editieren. Ein wichtiges Ergänzungselement für solch ein grundlegendes Formular wäre eine Komponente der Klasse *TDBNavigator*, sie wird in Kapitel 7.2.3 besprochen.

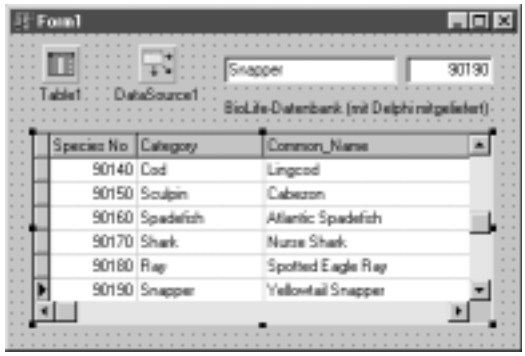


Abbildung 7.6: Ein Datenbankformular noch ohne selbst geschriebenen Quelltext (zur Anzeige des Tabelleninhalts wurde das Active-Property der Datenmenge eingeschaltet)

Das Konzept der Datenmodule

Ein Datenmodul ist ein Container für nicht-visuelle Komponenten, die in mehreren Formularen verwendet werden können. Zur Laufzeit ist ein Datenmodul in jedem Fall unsichtbar. Zur Entwurfszeit wird es als formularähnliches Fenster mit weißem Hintergrund dargestellt; speziell in Delphi 5 sind in dieses Fenster auch eine Baumansicht und ein Datenmodul-Designer integriert (in Delphi 6 sind diese beiden Funktionen als Objekthierarchie-Fenster und Diagramm-Ansicht des Code-Editors verallgemeinert und ausgelagert worden).

Um ein Datenmodul manuell zu erzeugen, wählen Sie im DATEI | NEU-Dialog das DATENMODUL. Danach können Sie alle nicht-visuellen Komponenten (Tabellen, Timer, Bilderlisten etc.) im Fenster des Moduls platzieren wie in einem Formular. Im Objektinspektor können Sie die Datenmodul-Properties *Tag* und *Name* sowie die Ereignisse *OnCreate* und *OnDestroy* bearbeiten.

Der große Vorteil der Datenmodule kommt ins Spiel, wenn Sie die in dem Modul aufbewahrten Komponenten in den verschiedenen Formularen Ihrer Anwendung verwenden wollen, z.B. um aus mehreren Formularen auf dieselbe Datenbank zuzugreifen. Statt jedes Formular, das diese Datenbank verwendet, mit einem eigenen Satz von Datenzugriffs-Komponenten zu versehen, brauchen Sie alle Datenzugriffskomponenten nur einmal in einem Datenmodul unterzubringen. Sie können diese Komponenten dann in allen Formularen Ihres Projekts benutzen.

Datenmodule verwenden

Um den Inhalt eines Datenmoduls in einem anderen Formular zu benutzen, führen Sie die folgenden Schritte aus:

- Holen Sie das Formular in den Vordergrund und wählen Sie den Menüpunkt DATEI | UNIT VERWENDEN..., wählen Sie den Namen der Datenmodul-Unit aus dem folgenden Dialogfenster aus und drücken Sie *OK*. Auf diese Weise binden Sie die Unit des Datenmoduls in der Unit des Formulars ein, so dass das Formular die Komponenten des Datenmoduls »sehen« kann.

  Dieser Schritt ist nur einmal notwendig, und zwar bevor Sie die erste Komponente des Datenmoduls im Formular benutzen. Wenn Sie ein Datenmodul mit dem Datenbankformular-Experten erstellen, so verbindet dieser die beiden neu erzeugten Units automatisch miteinander.

- Nun können Sie die Komponenten des Moduls wie gewohnt im Formular benutzen. Um beispielsweise eine *DataSource*-Komponente eines Datenmoduls mit einem *DBGrid* eines Formulars zu verbinden, wählen Sie die *DataSource* einfach aus der aufklappbaren Liste des Properties *TDBGrid.DataSource* aus. In dieser Liste ist dem Namen der Komponente noch der Name des Datenmoduls vorangestellt, so wie es die übliche Pascal-Syntax verlangt (z.B. *DataModule.DataSource*).

Ansonsten ändert sich durch Datenmodule nichts in der Art, wie die Komponenten verwendet werden. Insbesondere wird bei Datenbankanwendungen der Aufbau der Daten-Pipeline nicht beeinflusst.

Zur Laufzeit ist ein Datenmodul für den Benutzer unsichtbar, nicht aber für Ihren Programmcode: Dieser kann – abgesehen von der direkten Nennung eines Moduls bei seinem Namen – die Module über Properties des *Screen*-Objekts erkennen (*DataModules* und *DataModuleCount*, zu *TScreen* siehe Kapitel 3.2.4).

Formularverknüpfung

Die Verwendung von Komponenten eines anderen Formulars durch *Formularverknüpfung* ist eine grundlegende Funktion, die unabhängig von den oben beschriebenen Datenmodulen ist. Sie steht Ihnen nicht nur zwischen Datenmodulen und Formularen, sondern auch zwischen mehreren Formularen zur Verfügung und wird allgemein als Formularverknüpfung bezeichnet. Erst das Vorhandensein der Formularverknüpfung macht die Datenmodule möglich. Diese sind zwar nicht unbedingt notwendig, da Sie gemeinsam benutzte Komponenten auch in einem beliebig ausgewählten Formular unterbringen könnten, sie fördern aber die Übersichtlichkeit eines Projekts erheblich.

7.2.2 Datensensitive Steuerelemente

TDBGrid ist nur eine von vielen datensensitiven Komponenten auf der Seite DATENSTEUERUNG. Die meisten anderen Komponenten unterscheiden sich zuerst einmal darin, dass sie weder mehrere Datensätze noch mehrere Felder eines Datensatzes anzeigen, sondern nur genau ein Feld des gerade aktuellen Datensatzes (das ist der Datensatz, der in der *TDBGrid*-Komponente selektiert ist, falls eine solche vorhanden ist). Wie schon in Kapitel 1.4.1 erwähnt, bedeutet die Bezeichnung *Datensteuerung* eher *Datensteuerelemente*, sollte also nicht mit dem Begriff *Datenzugriff* verwechselt werden.

Diese auf ein einzelnes Feld spezialisierten Datensteuerelemente geben Ihnen zwei wichtige Möglichkeiten:

- Sie können damit Masken aufbauen, die eine komfortablere Dateneingabe erlauben als eine *TDBGrid*-Tabelle. Mit dem Datenbankformular-Experten können Sie sogar automatisch Masken erstellen lassen, die alle Spalten einer Tabelle abdecken.
- Darüber hinaus gibt es einige Feldtypen, die in einem *TDBGrid* nicht dargestellt werden können, und zwar sind das die *Binary Large Objects* (BLOBs), zu denen im Wesentlichen Bilder und Texte variabler Länge gehören. Für diese Feldtypen sind die Komponentenklassen *TDBMemo* und *TDBImage* zuständig.

Verbindung zum Feld

Sie bauen eine solche Komponente fast genauso in das Formular ein wie das *TDBGrid*, erforderlich ist zuerst einmal das Setzen des Properties *DataSource*, um die Herkunft der Daten aus der »Pipeline« anzugeben. In einem zusätzlichen Schritt müssen Sie nun noch das Feld auswählen, das angezeigt werden soll. Hierzu finden Sie im Objektinspektor beim Property *DataField* wieder eine Liste aller zur Auswahl stehenden Felder, die über die von Ihnen gewählte *DataSource* erreichbar sind.

In einem Formular mit einer *TDBGrid*-Komponente, die über eine *DataSource* mit einer Tabelle verbunden ist, können Sie die grundsätzliche Funktionsweise der datensensitiven Steuerelemente einfach testen. Ergänzen Sie das Formular wie in Abbildung 7.6 beispielsweise um eine *TDBEdit*-Komponente, setzen Sie deren Property *DataSource* auf denselben Wert wie die *DataSource* der *TDBGrid*-Tabelle und wählen Sie ein Feld für *DataField* aus. Wenn Sie die Anwendung starten und im *DBGrid* einen anderen Datensatz auswählen, wechselt auch der Inhalt der *DBEdit*-Komponente, wie im folgenden Abschnitt beschrieben.

Automatische Navigation

Der Inhalt der Datensteuerelemente ist immer synchron zum aktuellen Datensatz in einer *TDBGrid*-Tabelle. Wenn Sie den Feldinhalt im Datensteuerelement editiert haben und zur Tabelle wechseln, wird die Änderung automatisch gespeichert und in das *DBGrid* übernommen.

Die Tatsache, dass der Inhalt des Editierfelds und die aktuelle Position im DBGrid immer übereinstimmen, beruht nicht auf einer geheimen Verbindung zwischen Editierfeld und DBGrid, sondern darauf, dass beide von der gleichen Datenbank-Tabellen (allgemein: von der gleichen Datenmenge) abhängen. Wenn Sie sich in einem DBGrid bewegen, ändert dieses die aktuelle Position in der Datenmenge, woraufhin auch alle anderen über *TDataSource*-Komponenten an diese Datenmenge angeschlossenen Datensteuerelemente angepasst werden.

Ein Datensteuerelement, das nur ein einzelnes Feld anzeigt, kann also nicht aus eigener Kraft zu einem anderen Datensatz wechseln, sondern benötigt die Unterstützung durch eine andere Komponente, mit deren Hilfe der Anwender durch die Tabelle blättern kann. Neben einer *TDBGrid*-Komponente kommt hier als einfachere Möglichkeit die Komponente *TDBNavigator* in Betracht, die in Kapitel 7.2.3 besprochen wird.

Verbindung zu BLOB-Feldern

Felder, die sich nicht innerhalb einer *TDBGrid*-Tabelle darstellen lassen, können Sie nur mit Hilfe eines eigenständigen Datensteuerelements darstellen, z.B. mit der Grafikkomponente *TDBImage*. Auf der Basis eines grundlegenden Datenbankformulars wie in Abbildung 7.6 genügen wenige Schritte, um diese Komponente auszuprobieren:

- Verbinden Sie eine Datenmengen-Komponente (*TTable*, *TClientDataSet*) mit einer der mit Delphi mitgelieferten Beispieldatenbanken, die auch Bilddaten enthalten (*Animals*, *Biolife* oder *Clients*).
- Setzen Sie eine *DBImage*-Komponente in das Formular.
- Setzen Sie deren *DataSource*-Property auf die *DataSource* Ihres Formulars.
- Setzen Sie schließlich das Property *DataField* auf den Namen des Grafikfelds (in den drei oben genannten Tabellen heißen diese Felder *BMP*, *Graphic* und *IMAGE*).

Diese drei Schritte können Sie genauso mit einer Memo-Komponente *TDBMemo* für Datenbanken mit einem Memo-Feld durchführen. (Der Browser aus Kapitel 7.3.4 verwendet eine *TDBImage*- und eine *TDBMemo*-Komponente, um diese zur Laufzeit dynamisch an eventuell passende Tabellenfelder zu binden.)

Steuerelement-Komponenten für Datenfelder

Die folgende Tabelle listet alle in Delphi zur Verfügung stehenden Datensteuerelement-Komponenten auf. Die weniger komplexen sind jeweils mit einer Komponente der Standardseite verwandt (»verschwistert«), die bis auf das »DB« den gleichen Namen hat.

Komponente	**Aufgabe**
TDBNavigator	stellt die in Kapitel 7.2.3 beschriebenen Bedienfunktionen zur Navigation durch die Tabelle und die Einstellung bestimmter Modi über eine Schalterleiste zur Verfügung.
TDBText	zeigt ein Feld des aktuellen Datensatzes als Text an (ist wie *TLabel* abgeleitet von *TCustomLabel*).
TDBEdit	erlaubt zusätzlich zu *TDBText* das Editieren des angezeigten Textes.
TDBCheckBox	zeigt den Wert von booleschen Feldern des aktuellen Datensatzes an (editierbar).
TDBListBox, TDBRadioGroup	bieten eine Auswahl an möglichen Werten für ein einzelnes Feld des aktuellen Datensatzes. Der Feldinhalt kann nur durch Auswahl eines der vorgegebenen Werte geändert werden.
TDBComboBox	lässt durch ein zusätzliches Editierfeld die Beschränkung der *TDBListBox* fallen, dass keine nicht vorgegebenen Werte eingegeben werden können.
TDBGrid	zeigt mehrere Datensätzen komplett an, wobei es die einzelnen Felder als Text darstellt, und verfügt auch über eine Editiermöglichkeit.
TDBMemo, TDBRichEdit	zeigen den Inhalt eines BLOB-Feldes als Text an (editierbar), wobei *TDBRichEdit* auch RTF-Formatierungen berücksichtigt, aber wie die *TRichEdit*-Komponente keine Benutzerschnittstelle zum Editieren der Formatierungen anbietet (siehe Kapitel 3.6.1).

Komponente	Aufgabe
TDBImage	zeigt ein Feld des aktuellen Datensatzes mit Bilddaten an und erlaubt über Zwischenablageoperationen (Ausschneiden und Einfügen) sogar in geringem Umfang das Editieren des Bildes.
TDBLookupListBox	funktioniert wie *TDBListBox*, die Listeneinträge sind jedoch nicht fest vorgegeben, sondern werden von der Komponente zur Laufzeit des Programms aus einer anderen Tabelle ausgelesen.
TDBLookupComboBox	ergänzt die von *TDBLookupListBox* bereitgestellte Funktionalität durch das Editierfeld einer *TComboBox*.

Außerdem gibt es noch eine besondere Form der Tabellenkomponente: Auf der Oberfläche einer *TDBCtrlGrid*-Komponente legen Sie zur Entwurfszeit eine Maske aus anderen Datensteuerelementen an, die Sie über das *DataField*-Property mit einer Tabellenspalte verknüpfen. Zur Laufzeit dupliziert das *DBCtrlGrid* diese Maske dann in jedes Feld der Tabelle, in dem jeweils ein eigener Datensatz angezeigt wird.

7.2.3 Elementare Funktionen

Die Komponente *TDBNavigator* eignet sich sehr gut zur weiteren Einführung in die Datenbankprogrammierung, denn sie stellt dem Anwender elementare Funktionen, mit denen auch der Programmierer arbeitet, über Schalter zur Verfügung (in der nächsten Abbildung 7.7 ist ein solcher Navigator zu sehen). Diese Schalter betreffen die Navigation durch die Tabelle und das Editieren derselben. Selbst wenn Sie meinen, dass Sie diese Komponente niemals verwenden werden, können Sie von ihr viel über die Funktionsweise der Komponente *TTable* erfahren, denn jeder Schalter entspricht einer ihrer wichtigen Methoden.

TDBNavigator

Die Aufgabe von *TDBNavigator* ist es lediglich, die üblichen Bedienfunktionen für eine Datenbank in einer Komponente zusammenzufassen und die einzelnen Schalter zu verwalten (wenn Sie das *ShowHint*-Property einschalten, zeigt *TDBNavigator* sogar Hinweise zu diesen an). Mit der internen Funktion der Datenbankanwendung hat diese Komponente nichts zu tun. Für jeden Schalter ruft sie nämlich lediglich eine gleichbedeutende *TDataSet*-Methode auf, nur beim Löschen führt sie eine zusätzliche Sicherheitsabfrage durch, wenn das Property *ConfirmDelete* eingeschaltet ist (wie es die Voreinstellung ist).

Um eine Basis zum praktischen Experimentieren zu haben, erweitern Sie eine Beispielanwendung (*FileDB* oder *DBBrowse*) oder ein grundlegendes Datenbankformular um eine *TDBNavigator*-Komponente und setzen ihr *DataSource*-Property auf die Data-

Source-Komponente für die gewünschte Datenmenge. Ohne eine Zeile Code sollte die Anwendung sofort lauffähig sein.

Grundkonzepte der Datenbankprogrammierung

Für die Navigationsschalter spielen der Positionszeiger und der Modus, in dem sich die Tabelle befindet, eine große Rolle. Der Positionszeiger (auch *Cursor* genannt) weist einen Datensatz aus der Tabelle als *aktuellen Datensatz* aus, der Modus unterscheidet (unter anderem) zwischen dem Editieren und dem Blättern in der Tabelle. Auf den ersten Blick sind diese beiden Eigenschaften zwar nur für die Bedienung der Datenbankanwendung wichtig, denn der Benutzer kann immer nur einen Datensatz gleichzeitig editieren und muss zwischen Editieren und Blättern in der Tabelle wechseln. Tatsächlich müssen Sie sich aber auch bei der Programmierung an diesen aktuellen Datensatz und an den gerade eingestellten Modus halten. Beides sind grundlegende Eigenschaften der Komponente *TDataSet* und damit aller speziellen Datenmengen-Komponenten wie *TTable* und *TClientDataSet*.

Aktueller Datensatz und Modus beeinflussen die Programmierschnittstelle der Datenbankkomponenten entscheidend: Die einzelnen Datensätze einer Tabelle sind nicht direkt und beliebig über ein indiziertes Array-Property ansprechbar (so können Sie beispielsweise nicht irgendeinen Datensatz mit einer Anweisung wie *Table.Datensatz[100].Name := 'niemand'* setzen). Statt dessen können Sie nur den Datensatz an der aktuellen Position erreichen. Um auf beliebige Datensätze zugreifen zu können, müssen Sie für jeden Datensatz zuerst die aktuelle Position verändern. Dazu geben Sie jedoch keine Datensatznummer ein, sondern bewegen sich relativ zum aktuellen Datensatz, wie es im Folgenden beschrieben wird, oder suchen über einen Schlüssel gezielt einen bestimmten Datensatz auf (siehe Kapitel 7.4.3).

Wenn Sie bereits Erfahrung mit Datenbanken oder deren Programmierung haben, wird Ihnen dies wahrscheinlich völlig normal vorkommen, da die Position der Datensätze sich innerhalb einer Datenbank ständig ändern kann und feste Indizes daher nicht sehr hilfreich wären. Diese Art der Programmierung bildet jedoch einen Gegensatz zur einfachen Property-Programmierung in Delphi, wo sich fast alle gleichförmigen Datenmengen in ein Array-Property fassen lassen, wie es die VCL in zahlreichen Standard-Komponenten demonstriert.

Hinweis: Die ClientDataSet-Klassen verfügen über das Property *RecNo*, über das Sie den aktuellen Datensatz auch über seinen Index setzen können, was jedoch normalerweise nicht zu empfehlen ist – schon deshalb, weil sich dieser Index durch Einfügen neuer Datensätze verändern kann.

Die aktuelle Position

Es ist ein wichtiger Vorteil, dass die aktuelle Position nicht etwa eine Eigenschaft von *TDBGrid*, sondern von der Datenmenge selbst ist. Wenn sich also statt oder zusätzlich zu *TDBGrid* andere Komponenten zur Anzeige einzelner Felder im Formular befinden, so müssen Sie nicht jede Komponente einzeln auf die aktuelle Position setzen, sondern brauchen die Position nur einmal, und zwar in der *DataSet*-Komponente, zu ändern. Die VCL passt dann alle über die Daten-Pipeline angeschlossenen Datensteuerelemente an (sofern Sie diese Aktualisierung nicht gezielt unterbinden, siehe Kapitel 7.3.5, Abschnitt *Arbeiten in der Tabelle*).

Die einfachsten Arten, die aktuelle Position anzupassen, sind sowohl über eine Methode von *TDataSet* als auch über einen Schalter von *TDBNavigator* zugänglich und in der folgenden Tabelle zusammengefasst:

TDataSet-Methode	Positionierung	TDBNavigator-Schalter
First	zum ersten Datensatz des *DataSets*	nbFirst
Prior	einen Datensatz von der aktuellen Position rückwärts	nbPrior
Next	einen Datensatz weiter	nbNext
Last	zum letzten Datensatz des *DataSets*	nbLast

Die Reihenfolge in der obigen Tabelle entspricht den ersten vier Schaltern des *DBNavigators* von links nach rechts. Sie können im Property *TDBNavigator.VisibleButtons* diese und jeden anderen Schalter einzeln an- und ausschalten, indem Sie die in der Tabelle genannten Unterelemente von *VisibleButtons* (*nb...*) ändern.

Hinweis: Eng mit den obigen Methoden verknüpft sind auch die Properties *BOF* und *EOF*. Sie zeigen an, ob der Beginn (*BOF*) oder das Ende (*EOF*) der Datenmenge erreicht ist. Wenn Sie beispielsweise mit *Next* alle Elemente der Menge durchlaufen, erfahren Sie durch *EOF=True*, dass Sie das Ende erreicht haben (*EOF* gibt erst dann *True* zurück, wenn Sie sich schon im letzten Datensatz befinden *und* versucht haben, mit *Next* noch einen Schritt weiter zu gehen; *BOF* verhält sich analog dazu).

Funktionsweise des Editierens

Wenn Sie die Properties der Datenbankkomponenten bei ihren Voreinstellungen belassen, können Sie die Daten zur Laufzeit ohne Probleme editieren. Wenn Sie beispielsweise in einem *DBGrid* mit der Eingabe von Zeichen beginnen, wechselt das *DBGrid* automatisch in den Editiermodus. Auch alle anderen Datensteuerelemente sind editierbar. Wenn Sie zu einem anderen Datensatz wechseln, werden die Änderungen am

bisher aktuellen Datensatz automatisch gespeichert (bei Verwendung der BDE werden sie in der Datenbank gespeichert, bei Verwendung von dbExpress in den Änderungsspeicher geschrieben).

Sie haben eine Reihe von Möglichkeiten, auf das Editieren Einfluss zu nehmen – jede im bisherigen Beispielformular verwendete Komponente aus der Daten-Pipeline bietet hierzu ein bestimmtes Property:

- So können Sie in den editierbaren Datensteuerelementen das Editieren unterbinden, indem Sie das Property *ReadOnly* auf *True* setzen oder das Flag *dgEditing* im *Options*-Property der *TDBGrid*-Komponente löschen.
- Um die gesamte Tabelle effektiv gegen Veränderungen zu schützen, setzen Sie das *ReadOnly*-Property der *DataSet/Table*-Komponente auf *True*.
- Die Komponente *TDataSource* bietet schließlich das Property *AutoEdit*, das per Voreinstellung eingeschaltet ist, so dass Sie wie oben beschrieben alle Felder direkt editieren können. Wenn Sie *AutoEdit* auf *False* setzen, erreichen Sie, dass der Benutzer zuerst explizit in den Editiermodus wechseln muss, indem er beispielsweise den Editieren-Schalter des *DBNavigators* drückt. Wenn er nach dem Editieren zu einem anderen Datensatz wechselt, werden die Änderungen gespeichert, der Editiermodus aber automatisch wieder abgeschaltet.
- Die niedrigste Ebene, auf der Sie das Editieren einer bestimmten Tabellenspalte verhindern können, ist das *ReadOnly*-Property der Feldkomponenten. Solche sind in unseren aktuellen Experimental-Formular jedoch noch nicht verfügbar. Kapitel 7.3 wird Ihnen die Feldkomponenten im Detail vorstellen.

Hinweis: Hinter den Kulissen bestimmt nicht die Klasse *TDataSource*, dass das Editieren im nicht-automatischen Modus wie beschrieben abläuft, denn theoretisch können die Datensteuerelemente die Tabelle trotz *AutoEdit=False* in den Editiermodus setzen und verändern, genauso wie Sie das mit eigenem Code jederzeit tun können (sofern eben nicht die *DataSet*-Komponente per *ReadOnly* geschützt ist). Die Funktion des Properties *TDataSource.AutoEdit* ist also davon abhängig, dass sich alle Steuerelement-Komponenten daran halten und nur dann das Editieren zulassen, wenn der Editiermodus auf andere Weise aktiviert wird, beispielsweise über den *DBNavigator*-Schalter.

TDataSet.Edit

Unabhängig von allen Einstellungen in *TDataSource*-Komponenten und Datensteuerelementen können Sie die Tabelle jederzeit durch den Aufruf der Methode *Edit* in den Editiermodus versetzen. Dieser Schritt wirkt sich sofort auch auf die anderen mit der

Tabelle verknüpften Komponenten aus, *DBNavigator* zeigt den Editiermodus automatisch in seiner Schalterdarstellung an und die Datensteuerelemente, bei denen das *ReadOnly*-Property nicht *True* ist, lassen sich editieren.

Sie können also auch diesen Schalter des *DBNavigators* leicht durch einen eigenen Schalter ersetzen, in dessen *OnClick*-Methode Sie einfach die entsprechende *TDataSet*-Methode (hier *Edit*) aufrufen. In allen anderen Fällen werden Sie die Methode *Edit* wahrscheinlich nur aufrufen, um den aktuellen Datensatz durch weitere Programmanweisungen zu verändern. Wie das geschieht, lesen Sie in Kapitel 7.3.5.

Speichern, Verwerfen und Aktualisieren

Mit dem Editiermodus alleine ist es jedoch noch nicht getan, es kommt nun darauf an, wann die Änderungen in die Tabelle geschrieben werden. Es ist klar, dass die Datenbankdatei nicht jedes Mal, wenn der Benutzer ein einzelnes Zeichen verändert, aktualisiert werden kann.

Wie schon erwähnt sorgt die voreingestellte Automatik dafür, dass die Daten jedes Mal, wenn der Benutzer den Datensatz wechselt, in ein Änderungsprotokoll geschrieben bzw. bei Verwendung der BDE in der Datenbank gespeichert werden. Im *DBNavigator* hat der Anwender mit den Schaltern *Cancel* und *Post* zwei Möglichkeiten, diesen automatischen Ablauf zu ändern:

- Als Erstes sollte es natürlich die Möglichkeit geben, die Übernahme einer verunglückten Editieroperation in das Änderungsprotokoll zu verhindern und damit die Änderungen zu verwerfen. *TDataSet* stellt zu diesem Zweck die Methode *Cancel* zur Verfügung und im *DBNavigator* gibt es hierfür den Schalter mit dem großen »X« . In beiden Fällen werden die Steuerelemente automatisch mit den noch unveränderten Werten des Datensatzes neu geladen.
- Das Gegenteil der *Cancel*-Methode ist die Methode *Post*. Sie fordert die Datenmenge explizit auf, die Änderungen in das Änderungsprotokoll zu schreiben, so dass sie nicht mehr über *Cancel* rückgängig gemacht werden können (wohl aber noch über den Aufruf *CancelUpdates*, mit dem das Änderungsprotokoll verworfen wird, siehe Kapitel 7.2.7; oder über einen Aufruf von *UndoLastChange*, mit dem die letzte Änderung aus dem Protokoll rückgängig gemacht wird).

Der letzte Schalter von *TDBNavigator* entspricht der Methode *Refresh*. Sie liest alle in visuellen Komponenten angezeigten Datensätze aus der Tabelle neu ein. Dies ist z.B. dann sinnvoll, wenn sie zwischenzeitlich außerhalb der Delphi-Anwendung verändert worden sein könnten, beispielsweise von einer anderen Person, die auf dieselbe Datenbank zugreift.

Die in diesem Abschnitt besprochenen Funktionen sowie die *Edit*-Methode sind in der folgenden Tabelle zusammengefasst:

TDataSet-Methode	Funktion	TDBNavigator-Schalter
Edit	versetzt die Tabelle in den Editiermodus.	nbEdit
Post	schreibt die Änderungen in die Tabelle.	nbPost
Cancel	macht die Änderungen rückgängig, indem sie die Original-Daten erneut aus der Tabelle liest.	nbCancel
Refresh	liest alle Datensätze neu ein und ändert die Anzeige in den Datensteuerelementen (bereits editierte Feldinhalte werden vorher automatisch mit *Post* gespeichert).	nbRefresh

Löschen und Einfügen von Datensätzen

Die beiden noch nicht besprochenen Schalter des *DBNavigators* machen dem Benutzer die beiden Funktionen zugänglich, die noch fehlen, um zur Laufzeit theoretisch beliebige Änderungen der Tabelle durchführen zu können:

TDataSet-Methode	Funktion	TDBNavigator-Schalter
Insert	einen neuen Datensatz einfügen	nbInsert
Delete	den aktuellen Datensatz löschen	nbDelete

Wie schon erwähnt, macht ein *DBNavigator* vor dem Löschen eines Datensatzes eine Sicherheitsabfrage, während die Methode *TDataSet.Delete* sofort zur Tat schreitet.

Das Einfügen findet immer einen Datensatz vor der aktuellen Position statt. Nachdem der Benutzer (oder das Programm) die Daten eingegeben und bestätigt hat, wird die Position des Datensatzes natürlich an die Sortierreihenfolge angepasst.

7.2.4 Automatischer Aufbau der Daten-Pipeline

Delphi verfügt über einen Experten zur Erzeugung von BDE-Datenbankformularen, der neben einfachen Formularen auch Haupt-/Detailformulare und SQL-Abfragen generieren kann. Wir beschäftigen uns an dieser Stelle jedoch nur mit den einfacheren Möglichkeiten dieses Experten.

Jedes vom Datenbankformular-Experten erzeugte Formular enthält bereits Datensteuerelemente und Datenzugriffskomponenten (*TTable/TQuery* und *TDataSource*), deren Properties so eingestellt sind, dass sie eine Verbindung zu einer Tabelle herstellen. Das Einzige, was Sie noch tun müssen, ist, das *Active*-Property der *DataSet*-Komponente einzuschalten.

Neben dem Typ der *DataSet*-Komponente lässt Ihnen der Experte (abgesehen von der Haupt-/Detail-Option) die Wahl zwischen drei verschiedenen visuellen Darstellungen für das Formular, doch dazu später.

Grundsätzlicher Ablauf

Der Datenbankformular-Experte wird über DATENBANK | FORMULAR-EXPERTE... aufgerufen. Dieser Menüpunkt führt Sie in den visuellen Teil des Experten, ein Dialogfenster (Abbildung 7.7), in dem Sie auf mehreren Seiten verschiedene Einstellungen machen. Es handelt sich aus technischen Gründen (manche Seiten hängen von der Auswahl auf den vorherigen Seiten ab) jedoch nicht um einen normal blätterbaren Dialog, in dem Sie jede Seite direkt anwählen können, sondern Sie können über die beiden Schalter jeweils nur schrittweise eine Seite vor- und zurückblättern. Statt des Schalters zum Vorblättern finden Sie auf der abschließenden Seite einen Schalter zum Generieren des Formulars.

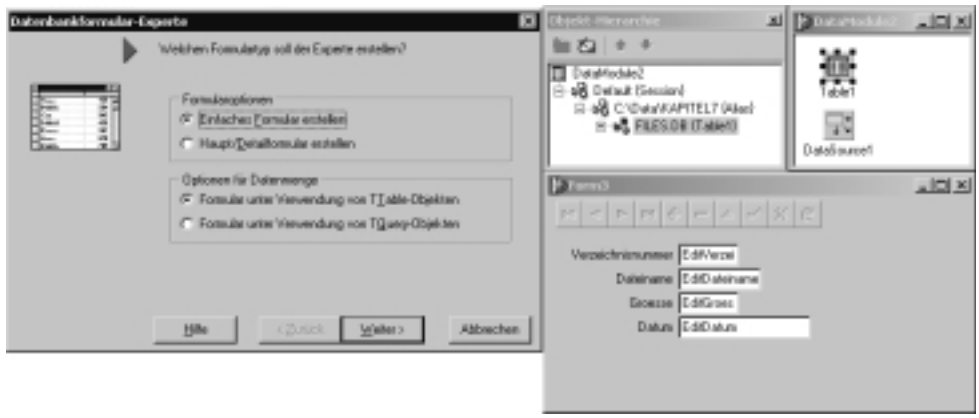


Abbildung 7.7: Der Datenbankformular-Experte und ein von ihm erzeugtes Formular-Datenmodul-Gespann

Wenn das Formular erst einmal erzeugt ist, macht der Experte seinem Namen jedoch keine Ehre mehr: Er leistet keinerlei Unterstützung für das von ihm erzeugte Formular. Das Einzige, was Sie von ihm erwarten können, wenn mit dem Formular etwas nicht stimmt, ist, dass er ein völlig neues Formular generiert.

Die einzelnen Seiten

Die einzelnen Seiten des Experten geben Ihnen für *TTable*-basierte einfache Datenbankformulare die folgenden Optionen:

- Auf der ersten Seite geben Sie an, ob Ihr Formular eine *TTable*- oder eine *TQuery*-Komponente verwenden soll und ob es sich um ein einfaches oder um ein Haupt-/

Detailformular handeln soll. Die folgende Beschreibung gilt für den Fall, dass Sie ein normales, mit *TTable* arbeitendes Formular erzeugen.

- Auf der zweiten Seite geben Sie die Tabelle an, deren Daten im Formular bearbeitet werden sollen. Sie können wie in der Datenbankoberfläche und anderen Datenbanktools auch aus eventuellen Alias-Namen auswählen, um die zu verwendende Datenbank anzugeben.
- Auf Seite 3 geht es darum, die Felder auszuwählen, die im Formular erscheinen sollen; auch die Reihenfolge der Felder können Sie hier festlegen.
- Auf der nächsten Seite fällt die Entscheidung über die optische Gestaltung des Formulars: Sie können die einzelnen Felder horizontal oder vertikal (wie in Abbildung 7.7) anordnen lassen, wobei der Experte bei der horizontalen Anordnung automatisch einen »Zeilenumbruch« durchführt, falls ein Feld sonst über den rechten Rand hinausragen würde. Eine dritte Möglichkeit ist die Verwendung der *TDBGrid*-Komponente, die alle Felder darstellt.
- Wenn Sie auf der zuletzt beschriebenen Seite die vertikale Anordnung ausgewählt haben, können Sie auf einer weiteren Seite wählen, ob der Experte die Beschriftung der einzelnen Felder links neben diesen oder über diesen anbringt.
- Auf der letzten Seite haben Sie schließlich die Möglichkeit, durch Abwahl des Markierungsschalters zu verhindern, dass der Experte das neue Formular automatisch zum Hauptformular des Projekts macht.
- Außerdem können Sie auf dieser letzten Seite verlangen, dass die Datenzugriffskomponenten nicht im generierten Formular, sondern in einem eigenen Datenmodul untergebracht werden, wie es beispielsweise in Abbildung 7.7 dargestellt ist.

7.2.5 Ein Browser für BDE- und MyBase-Tabellen

Zum Abschluss dieses Einführungskapitels geht es nun um den ersten Teil einer Datenbank-Browser-Anwendung, die demonstrieren wird, wie Sie auch zur Laufzeit flexibel auf verschiedene Tabellen und Datenbanken reagieren können. Abbildung 7.8 zeigt die fertige Version, die die beiden folgenden Aufgaben dynamisch erledigt:

- Aufbau einer Verbindung zu einer Datenbank nach Wahl des Benutzers,
- Angabe einer Liste aller Spalten der Tabelle und Auswahl von Grafik- und Memofeldern zur Darstellung in zwei einzelnen Datensteuerelementen.

Die letztgenannte Aufgabe kann erst in Kapitel 7.3.4 gelöst werden, hier bleibt die sehr einfache Aufgabe, Tabellen dynamisch zur Laufzeit anzubinden.

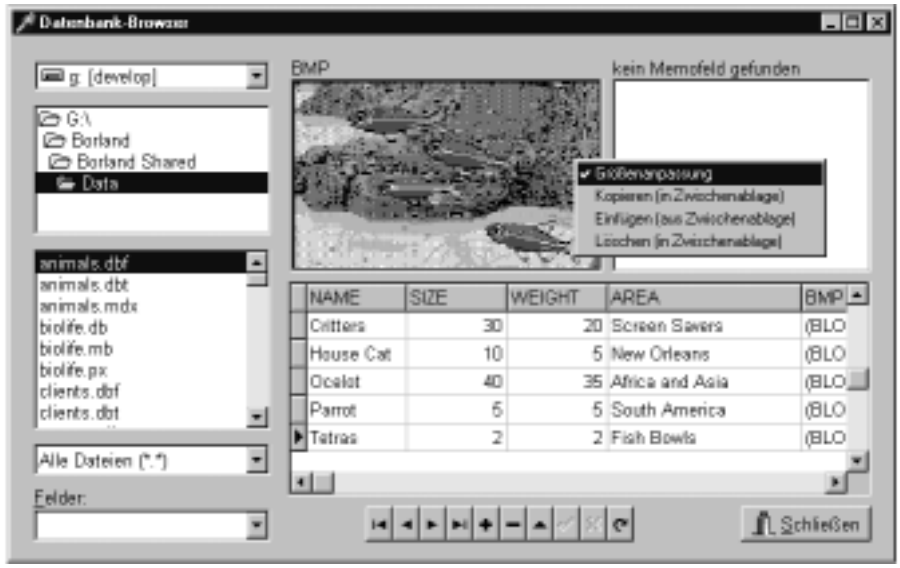


Abbildung 7.8: Der fertige Datenbank-Browser

Auf der CD finden Sie insgesamt drei Versionen dieses Browsers:

- *dbBrowse* verwendet *TTable* als Datenzugriffskomponente und kann auf Paradox- und dBase-Tabellen zugreifen.
- *cdsBrowser* arbeitet statt dessen mit einem *ClientDataSet* und kann daher nur MyBase-Tabellen anzeigen.
- *gdbBrowser* ist auf Interbase-Datenbanken ausgelegt, was einen geringfügig erweiterten Browsing-Ablauf erforderlich macht (siehe Kapitel 7.2.6); außerdem verwendet diese Version dbExpress als Datenzugriffsmethode, jedoch könnte auch für den Interbase-Browser die BDE verwendet werden.

Die ersten beiden Programmversionen ließen sich übrigens leicht vereinigen: Das Programm könnte ohne Weiteres sowohl eine *TTable*- als auch eine *TClientDataSet*-Komponente verwenden und zur Laufzeit zwischen beiden umschalten (über eine Änderung des Properties *DataSource.DataSet*). Um jedoch nicht die verschiedenen Datenzugriffsmethoden zu vermischen, wurde es hier bei getrennten Programmversionen belassen.

Browser-Steuerung durch fertige Listenkomponenten

Der Hauptteil der bisherigen Aufgabe besteht darin, dem Benutzer eine komfortable Browser-Schnittstelle zur Verfügung zu stellen. Diese steht im Gegensatz zu der für den Programmierer einfacheren Methode, die Auswahl einer neuen Datenbank über eine DATEI ÖFFNEN...-Dialogbox anzubieten. Ähnlich wie bei vielen Browsern und

Explorern, speziell auch bei Delphis Datenbank-Explorer, soll der Benutzer im linken Teil des Fensters auswählen, was er im rechten Teil sehen will. Wenn er eine Datenbank aus der Liste auswählt, soll diese sofort in den Datenbank-Steuerelementen dargestellt werden. Das Formular erhält zu diesem Zweck je ein Exemplar der Klassen *TFilterComboBox*, *TDriveComboBox*, *TDirectoryListBox* und *TFileListBox*.

Für das Formular ist nur der gewählte Eintrag der Dateiauswahlbox von Interesse, denn wenn dieser geändert wird, muss es eine neue Datenbankverbindung aufbauen. *DirectoryListBox* und *FilterListBox* geben Änderungen automatisch an die Dateiauswahlbox weiter (da ihre *FileListBox*-Properties auf diese Listbox gesetzt sind), so dass die Dateiliste automatisch aktualisiert wird. (Zu weiteren Details bezüglich der Dateiauswahlkomponenten siehe die Übersicht in Kapitel 1.9.3.)

Zur Laufzeit bleibt uns dann nur noch die Aufgabe, das *OnChange*-Event der Dateiwahlbox zu bearbeiten. In der zugehörigen Methode erhalten wir die volle Pfadangabe der ausgewählten Datei vom Property *FileName* der *DirectoryListBox*.

Komponenten des Formulars

Für diesen ersten Schritt benötigen wir neben den besprochenen Komponenten zur Auswahl einer Datei lediglich die grundlegenden Datenbankkomponenten: je ein Exemplar der Klassen *TTable* (BDE-Version des Programms) bzw. *TClientDataSet* (MyBase-Version), *TDataSource*, *TDBGrid* und *TDBNavigator*. Sie werden miteinander verknüpft, wie in den bisherigen Kapiteln besprochen. Über diese Verknüpfung hinausgehende Property-Einstellungen sind nicht erforderlich.

Verbindungswechsel

Der Datenbank-Browser soll also beim *OnChange*-Event der Dateiliste eine neue Datenbank-Verbindung aufnehmen. Dies geschieht auch im Quelltext durch Setzen der Tabellen-Properties *DatabaseName* und *TableName* (bei BDE-Verbindungen) bzw. durch Setzen von *FileName* (bei MyBase-Tabellen). Vorher muss der Browser jedoch die aktive Verbindung auflösen, indem er das *Active*-Property der Datenmenge auf *False* setzt. Sobald er die Verbindungsdaten der Datenbank gesetzt hat, kann er *Active* wieder einschalten. Die BDE-Version des Browsers enthält den folgenden Code:

```
procedure TForm1.FileListBox1Change(Sender: TObject);
begin
  if FileListBox1.FileName <> '' then begin
    Table1.Active := False; // alternativ: Table1.Close;
    Table1.DatabaseName := DirectoryListBox1.Directory;
    Table1.TableName := FileListBox1.FileName;
    Table1.Active := True;  // alternativ: Table1.Open;
  end;
end;
```

Diese wenigen Veränderungen sorgen schon dafür, dass die Datenbankkomponenten, insbesondere das *DBGrid*, entsprechend aktualisiert werden.

Die MyBase-Version des Browsers unterscheidet sich im Wesentlichen nur durch die Verwendung einer *TClientDataSet*-Komponente und dadurch, dass statt *DatabaseName* und *TableName* nun das *FileName*-Property gesetzt wird:

```
ClientDataSet1.TableName := FileListBox1.FileName;
```

Kapitel 7.3.4 wird diese extrem einfache Aktion stark erweitern, so dass auch Bildfelder der Datenbank automatisch angezeigt werden, wie in Abbildung 7.8 gezeigt.

7.2.6 Ein Browser für Interbase-Tabellen

Nun soll ein ähnlicher Browser wie im letzten Kapitel für BDE- und MyBase-Dateien auch für Interbase-Dateien entwickelt werden. Das fertige Projekt finden Sie auf der CD unter dem Namen *gdbBrowser*. Zur Abwechslung verwendet dieses Programm dbExpress als Datenzugriffsmethode, jedoch ließe sich der Interbase-Browser genauso gut auch mit der BDE implementieren. Das neue Browser-Formular unterscheidet sich wie folgt von der im letzten Kapitel besprochenen dBase / Paradox / MyBase-Version:

- Für den Datenzugriff wird nun ein Duo aus *TSQLClientDataSet* und *TSQLConnection* verwendet. Normalerweise müssten wir in der *SQLConnection* eine Verbindung auswählen oder einrichten (siehe Kapitel 7.1.6), jedoch wird dies im aktuellen Beispielprogramm programmgesteuert erledigt. Zur Entwurfszeit wird lediglich der Treibername der Komponente (Property *DriverName*) auf *INTERBASE* gesetzt. Für die Verbindung zwischen *SQLConnection* und *ClientDataSet* wird das Property *DBConnection* des ClientDataSets auf die *SQLConnection*-Komponente eingestellt. Außerdem enthält das Formular natürlich eine *DataSource*-Komponente, die mit dem ClientDataSet verbunden ist.
- Da Interbase-Dateien nicht nur eine einzelne Tabelle enthalten, sondern eine komplette Datenbank aus mehreren Tabellen, genügt nicht wie bisher die Auswahl einer Datei, um in der DBGrid-Komponente eine Tabelle darzustellen. Der Benutzer braucht auch eine Möglichkeit, eine Tabelle innerhalb der Datei auswählen zu können. Zu diesem Zweck finden Sie in Abbildung 7.9 unter der Dateiliste des neuen Formulars eine Kombobox, die eine Liste der in der aktuellen Datei verfügbaren Tabellen enthält.
- Eine weitere Modifikation hängt nicht mit der Umstellung auf dbExpress oder Interbase zusammen, sondern verleiht dem Programm lediglich ein moderneres Äußeres: Für die Auswahl der Datei werden nun statt der alten Komponenten *TDirectoryListBox* und *TFileListBox* die mit Delphi 6 eingeführten Beispielkomponenten *TShellTreeView* und *TShellListView* verwendet.

Abbildung 7.9: Der Browser für Interbase-Datenbankdateien

Das Programm hat zwei Benutzerereignisse zu verarbeiten: Die Selektierung eines anderen Dateinamens im ShellListView und die Auswahl eines anderen Tabellennamens aus der Kombobox. Wir beginnen mit dem Ereignis der Dateiliste.

Vor dem Wechsel der Tabellen und der Datenbank müssen die Verbindungen zur alten Datenbank / Tabelle deaktiviert werden; hierzu werden die Properties *TSQLClientDataSet.Active* und *TSQLConnection.Connected* abgeschaltet.

Die Definition der Verbindung geschieht im Programm durch Setzen des Objekts *TSQLConnection.Params*. Dieses enthält exakt die Werte, die Sie zur Entwurfszeit auch im Verbindungsdialog für vordefinierte Verbindungen einstellen können. So sprechen Sie etwa den Eintrag *DriverName* aus dem Dialog im Programm mit dem Ausdruck *Params.Values['DriverName']* an. Da der Treibername bereits zur Entwurfszeit auf *INTERBASE* festgelegt wurde, genügt es für das Programm, den Parameter *Database* auf die vom Benutzer gewählte Datei zu setzen. Der Dateiname selbst wird nun aus den Komponenten *ShellTreeView1* und *ShellListView1* ausgelesen (siehe auch Kapitel 8.4.1):

```
procedure TForm1.ShellListView1Change(Sender: TObject; Item: TListItem;
  Change: TItemChange);
begin
  if Item.Selected then begin
    SQLClientDataSet1.Active := False;
    SQLConnection1.Connected := False;
```

```
    SQLConnection1.Params.Values['Database'] :=
       IncludeTrailingPathDelimiter(
          ShellTreeView1.SelectedFolder.PathName)
       +ShellListView1.Folders[ShellListView1.Selected.Index].DisplayName;
    SQLConnection1.Connected := True;
```

Nun muss noch die Kombobox *cbTableNames* mit den Tabellennamen gefüllt werden. Die Verbindungskomponente bietet hierfür die Methode *GetTableNames* an, die bereits ein *TStrings*-Objekt wie das *Items*-Property der Kombobox als Parameter erwartet und es mit den Namen der Tabellen füllt. Vorher können wir der Verbindungskomponente über das Property *TableScope* noch mitteilen, welche Arten von Tabellen aufgelistet werden sollen. Die im Folgenden genannten Arten *tsTable or tsView* entsprechen der Voreinstellung, daher ist die Property-Zuweisung hier nicht wirklich notwendig (die weiteren möglichen Tabellenarten sind *tsSysTable* und *tsSynonym*):

```
  SQLConnection1.TableScope := [tsTable or tsView];
  cbTableNames.Items.Clear;
  SQLConnection1.GetTableNames(cbTableNames.Items);
  if cbTableNames.Items.Count > 0 then
    cbTableNames.ItemIndex := 0;
  cbTableNames.DroppedDown := True;
end;
```

Die letzten drei Zeilen der Methode dienen der bequemeren Handhabung der Kombobox: Falls mindestens eine Tabelle aufgelistet wurde, wird der erste Eintrag der Kombobox selektiert. Außerdem wird die Liste bereits aufgeklappt, denn das Programm verlangt in jedem Fall eine Auswahl der Benutzers, bevor es in seinem Datenbereich eine Tabelle anzeigt.

Die Auswahl des Benutzers in der Kombobox (*OnSelect*) ist das zweite zu bearbeitende Ereignis. Es ist der Bearbeitung des *OnSelectItem*-Ereignisses aus dem MyBase-Browser des letzten Kapitels ähnlich, da nur der Inhalt eines ClientDataSets neu festgelegt werden muss. An dieser Stelle handelt es sich jedoch um ein *TSQLClientDataSet*, das seine Daten nicht aus einer MyBase-Datei (*FileName*-Property) bezieht, sondern über eine SQL-Abfrage, die über die Properties *CommandType* und *CommandText* eingestellt werden will. *CommandType* wird hier auf *ctTable* gesetzt, damit einfach die gesamte Tabelle abgerufen wird. Die dafür notwendige SQL-Anweisung wird automatisch von der Komponente erzeugt; was für uns zu tun bleibt ist, die gewünschte Tabelle im *CommandText*-Property anzugeben:

```
procedure TForm1.cbTableNamesSelect(Sender: TObject);
begin
  SQLClientDataSet1.Active := False;
  SQLClientDataSet1.CommandType := ctTable;
  SQLClientDataSet1.CommandText :=
```

```
      cbTableNames.Items[cbTableNames.ItemIndex];
    SQLClientDataSet1.Active := True;
  end;
```

Außer diesen beiden Methoden enthält das komplette Beispielprogramm die gleichen Erweiterungen, die in Kapitel 7.3.4 auch dem Browser aus dem vorherigen Kapitel hinzugefügt werden (Anzeige von BLOB-Daten in zwei zusätzlichen Datenanzeigekomponenten und ein Zwischenablage-Popup-Menü für die *TDBImage*-Komponente).

Hinweis: Neben der von diesem Beispielprogramm gewählten »Browse-Strategie« gibt es noch viele andere Möglichkeiten, wie Sie durch vorhandene Datenbanken browsen können. Statt der Wahl eines Verzeichnisses und einer Interbase-Datei könnten Sie den Benutzer etwa aus den vordefinierten Verbindungen wählen lassen (wie Sie es in der Delphi-IDE selbst über den Verbindungsdialog tun können; bei einer Benutzerauswahl zur Laufzeit würden Sie dann das *ConnectionName*-Property der Verbindungskomponente auf den Namen der gewählten Verbindung einstellen). Dies hätte beispielsweise den Vorteil, dass Sie nicht nur Interbase-Datenbanken laden könnten wie das aktuelle Beispielprogramm.

7.2.7 Anwenden und Verwerfen von Updates mit dbExpress

Die in Kapitel 7.2.3 bereits vorgestellte *Post*-Methode einer Datenmenge dient als Bestätigung der Änderungen, die am aktuellen Datensatz durchgeführt wurden. *Post* kann durch den Benutzer explizit über einen Schalter einer *DBNavigator*-Komponente oder implizit durch Wechseln des aktuellen Datensatzes aufgerufen werden – oder auch durch das Programm über einen Aufruf der Methode *Post*.

Post bedeutet jedoch nicht, dass die Änderungen auch in der Datenbank gespeichert werden. Dies ist nur bei Verwendung der BDE, also z.B. bei der Komponente *TTable* der Fall, sofern dort keine Cached Updates verwendet werden. Bei dbExpress müssen Sie als Entwickler in jedem Fall gesonderte Vorkehrungen zur Speicherung der Daten treffen, wobei Sie im einfachsten Fall mit ein oder zwei Zeilen Code auskommen. Die folgende Beschreibung gilt nicht nur für dbExpress, sondern allgemein für alle ClientDataSets (*TClientDataSet* kann auch unabhängig von dbExpress, beispielsweise in BDE-Anwendungen, verwendet werden und existierte ja auch schon in früheren Delphi-Versionen, in denen dbExpress noch in weiter Ferne lag).

Abbildung 7.10 zeigt den Weg der Änderungen eines Datensatzes in einer ClientDatenmenge von der Entstehung bis zur Speicherung in der Datenbank. Die *Post*-Operation führt zunächst dazu, dass die Änderungen des Datensatzes im Änderungsprotokoll *Delta* der Datenmenge gespeichert werden. Dabei werden die gesamten Felder des alten Datensatzes, die geänderten Felder des neuen Datensatzes und die Art der Änderung gespeichert. Als Änderungsarten gibt es

- das Verändern eines bestehenden Datensatzes,
- das Löschen eines Datensatzes (hier gibt es natürlich keinen neuen Feldwerte, die gespeichert werden könnten) sowie
- das Einfügen eines neuen Datensatzes (hier entfällt die Speicherung der alten Feldinhalte).

Kapitel 7.5.4 wird noch einmal auf dieses Änderungsprotokoll zurückkommen und erläutern, wie Sie es in einem DBGrid anzeigen und wie Sie die drei erwähnten Änderungsarten im ClientDataSet sichtbar machen können.

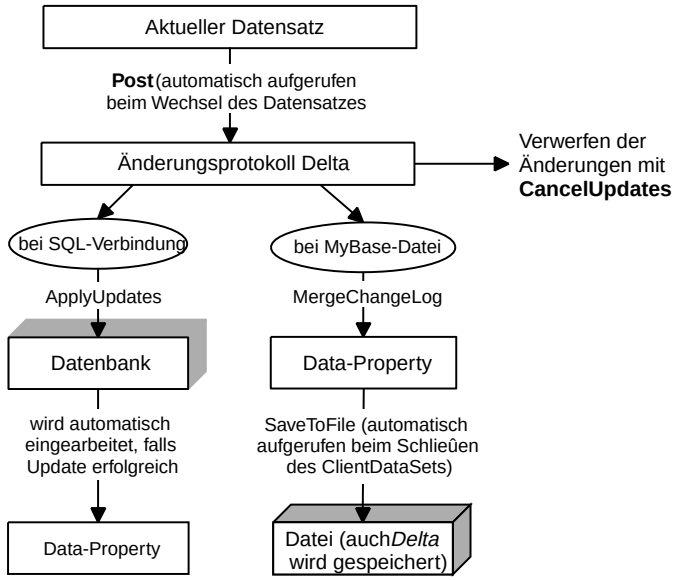


Abbildung 7.10: Verschiedene Ebenen der Änderungsspeicherung in dbExpress

Vom Änderungsprotokoll gibt es zwei Wege zur festen Speicherung der Änderungen, abhängig davon, ob diese Speicherung in einer MyBase-Datei oder auf dem Datenbank-Server stattfinden soll:

- Die Änderungen der MyBase-Datei können durch einen Aufruf von *TClientDataSet.MergeChangeLog* in den festen Datenbestand übernommen werden, jedoch ist zu beachten, dass *TClientDataSet* die gesamten Daten im Hauptspeicher aufbewahrt. Eine Übertragung der Daten auf Festplatte findet nur zu zwei besonderen Anlässen statt: Falls Sie im *FileName*-Property eine Datei angegeben haben, werden die Daten automatisch gespeichert, sobald die Datenmenge geschlossen wird. Unabhängig davon können Sie durch einen *SaveToFile*-Aufruf jederzeit explizit für eine Speicherung in einer beliebigen Datei sorgen.

Wenn die Speicherung stattfindet, bevor die Änderungen mit *MergeChangeLog* in den festen Datenbestand übernommen wurden, gehen die Änderungen nicht verloren, denn das ClientDataSet speichert auch das gesamte Änderungsprotokoll in der Datei und stellt es beim Laden komplett wieder her, so dass der Anwender mit den geänderten Daten weiter arbeiten kann und das Änderungsprotokoll fortlaufend erweitert wird, bis irgendwann *MergeChangeLog* aufgerufen wird (häufig wird *MergeChangeLog* allerdings nie aufgerufen wird; siehe Hinweis unten).

- Sollen die Änderungen an einen Datenbank-Server übertragen werden, muss statt *MergeChangeLog* die Methode *TClientDataSet.ApplyUpdates* aufgerufen werden. Hierbei kann es – vor allem, wenn mehrere Benutzer gleichzeitig auf die Datenbank zugreifen – zu Update-Konflikten kommen, die in Kapitel 7.5.5 behandelt werden. An dieser Stelle genügt es zu wissen, dass die Änderungen nur dann in den im Hauptspeicher gehaltenen Datenbestand der Komponente (*Data*) übernommen werden, wenn sie mit *ApplyUpdates* auch erfolgreich auf dem Server eingetragen werden konnten.

Das Verwerfen des Änderungsprotokolls funktioniert in beiden Fällen gleich: Ein Aufruf von *CancelUpdates* löscht alle noch im Protokoll befindlichen Änderungen, und die an die Datenmenge angeschlossenen datensensitiven Komponenten zeigen automatisch wieder den Ursprungszustand der Datenmenge an (vor der ersten verworfenen Änderung).

Hinweis: Beachten Sie, dass *jede* ClientDataSet-Komponente (auch *TSQLClientDataSet* und *TBDEClientDataSet*) ihre Daten jederzeit mit *SaveToFile* in eine MyBase-Datei speichern kann, auch wenn die Daten eigentlich von einem Datenbank-Server stammen. Dies ist für Anwendungen erforderlich, die nach dem Aktenkoffermodell arbeiten. Hier wird zunächst – während einer Verbindung zum Server – die ClientDataSet-Menge mit Daten vom Server geladen (die ClientDataSet-Komponente bewahrt das komplette Datenpaket in ihrem *Data*-Property auf).

Nun kann die Verbindung zum Server getrennt werden und die Daten können im ClientDataSet editiert sowie mit *SaveToFile* in lokalen (»MyBase-«)Dateien gespeichert werden. Die Änderungen werden *nicht* mit *MergeChangeLog* eingearbeitet, da das Änderungsprotokoll später, wenn wieder eine Verbindung zum Server besteht, mit *ApplyUpdates* zu diesem übertragen werden muss. Nach einem Aufruf von *MergeChangeLog* wäre diese Übertragung nicht mehr möglich, so dass in dieser Situation die Änderungen mit *MergeChangeLog* quasi »verloren« gingen (zumindest für den Server, auf den es hier ja ankommt).

Bei einem Aufruf von *ApplyUpdates* wird auch das *Data*-Property des ClientDataSets aktualisiert, daher ist danach ein *MergeChangeLog* nicht mehr erforderlich (wenn das ClientDataSet mit neuen Daten vom Server geladen wird, geht der bisherige Inhalt des *Data*-Properties sowieso verloren).

7.2.8 Objekt-Hierarchie und Diagramme

Sehr nützlich beim Entwurf größerer Datenbankanwendungen mit vielen Datenmengen sind das Objekthierarchie-Fenster und die Diagramm-Ansicht des Code-Editors (Abbildung 7.11), welche unter Delphi 5 noch fest in das Datenmodul-Fenster integriert waren, unter Delphi 6 jedoch auch für Formulare zur Verfügung stehen.

Die Objekthierarchie verdeutlicht die Beziehungen den Komponenten des Datenmoduls untereinander und zu verschiedenen im Hintergrund wirkenden Datenbankkomponenten. Diese Beziehungen lassen sich weder durch eine einfache Besitzhierarchie noch durch eine Klassenhierarchie darstellen, können also insbesondere mit Delphis allgemeinem Browser *nicht* klar erkannt werden. Die verschiedenen Arten von Beziehungen werden später noch anhand des Datendiagramms erläutert, welches dafür verschiedene Arten von Pfeilen verwendet.

Für die Baumansicht wurde eine prinzipiell willkürliche, aber doch sehr logische und vor allem nützliche Hierarchie-Aufteilung für die Datenbankkomponenten gewählt, die im Folgenden am Beispiel einer BDE-Anwendung erläutert ist: Der oberste Knoten ist immer das Datenmodul selbst. Sobald Sie eine Datenmenge in das Modul einfügen, also z.B. eine *TTable*-Komponente, erscheint diese in der Baumansicht drei Hierarchieebenen unter dem Datenmodul:

- Auf der ersten Ebene befindet sich ein vordefiniertes globales *TSession*-Objekt mit dem Namen *Session*. In der Komponentenpalette finden Sie unter BDE (bis Delphi 5: DATENZUGRIFF) allerdings eine *TSession*-Komponente, mit der Sie weitere Sessions erzeugen können, die dann in der Baumansicht auf dieser Ebene angeordnet werden. Das vordefinierte *Session*-Objekt wird auch als implizites Objekt bezeichnet. Solche Objekte, die Sie nicht selbst in das Datenmodul eingefügt haben, werden im Baumdiagramm mit einem blass schimmernden Icon symbolisiert.

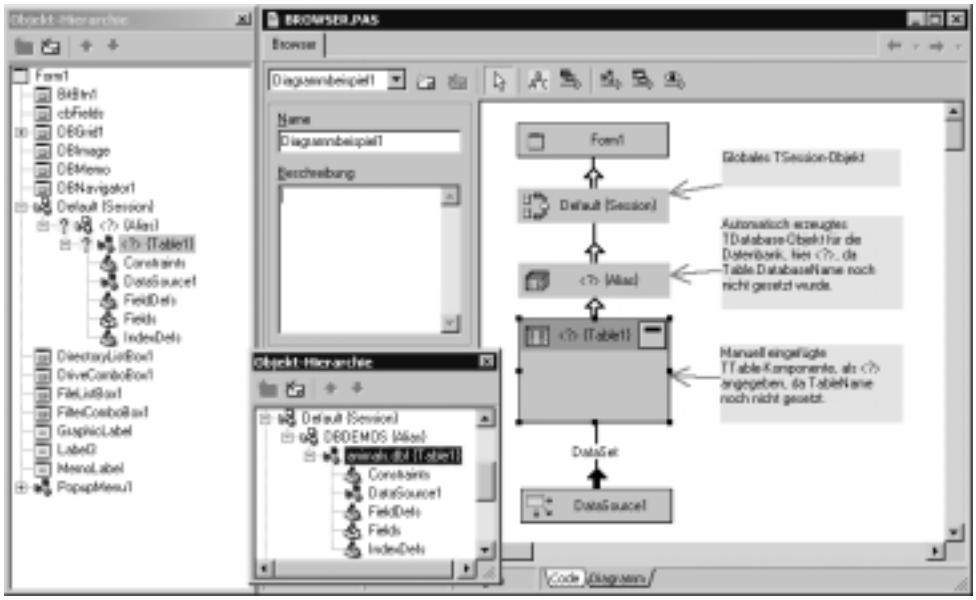


Abbildung 7.11: Die Objekthierarchie einer noch nicht komplett definierten BDE-Datenbankverbindung und ein auf dieser Grundlage angefertigtes Diagramm in Delphis Editorfenster. Das Bild des kleineren Objekthierarchie-Fensters im Vordergrund wurde nach Vervollständigung der Verbindungs-Daten angefertigt (DatabaseName und TableName wurden gesetzt).

- Die zweite Ebene wird durch die Datenbank selbst gebildet, und zwar die Datenbank, die Sie im *DatabaseName*-Property der Tabellen-Komponente angegeben haben. Falls Sie Tabellen mit verschiedenen *DatabaseName*-Properties verwenden, werden auf dieser Ebene automatisch entsprechende weitere Datenbank-Einträge gebildet. Sie können auch manuell einen neuen Eintrag erzeugen, indem Sie eine *TDatabase*-Komponente aus der Komponentenpalette einfügen.
- Auf der dritten Eben finden Sie schließlich die von Ihnen eingefügten Datenmengenkomponenten.
- Jeder Datenmenge ist eine weitere Ebene untergeordnet, in der die Eigenschaften dieser Datenmenge beschrieben werden, beispielsweise die Felder und Indizes. Aber auch *DataSource*-Komponenten, die mit dieser Datenmenge verknüpft sind, werden auf dieser Ebene aufgelistet.
- Zu einigen der Knoten der zuletzt beschriebenen Ebene gibt es weitere Ebenen, so z.B. für die Feld- und Indexlisten. Visuelle Komponenten, die an eine *DataSource* angeschlossen sind, werden übrigens nicht in untergeordneten Ebenen aufgelistet.

Hinweis: In dbExpress-Anwendungen finden Sie statt der Ebenen für die Session und die Datenbank eine Verbindungs-Ebene, die der Komponente *TSQLConnection* entspricht.

Neben der Veranschaulichung dient die Baumansicht auch dem Editieren selbst. Sie können Komponenten direkt in das Datenmodul einfügen. Wenn Sie dies intuitiv mit Drag&Drop versuchen sollten, erinnern Sie sich, wie Komponenten normalerweise in ein Delphi-Formular eingezeichnet werden: Die Komponente wird nicht aus der Palette gezogen, sondern dort nur markiert und dann mit einem Klick in das Formular eingefügt. So funktioniert es auch in der Baumansicht, wobei für jede Komponente natürlich nur bestimmte Positionen erlaubt sind. Innerhalb der Baumansicht können Sie Einträge jedoch erwartungsgemäß per Drag&Drop verschieben.

Eine besonders schöne Funktion der Baumansicht ist, dass sie die Properties der Komponenten beim Drag&Drop automatisch verändert. Wenn Sie z.B. eine *DataSource*-Komponente auf *Table1* ablegen, werden diese beiden automatisch miteinander verknüpft, indem *DataSource.DataSet* auf *Table1* gesetzt wird. Verschieben Sie die *DataSource* per Drag&Drop auf eine andere Tabelle, wird das *DataSet*-Property automatisch angepasst. Auf diese Weise können Sie die in Kapitel 7.2.1 beschriebene »Daten-Pipeline« visuell aufbauen.

Das Diagramm

In der Diagramm-Ansicht können Sie die Beziehungen der Komponenten auch grafisch und viel flexibler anzeigen als in der Baumansicht der Objekthierarchie. Damit in einem Diagramm überhaupt etwas angezeigt wird, müssen Sie die gewünschten Komponenten zuerst von der Baumansicht auf das Diagramm ziehen. Beziehungen zwischen den Komponenten werden dann über verschiedene Arten von Pfeilen dargestellt. Falls eine vom Diagramm unterstützte Beziehung bereits besteht, wird diese grundsätzlich automatisch angezeigt. Sie brauchen nur dann selbst Verbindungslinien zu zeichnen, wenn Sie neue Beziehungen definieren wollen.

Abbildung 7.11 zeigt ein Diagramm mit den Komponenten der verschiedenen Hierarchieebenen der Baumansicht für minimale Datenbankanwendungen mit nur einer Tabelle und einer Datenquelle. Das Diagramm gehört zum Datenbank-Browser aus Kapitel 7.2.5 und fällt ein wenig aus dem Rahmen, weil dieses Beispielprogramm ja erst zur Laufzeit eine Datenbankverbindung herstellt. Die Tabellen-Properties *DatabaseName* und *TableName* sind also zur Entwurfszeit noch undefiniert, weshalb die Tabellen- und Datenbankkomponente mit Fragezeichen versehen und deren Icons in der Baumansicht durch eine Umrandung markiert sind.

Die Abbildung zeigt zwei Arten von Verbindungslinien zwischen den Objekten:

- Der gefüllte Pfeil besagt, dass zwei Komponenten über ein Property miteinander verbunden sind. Der Pfeil beginnt bei der Komponente, deren Property auf die andere Komponente weist. Der Name des Properties wird als Beschriftung für den Pfeil angezeigt.
- Nicht gefüllte Pfeile bedeuten, dass die eine Komponente der anderen in der Hierarchie des Baumdiagramms direkt untergeordnet ist. Diese Pfeile werden allerdings nur angezeigt, wenn keine anderen Beziehungen existieren. Im Falle von Abbildung 7.11 wird für die Datenquelle beispielsweise nur ein gefüllter Pfeil für die Property-Verknüpfung angezeigt, obwohl auch ein leerer Pfeil zutreffend wäre.

Außerdem sind im linken Diagramm des Bildes noch die simplen Pfeile von den Textfeldern zu den drei mittleren Komponenten zu sehen. Diese sind selbst gezeichnet, wie die Textfelder selbst auch. Hierzu wird die Schalterleiste über dem Diagramm auf eine intuitive Weise verwendet.

Komponenten-Beziehungen editieren

Diese Schalterleiste kommt auch ins Spiel, wenn Sie die Komponenten-Beziehungen im Diagramm ändern wollen. Der in der Abbildung als Drittletztes zu sehende Schalter etwa (»Eigenschafts-Konnektor«) steht für einen Eigenschafts-Pfeil (zur Verknüpfung über ein Property). Wenn Sie mit der Maus eine solche Verbindung zwischen zwei Komponenten einzeichnen, bewirkt dies automatisch eine Verknüpfung dieser beiden Komponenten, sofern es zumindest ein passendes Property dazu gibt. Bei mehreren Properties werden Sie nach dem Einzeichnen der Linie zur Auswahl eines Properties aufgefordert.

Neben den Property-Verbindungen gibt es noch zwei weitere Beziehungsarten, die im Diagramm angezeigt werden können. Sie werden in den folgenden Kapiteln unter den Themen der Haupt-/Detailformulare und der Lookup-Felder erläutert.

7.3 Programmieren mit Feldern

Für eine Reihe wichtiger Programmieraufgaben genügt es nicht, nur mit den bisher beschriebenen Komponenten (*DataSet*-Komponenten, *DataSource* und Datensteuerelemente) zu arbeiten. Um beispielsweise bestimmte Datensätze (ohne SQL-Anweisungen) zu suchen oder zu editieren, benötigen Sie den Zugriff auf die einzelnen Felder der Tabelle.

7.3.1 TField und die Fields

Bevor eine Datenbankanwendung die Daten einer Tabelle darstellen kann, muss ihr die Struktur der Tabelle bekannt sein. Als Kenner dieser Struktur hat sich bisher besonders die *TDBGrid*-Komponente hervorgetan, denn sie erstellt automatisch alle am Bildschirm sichtbaren Spalten, die zur Darstellung der Tabellenfelder notwendig sind, passt ihre Breite an und richtet sich so auf jede Tabellenstruktur ein.

Bei allen anderen Datensteuerelement-Komponenten mussten wir das Property *DataField* auf das gewünschte Feld der Tabelle setzen. Da wir dieses aus einer Liste aller zur Verfügung stehenden Felder auswählen konnten, liegt der Gedanke nahe, dass diese Feldliste auch unter Object Pascal ansprechbar ist.

Natürlich ist sie das auch, aber das ist noch nicht alles: Die Feldliste ist nicht nur zur Laufzeit ansprechbar, sondern Sie können sie mit Hilfe des Felder-Editors und des Objektinspektors schon zur Entwurfszeit anpassen. Bei dieser Anpassung geht es jedoch nicht darum, die Struktur der Tabelle zu verändern, sondern darum, die Ausgabe der Felder in den Datensteuerelementen zu formatieren.

Die Tabellenstruktur zur Laufzeit

Unter Object Pascal ist die Felderliste aus nahe liegenden Gründen über ein Property des *DataSets* ansprechbar. Das Property heißt *Fields* und ist durch die Vererbung von *TDataSet* in allen Datenmengen-Komponenten vorhanden. Es ist wie ein Array-Property ansprechbar, das für jedes Feld der Tabelle ein *TField*-Objekt enthält. Die Zahl der definierten Felder erhalten Sie über das Property *TDataSet.FieldCount*.

Haben Sie beispielsweise eine ganz einfache Tabelle *Table1* mit den Feldern *Name* und *Vorname*, erhalten Sie mit *Table1.Fields[0]* ein *TField*-Objekt, das Informationen über das Feld *Name* enthält. *Table1.Fields[1]* oder *Table1.Fields[Table1.FieldCount-1]* gibt dementsprechend die Daten des Felds *Vorname* zurück.

Wenn Sie den Index eines Feldes nicht kennen, aber seinen Namen, können Sie das Feld mit der Funktion *FieldByName* ermitteln, beispielsweise mit *Table1.FieldByName('Schraubenzahl')*.

Tatsächlich handelt es sich beim *Fields*-Property jedoch nicht um ein Array, sondern um ein Objekt der Klasse *TFields*. In dieser Klasse finden Sie neben einigen anderen Elementen auch eine Methode *Add* (die dem Hinzufügen eines Feldes im Felder-Editor entspricht), ein Property *Count* (entspricht *TDataSet.FieldCount*) und eine Methode *FieldByName*.

Hintergrund: Zwischen dem Aufruf von *DataSet.Fields.FieldByName* und *DataSet.FieldByName* ist nur dann ein Unterschied feststellbar, wenn das *DataSet* Aggregat-Felder enthält. Diese sind nämlich nicht dem *Fields*-Objekt zugeordnet und deshalb auch nicht von seiner *FieldByName*-Methode ansprechbar.

TField-Properties zur Feldstruktur

Die folgende Tabelle listet die grundlegenden Properties der Klasse *TField* auf:

Da die Veränderung der Tabellenstruktur nicht so ohne weiteres möglich ist (wenn der bisherige Inhalt der Tabelle weiterbestehen soll, benötigt man hierfür einen Spezialisten wie die Datenbankoberfläche oder eine Menge Object-Pascal-Code), können Sie die *TField*-Properties, die die Tabellenstruktur beschreiben, nur lesen:

Property	Bedeutung
FieldName	gibt den Namen der Spalte an, so, wie er in der Tabelle selbst definiert ist.
Calculated	gibt mit *True* an, dass es sich nicht um ein Feld der physikalischen Tabelle handelt, sondern dass der Feldinhalt zur Laufzeit berechnet wird (siehe auch den Abschnitt *Erzeugung persistenter Felder* in Kapitel 7.3.2).
DataSize	enthält die Zahl der Bytes, die der Inhalt eines Felds im Hauptspeicher benötigt.
DataType	enthält eine der im nächsten Abschnitt beschriebenen Konstanten, die den Datentyp des Felds angeben.
DataSet	weist auf die *TDataSet*-Komponente, zu der das *Field* gehört.
IsIndexField	gibt an, ob der aktuelle Index dieses Feld enthält.
Index	gibt den Index des Feldes in der *Fields*-Liste an.
Lookup	Wenn *True*, handelt es sich um ein Lookup-Feld (siehe hierzu den Abschnitt *Lookup-Felder* in Kapitel 7.3.2). Wie bei *Calculated=True* handelt es sich auch bei einem Lookup-Feld nicht um ein Feld der physikalischen Tabelle. Nur eines der Properties *Calculated* und *Lookup* kann den Wert *True* haben.

TField-Typen

Abbildung 7.12 zeigt den Hierarchiebaum der *TField*-Klassen. Die VCL erstellt beim Öffnen einer Tabelle per Voreinstellung automatisch für jedes Feld der Tabelle ein *TField*-Objekt des passenden Typs.

Die Feldtypen, mit denen Sie es in Delphi zu tun haben, sind glücklicherweise unabhängig von den verschiedenen Tabellenformaten. In Delphi spielt es also keine Rolle, ob ein *TIntegerField* auf dem Feld einer dBase-Tabelle beruht, die die Zahl als Zeichenkette darstellt, oder auf einer Paradox-Tabelle, die mit Binärzahlen arbeitet. *TIntegerField* verbraucht im Hauptspeicher immer 4 Bytes für den Feldinhalt und wird bei der

Übertragung der Daten zur Tabelle vom Gespann VCL/BDE automatisch in das Zahlenformat der Tabelle umgewandelt.

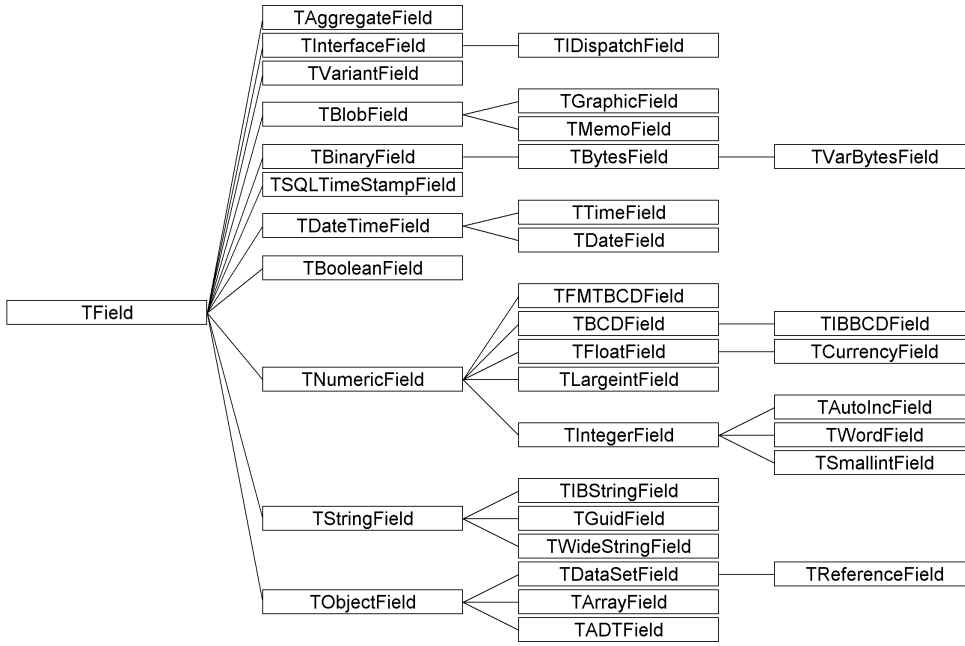


Abbildung 7.12: Die Hierarchie der TField-Klassen (Basisklasse von TField ist TComponent)

Wie bereits erwähnt, befindet sich unter den *TField*-Properties auch ein Property *DataType*. In diesem speichert die VCL den Datentyp des Feldes zusätzlich zu der Laufzeittypinformation, mit der Sie auch feststellen könnten, ob ein *TField*-Objekt beispielsweise die Klasse »TStringField« aufweist (zur Laufzeit-Typinformation siehe Kapitel 2.3.6). Da das Feld *DataType* jedoch wesentlich schneller anzusprechen ist und extra für die Datentypspeicherung gedacht ist, sollten Sie seine Abfrage der Verwendung der Laufzeittypinformationen vorziehen.

Die folgende Tabelle listet die wichtigsten *TField*-Klassen auf:

Klasse	Feldtyp-Konstante	Typ des Value-Properties	Bemerkung
T(Wide)StringField	ftString	(Wide)String	Textdaten, Länge abhängig von Felddefinition
TIntegerField	ftInteger	LongInt	
TSmallIntField	ftSmallInt	LongInt	Wertebereich des Object-Pascal-Typs *SmallInt* (16-Bit)

Klasse	Feldtyp-Konstante	Typ des Value-Properties	Bemerkung
TLargeIntField	ftLargeInt	Int64	
TWordField	ftWord	LongInt	Wertebereich entspricht dem Object-Pascal-Typ *Word*
TBooleanField	ftBoolean	Boolean	*True* oder *False*
TFloatField	ftFloat	Double	Fließkommazahlen im Bereich von 5.0*10-324 bis 1.7*10308, Genauigkeit 15-16 Ziffern
TCurrencyField	ftCurrency	Double	Feldinhalt ist als Währungswert definiert, wird aber intern wie ein *TFloatField* dargestellt
TDateTimeField	ftDateTime	TDateTime	Datum und Uhrzeit im Format *TDateTime*
TDateField	ftDate	TDateTime	wie *TDateTimeField* im Format *TDateTime*, aber mit einer auf 0:00 Uhr eingestellten Zeit
TTimeField	ftTime	TDateTime	ebenfalls im Format *TDateTime*, aber mit gelöschtem Datumsanteil
TBlobField	ftBlob, ftFmtMemo, ftParadoxOle, ftDBaseOle, ftTypedBinary	String	binäres Datenfeld unbegrenzter Länge
TMemoField	ftMemo	String	Interpretation der Blob-Daten als Text
TGraphicField	ftGraphic	- (String, von Blob geerbt)	Blob-Daten, die als Grafik (Bitmap) interpretiert werden

TField-Properties zur Anzeigesteuerung

Neben den bisher besprochenen nicht beschreibbaren Properties verfügt *TField* über eine noch größere Anzahl von Properties, die festlegen, wie die Daten des entsprechenden Feldes in den visuellen Komponenten angezeigt werden. Alle von Delphi bereitgestellten datensensitiven Komponenten richten sich nach diesen Werten.

Die folgende Tabelle behandelt die Properties aller Feldtypen, wobei viele Properties nur bei bestimmten Feldtypen vorhanden sind. So gibt es z.B. *MinValue* / *MaxValue* nur bei den Feldern, die Zahlen beinhalten. Speziell diese beiden Properties haben sogar je nach Feldtyp unterschiedliche Typen. Während ein *TIntegerField* durch *LongInt*-Werte begrenzt ist, benötigen die Grenzen eines *TFloatField* beispielsweise den Datentyp *Double*. Manche der Properties werden innerhalb der VCL nur von *TDBGrid* beachtet.

Property	Bedeutung
Alignment	wählt zwischen Linksbündigkeit, Rechtsbündigkeit und Zentrierung.
Calculated	gibt an, dass der Feldinhalt zur Laufzeit beim *TTable*/*TQuery*-Event *OnCalcFields* berechnet wird.
Currency	gibt an, ob eine datensensitive Komponente den Wert als Währung darstellen soll.
DisplayFormat	gibt für Datums-, Zeit- und Zahlenwerte einen Formatierungs-String an, der letztlich von den *SysUtils*-Funktionen *FormatDateTime* und *FormatFloat* verwendet wird, um die Darstellung des Feldwerts zu erzeugen.
DisplayLabel	gibt die Spaltenüberschrift für ein *DBGrid* an, falls diese nicht mit *FieldName* übereinstimmt.
DisplayWidth	*TDBGrid* verwendet diesen Wert, um die Breite der zum Feld gehörenden Spalte in Zeichen festzulegen.
EditFormat	steht im Gegensatz zu *DisplayFormat* und spezifiziert das Format, das nur zum Editieren des Werts verwendet werden soll.
EditMask	gibt eine Maske an, die die Eingabe beim Editieren des Felds in einer *TMaskEdit*- oder *TDBGrid*-Komponente beschränken und formatieren soll (Details siehe Online-Hilfe).
FieldName	enthält den Namen der Spalte in der Originaltabelle und stellt damit die Verbindung vom *TField* zur tatsächlichen Spalte her.
MaxValue	gibt den maximalen Wert an, der beim Editieren des Feldes eingegeben werden darf.
MinValue	schließt den von *MaxValue* begonnenen Bereich nach unten ab.
ReadOnly	verhindert das Editieren des Feldes in den Datensteuerelementen, wenn *True*.
Size	steht zwar im Objektinspektor, ist aber von der verwendeten Tabelle abhängig und sollte normalerweise nicht geändert werden.
Tag	frei verfügbarer Wertespeicher, geerbt von *TComponent*

7.3.2 Persistente Felder und der Felder-Editor

Per Voreinstellung reagiert eine DataSet-Komponente flexibel auf die Struktur einer jeden Tabelle, mit der Sie sie verbinden. Jedes Mal, wenn Sie die Tabelle (*TableName*) wechseln und die Verbindung zu ihr durch das Property *Active* einschalten, liest die DataSet-Komponente die Tabellenstruktur ein und baut so eine neue dynamische *Fields*-Liste auf. Da die *TField*-Komponenten dieser Liste viele Properties beinhalten, insbesondere solche, die sich visuell auswirken, ist es wünschenswert, die *TField*-Komponenten schon zur Entwurfszeit im Objektinspektor anpassen zu können.

Der Felder-Editor

Die Auswahl der Felder zur Anpassung im Objektinspektor zu ermöglichen ist eine Aufgabe des Felder-Editors (Abbildung 7.13), den Sie über ein Popup-Menü von Datenmengen-Komponenten oder über einen Doppelklick auf dieselben aufrufen.

Der Felder-Editor zeigt eine Liste von *TField*-Komponenten an, die zur Entwurfszeit nur über den Felder-Editor selbst ausgewählt und dann im Objektinspektor editiert werden können – ähnlich der Situation des Menü-Designers und der *TMenuItem*-Komponenten. Beim ersten Aufruf des Editors ist die Felderliste noch leer, weil alle Felder dynamisch angelegt werden, selbst wenn Sie das Formular mit dem Datenbankformular-Experten erzeugt haben.

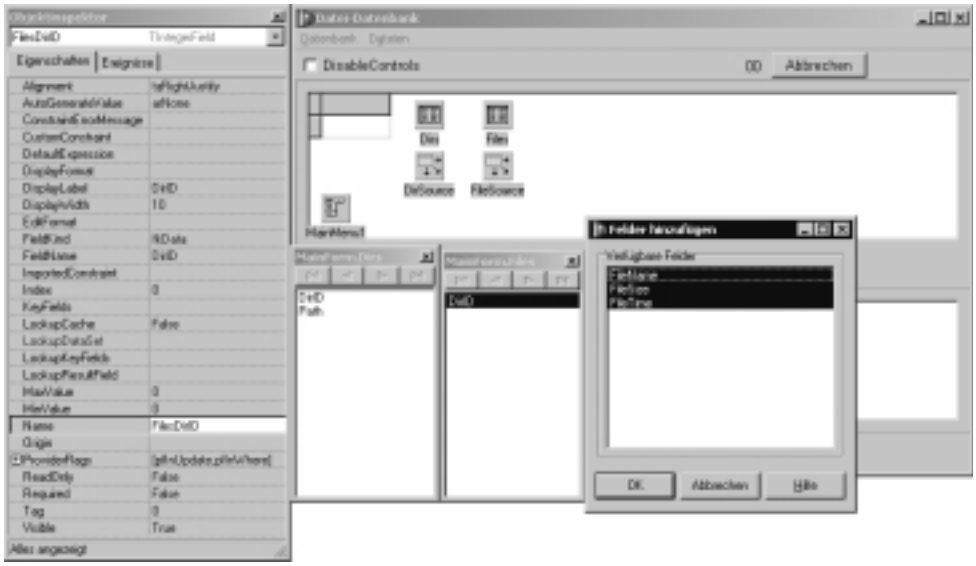


Abbildung 7.13: Felder-Editoren treten in Form von unscheinbaren Listen wie hier die Fenster mit den Überschriften MainForm.Dirs und MainForm.Files auf. Der Objektinspektor zeigt die Attribute des Feldes Files.DirID; der Felder-hinzufügen-Dialog stellt weitere Felder der Files-Tabelle, für die noch keine persistenten Feldkomponenten existieren, zur Wahl.

Funktion persistenter Felder

Das entscheidende Merkmal der Liste im Felder-Editor ist, dass sie nicht bei jeder Verknüpfung mit einer anderen Tabelle dynamisch neu aufgebaut wird, sondern dass sie statisch ist, und dies sogar über einen beliebig langen Zeitraum, denn diese Felder werden in der Formulardatei gespeichert und daher auch als *persistent* bezeichnet.

Die *DataSet*-Komponente kann natürlich zur Laufzeit des Programms feststellen, ob bereits *TField*-Komponenten aus der Formulardatei gelesen werden konnten. Ist das der Fall, so zieht sie daraus einen wichtigen Schluss: Sie erzeugt im Folgenden *keinerlei* dynamische Felder mehr, egal, mit welcher Tabelle sie verbunden wird. Die persistenten Felder müssen auf jede Tabellenstruktur, der die *DataSet*-Komponente begegnet, passen. Das heißt, dass zu jeder *TField*-Komponente eine Spalte der Tabelle vorhanden

sein muss, die mit dem Property *FieldName* der *TField*-Komponente übereinstimmt und einen passenden Datentyp aufweist.

Die Reihenfolge der Spalten in der Tabelle spielt keine Rolle, sie beeinflusst die Reihenfolge der Feld-Komponenten, die Sie im Felder-Editor festgelegt haben, nicht. Das bedeutet, dass Sie durch die Verwendung des Felder-Editors eine von der tatsächlichen Reihenfolge der Spalten unabhängige Fields-Liste erhalten.

Außerdem ist es möglich, dass die tatsächliche Tabelle mehr Spalten enthält, als *TField*-Komponenten vorhanden sind. Auch in diesem Fall erzeugt die *DataSet*-Komponente keine dynamischen *TField*-Komponenten für die unverknüpften Spalten, was zur Folge hat, dass diese Spalten beispielsweise in einem *TDBGrid* gar nicht erscheinen würden. Kapitel 7.3.3 zeigt, wie Sie diese Tabellenspalten ohne *TField*-Komponenten dennoch zur Laufzeit erkennen können.

Erzeugung persistenter Felder

Sie haben zwei Möglichkeiten, neue persistente Felder zu erzeugen:

- Wenn Sie die *DataSet*-Komponente vor dem Aufruf des Felder-Editors mit einer Tabelle verknüpfen (*DatabaseName* und *TableName* müssen gesetzt sein, nicht aber das *Active*-Property), kann der Felder-Editor seinerseits eine dynamische Liste aller in dieser Tabelle verfügbaren Spalten aufbauen, die er in einer seiner Listen zur Wahl stellt.

  Wählen Sie aus dem lokalen Menü des Felder-Editors den Punkt FELDER HINZUFÜGEN... und aus dem folgenden Dialogfenster die hinzuzufügenden *TField*-Komponenten. Wenn Sie einen geschlossenen Bereich wählen (in einer Maus-Ziehaktion oder mit Shift und Strg), können Sie mehrere Felder gleichzeitig hinzufügen.

- Sie können auch per Hand *TField*-Komponenten erstellen, indem Sie den Punkt NEUES FELD... wählen und in der folgenden Dialogbox einen Feldnamen, einen Namen für die *TField*-Komponente und einen Datentyp angeben. Auf diese Weise können Sie Felder für Tabellen definieren, die zurzeit nicht auf Ihrem lokalen Computersystem verfügbar sind (geben Sie dazu im Eingabefeld *Feldname* den Namen ein, den das zurzeit nicht verfügbare Feld in der echten Tabelle hat). Vor allem dient der Dialog aber der Definition von Lookup-Feldern, berechneten Feldern oder Feldern einer Tabelle, die Sie neu erstellen wollen (zu Letzterem siehe Kapitel 7.1.2, auf Nachschlagefelder werden wir in Kürze zurückkommen).

Felder in der Objekthierarchie

Die persistenten Felder werden übrigens auch in der Objekthierarchie (bzw. unter Delphi 5 im Datenmodul-Fenster) angezeigt, und zwar innerhalb des *Fields*-Knotens einer Datenmengen-Komponente (siehe Abbildung 7.14). Sie können ein Feld auch dadurch

in den Objektinspektor laden, dass Sie es in dieser Baumansicht auswählen, und Sie haben sogar die Möglichkeit, neue Felder statt über den Felder-Editor in dieser Baumansicht zu definieren.

In der Diagrammansicht des Editorfensters können Felder auf zwei Arten angezeigt werden: Zum einen innerhalb des Tabellenobjekts, wenn Sie dieses mit dem Schalter rechts oben in der Ecke expandieren. Zum anderen, indem Sie es mit der Maus von der Baumansicht in das Diagramm ziehen. In diesem Fall wird auch automatisch eine Linie vom Feld zur Tabelle gezeichnet (falls sich die zugehörige Tabelle ebenfalls im Diagramm befindet).

Felder im Beispielprogramm

Das Beispielprogramm *FileDB* (Kapitel 7.3.6) verwendet persistente Felder, um die Formatierung der Datei- und Verzeichnisliste zur Entwurfszeit festzulegen und um über die im Objektinspektor festgelegten Namen der *TField*-Komponenten im Quelltext auf einfache Weise auf die Felder des aktuellen Datensatzes zugreifen zu können.

Funktionen des Felder-Editors

Einige wichtige Funktionen des Felder-Editors sollen nicht unerwähnt bleiben:

- In der Liste des Felder-Editors haben Sie zunächst die Möglichkeit, die Felder per Drag&Drop umzusortieren.
- Bei der Schalterleiste unter der Feldliste handelt es sich um einen *DBNavigator*, der auf vier Schalter beschränkt wurde. Mit ihm können Sie schon zur Entwurfszeit durch die Datenmenge blättern (falls diese aktiviert und mit datensensitiven Steuerelementen verbunden ist), um etwa die Wirkung der *TField*-Properties auf die Darstellung der einzelnen Felder zu begutachten. Das Blättern ist besonders dann nützlich, wenn das Formular kein *DBGrid* enthält, sondern nur einen einzigen Datensatz gleichzeitig anzeigt.
- Sie können die in ihm aufgelisteten Felder komfortabel per Drag&Drop auf eines Ihrer Formulare ziehen. Delphi fügt dann in diesem Formular ein Datensteuerelement ein, das zu dem Typ des Feldes passt, und verknüpft es gleichzeitig mit diesem.
- Aus dem Kontextmenü können Sie außerdem einige Aktionen starten, die mit den Data Dictionaries bzw. Datenwörterbüchern (die früher verwendete Übersetzung) zusammenhängen. Diese werden nachfolgend in einem eigenen Abschnitt behandelt.

Lookup-Felder *R179*

Lookup-Felder zeigen einmal mehr, dass ein *TField*-Objekt nicht immer mit einem wirklich in der Tabelle (bzw. in einem beliebigen *DataSet*) existierenden Datenfeld übereinstimmen muss, dass die Klasse *TField* also zunächst einmal nur ein abstraktes Konzept beschreibt. Wie die berechneten Felder (Property *Calculated=True*) enthalten auch Lookup-Felder (*lookup* = nachschlagen, Property *Lookup=True*) keine Werte aus der Tabelle, zu der sie gehören, sondern sie schlagen ihren Wert in einer anderen Tabelle, der Lookup-Tabelle, nach.

In den Lookup-Vorgang sind drei verschiedene Felder einbezogen; in den ersten beiden Feldern können Sie statt eines einzelnen Feldes auch eine Feldliste angeben:

- Die *Schlüsselfelder* (*KeyFields*) sind ein oder mehrere Felder der *DataSet*-Komponente, in der das Lookup-Feld enthalten ist. Beim Lookup-Vorgang wird in der Lookup-Tabelle nach dem Wert gesucht, der im Schlüsselfeld des aktuellen Datensatzes enthalten ist.
- Die *Lookup-Schlüssel* (*LookupKeyFields*) sind Felder (oder ein einzelnes Feld) der Lookup-Tabelle, in denen nach dem Wert der Schlüsselfelder (bzw. des Schlüsselfeldes) gesucht werden soll. Die Lookup-Tabelle wird also quasi über ihre *Lookup-Schlüssel* mit dem Wert der Schlüsselfelder der ersten Tabelle verbunden (Ähnliches geschieht bei einem Haupt-/Detailformular (siehe Kapitel 7.4.5).
- Das *Ergebnisfeld* (*LookupResultField*) ist das Feld der Lookup-Tabelle, das schließlich den Wert des Lookup-Feldes liefert.

Im Felder-Editor können Sie ein Lookup-Feld definieren, indem Sie NEUES FELD... aus dem Popup-Menü auswählen und in der Dialogbox den Schalter NACHSCHLAGEN (bis Delphi 4: LOOKUP) markieren. Die drei oben beschriebenen Felder(listen) und die Lookup-Tabelle (*LookupDataSet*) geben Sie im unteren Teil der Dialogbox ein. Die Werte des Dialogabschnitts *Feldeigenschaften* müssen außerdem mit dem Typ des *Ergebnisfeldes* übereinstimmen.

Nachdem Sie ein Lookup-Feld über diese Dialogbox erzeugt haben, können Sie alle genannten Parameter unter den in Klammern genannten Namen (*LookupDataSet*, *LookupKeyFields* usw.) auch im Objektinspektor finden, wenn Sie das Lookup-Feld im Feld-Editor auswählen.

Lookup-Beispiel

Als Beispiel sollen wieder die schon am Anfang von Kapitel 7.1 eingeführten Tabellen für Dateien und Verzeichnisse dienen, deren Erzeugung in den Kapiteln 7.1.1 bis 7.1.3 erläutert wurde. Wir verwenden hier zwei *TTable*-Komponenten, um auf die dBase/Paradox-Version der Tabellen zuzugreifen. Die Dateitabelle *Files* enthält anstatt des

Verzeichnispfades nur eine Verzeichnisnummer. Wird diese Tabelle unverändert in einem *DBGrid* angezeigt, ist die Verzeichnisnummer für den Benutzer wenig informativ.

Diese Verzeichnisnummer soll nun durch ein Lookup-Feld ersetzt werden. Dazu wird in der Felderliste der *TTable*-Komponente *Files* mit NEUES FELD... ein Feld hinzugefügt, dessen FELDTYP auf NACHSCHLAGEN eingestellt wird. Dieses Lookup-Feld soll den Namen des Verzeichnisses in der Verzeichnistabelle nachschlagen (*LookupResultField := 'Path'*), und zwar indem es die Nummer aus dem Feld *DirID* der Tabelle *Files* im Feld *DirID* der Tabelle *Dirs* sucht. Sowohl *LookupDetailFields* als auch *LookupKeyFields* enthalten somit den Feldnamen *'DirID'*. Das Property *LookupDataSet* wird auf *Dirs* gesetzt.

Nach Hinzufügen dieses Lookup-Feldes ist es möglich, alle in der Dateidatenbank gespeicherten Daten der Dateien, inklusive der Verzeichnisnamen, in einer einzigen Tabelle anzuzeigen. Sie finden dieses Beispiel auf der CD in zwei Variantionen:

- Ein Formular, welches nur die Dateitabelle anzeigt, allerdings wie oben beschrieben um das Lookup-Feld erweitert, finden Sie im Projekt *VzLookup*. Für den Datenzugriff wird hier eine *TTable*-Komponente verwendet.
- Das Projekt *IBFileDB* (die Interbase-Version der Dateidatenbank) verwendet statt *TTable* ein *TSQLClientDataSet* (also dbExpress statt BDE). Hier können Sie zur Laufzeit durch den Menüpunkt FENSTER | LOOKUPFIELD-DEMO ebenfalls ein Formular aufrufen, das eine um das Lookup-Feld erweiterte Dateitabelle anzeigt. Das Lookup-Feld wurde hier in die Felderliste der *TSQLClientDataSet*-Komponente eingefügt. In der Definition des Lookup-Feldes gibt es keine Unterschiede zur BDE-Variation.

Allerdings ist es im Falle der Dateidatenbank nach wie vor sinnvoller, beide Tabellen getrennt in einem Haupt-/Detailformular darzustellen, wie wir es in Kapitel 7.4.5 tun werden.

Hinweis: Wenn Sie ein Lookup-Feld nicht wie in diesem Beispiel in einem DBGrid anzeigen, sondern das Feld mit einer *DBLookupCombobox*-Komponente (oder einer *DBLookupListBox*) verknüpfen, kann der Benutzer zur Laufzeit einen der nachgeschlagenen Werte auswählen. Ein Beispiel dafür finden Sie im Termindialog von Kapitel 7.5.3.

Lookup-Felder in der Diagramm-Ansicht

In der Diagramm-Ansicht wird eine Lookup-Beziehung zwischen zwei Tabellen durch eine Linie gekennzeichnet, an deren Ende sich ein Augen-ähnliches Symbol befindet (das Symbol befindet sich an der Tabelle, in der nachgeschlagen werden soll). Die Linie wird mit dem Namen des Feldes, in das die nachgeschlagenen Werten eingesetzt werden, beschriftet.

Wenn Sie das Lookup-Feld außerdem noch als einzelnes Objekt in das Diagramm einfügen, werden von diesem ausgehend zweierlei Linien gezeichnet: die bei jedem Feld gezeichnete Hierarchielinie mit dem leeren Pfeil und die Property-Linie für das *LookupDataSet*-Property (siehe Abbildung 7.14; zum Einfügen des Feldes in das Diagramm expandieren Sie in der Baumansicht den *Field*s-Knoten der Tabelle und ziehen den Feld-Knoten manuell wie die Tabellenkomponenten in das Diagramm).

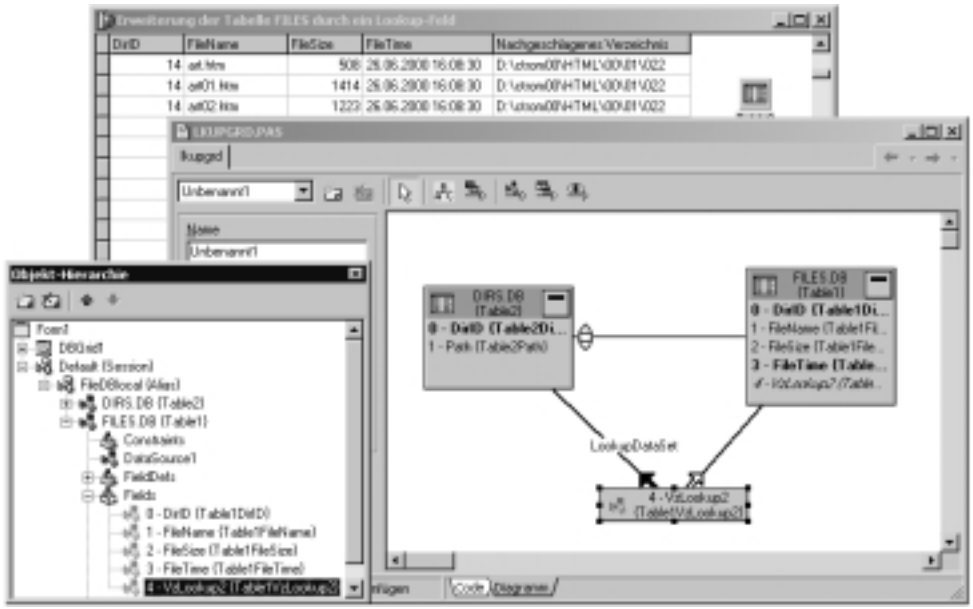


Abbildung 7.14: Die Diagramm-Ansicht mit Anzeige der Felder von Tabellen (ein Feld auch einzeln dargestellt) und einer Lookup-Beziehung

Auch in der Schalterleiste neben dem Diagramm finden Sie das Augensymbol wieder. Mit ihm können Sie manuell Lookup-Verknüpfungen herstellen. Durch das Einzeichnen der Linie wählen Sie lediglich zwei Tabellen aus und gelangen in einen Dialog, in dem Sie dann alles Weitere einstellen.

Datenwörterbücher / Data Dictionaries

Wenn Sie für den Datenzugriff die BDE verwenden, können Sie die Einstellungen von *TField*-Properties bestimmter Felder als Schablonen in so genannten Data Dictionaries (auch als »Datenwörterbücher« bezeichnet) ablegen. Diese Dictionaries können Sie im Datenbank-Explorer der Delphi-IDE (DATENBANK | EXPLORER oder lokales Menü einer DataSet-Komponente) auf der Seite DICTIONARY einsehen. Zur Nutzung dieser Funktion müssen folgende Voraussetzungen erfüllt sein:

- Der Zugriff auf die Datenbank muss über ein BDE-Alias erfolgen (siehe Seite *Datenbanken* des Datenbank-Explorers) und für dieses BDE-Alias muss ein Dictionary eingerichtet sein (Sie können ein neues Dictionary mit DICTIONARY | NEU... im Datenbank-Explorer erzeugen).
- Der Alias-Name muss auch im aktuellen Data Dictionary eingetragen sein. Um einen neuen Alias im Dictionary aufzunehmen, wählen Sie im Explorer den Menüpunkt DICTIONARY | AUS DATENBANK IMPORTIEREN...

Nach diesen Vorarbeiten können Sie die Attribute Ihrer persistenten Felder (oben beschrieben im Abschnitt *TField-Properties zur Anzeigesteuerung*) nach Belieben ändern und sie danach quasi als Schablone im Dictionary speichern. Hierzu wählen Sie den Menüpunkt ATTRIBUTE SPEICHERN (UNTER...) aus dem Kontextmenü des Felder-Editors. Sie erhalten eine Dialogbox, die einen Namen für den Attributsatz festlegt, mit dem das Feld so lange verbunden bleibt, bis Sie es mit ATTRIBUTE VERKNÜPFEN (Bis Delphi 4: ATTRIBUTE ZUORDNEN...) einem anderen Attributsatz zuordnen oder bis Sie die Zuordnung ganz aufheben (ATTRIBUTE-ZUORDNUNG LÖSEN).

Der Vorteil der Dictionaries kommt natürlich erst zum Tragen, wenn Sie einen Attributsatz auf mehrere Felder anwenden: Sie können ein Feld mit ATTRIBUTE VERKNÜPFEN... einem bestehenden Attributsatz zuordnen. Die Properties des Feldes werden dann mit den Werten gefüllt, die im Attributsatz gespeichert sind. Mit ATTRIBUTE HOLEN laden Sie die Properties neu, falls der Attributsatz in der Zwischenzeit per ATTRIBUTE SPEICHERN geändert wurde.

7.3.3 Feld-Definitionen

Im letzten Abschnitt wurde beschrieben, dass keine dynamischen Felder erzeugt werden, wenn Sie im Felder-Editor nur ein einziges persistentes Feld definieren. Das bedeutet jedoch nicht, dass Sie zur Laufzeit keinen Zugriff mehr auf die restlichen Spalten der Tabelle haben. Die *Fields*-Liste der *TDataSet*-Komponenten, die zur Laufzeit entweder die persistenten oder die dynamisch erzeugten Felder enthält, ist gar nicht dazu gedacht, die genaue Struktur der Tabelle wiederzugeben, sondern sie soll die Delphi-Anwendung von der tatsächlichen Tabellenstruktur abschirmen und die

Festlegung der Darstellungs-Properties zur Entwurfszeit sowie die Definition von berechneten und Lookup-Feldern ermöglichen.

Das *TDataSet*-Property, das die tatsächliche Tabellenstruktur enthält, ist *FieldDefs*. Es lässt sich wie ein indiziertes Property ansprechen, so dass Sie mit *TDataSet.FieldDefs[Index]* ein Element der Tabellenstruktur erhalten, und zwar die Definition eines einzelnen Felds, die in Form eines *TFieldDef*-Objekts vorliegt.

TFieldDefs und TFieldDef

FieldDefs ist ein Objekt der Klasse *TFieldDefs*, das sich wie ein Array-Property ansprechen lässt. Sein Standard-Property *Items* gibt die einzelnen *TFieldDef*-Werte an und sein Property *Count* die Anzahl dieser Werte.

Im Gegensatz zu einem *TField*, in dem die allermeisten Daten nicht zur eigentlichen Felddefinition, sondern zur Darstellung auf dem Bildschirm gehören, fällt die Property-Auswahl in einem *TFieldDef*-Objekt sehr klein aus: In *Name*, *DataType* und *Size* sind genau die Daten gespeichert, die Sie auch bei der Erstellung der Tabelle in der Datenbankoberfläche angeben. Dazu kommen unter anderem bei Zahlenangaben eine Genauigkeitsangabe (*Precision*), einige Flags im Property *Attributes* (z.B. *faRequired*, wenn das Feld nicht leer sein darf, oder *faReadOnly*, wenn es schreibgeschützt ist), und ein Property, das den Index des Felds in der Felderliste angibt (*FieldNo*).

Die Klasse *TFieldDefs* enthält außerdem zwei interessante Methoden zum Hinzufügen neuer *TFieldDef*-Objekte, von denen die Methode *Add* in Kapitel 7.1.2 bei der Erzeugung einer neuen Tabelle verwendet wird.

Hintergrund: Programmtechnisch interessant ist das Property *FieldClass*: Es enthält eine Klassenreferenz auf die Klasse der *TField*-Hierarchie, die für den Typ des Felds zuständig ist. Die Methode *TFieldDef.CreateField* hat die Aufgabe, ein *TField*-Objekt dieser Klasse zu erzeugen. Sie wird intern von der VCL verwendet, um bei der Verbindung mit einer Datenmenge, für die keine persistenten Feldkomponenten definiert wurden, dynamische Feldkomponenten zu erzeugen (zu jedem Element von *FieldDefs* wird dann eine Feldkomponente in *Fields* erzeugt).

Felddefinitionen zur Entwurfszeit editieren

Zur Entwurfszeit können Sie das *FieldDefs*-Property dazu verwenden, eine neue dBase- oder Paradox-Tabelle zu erzeugen. Dies läuft ähnlich ab wie es für das Definieren einer neuen MyBase-Tabelle in Kapitel 7.1.2 beschrieben wurde, allerdings mit dem wichtigen Unterschied, dass *nicht* der Felder-Editor zur Definition der neuen Felder verwendet wird, sondern der Editor, den Sie aus dem Objektinspektor für das Property *FieldDefs* aufrufen.

Wenn Sie darin die gewünschten Felder definiert haben, können Sie aus dem lokalen Menü der *TTable*-Komponente den Befehl TABELLE ERSTELLEN aufrufen. Falls Sie dieser Befehl nicht in Ihrem Menü auftaucht, sind nicht alle Bedingungen erfüllt: *FieldDefs* darf nicht leer sein, die Felderliste im Felder-Editor *muss* leer sein, und die Properties *TTable.DatabaseName* und *TTable.TableName* müssen eine noch nicht existierende Tabelle festlegen.

Falls *DataBaseName/TableName* eine existierende Tabelle bezeichnen, stehen Ihnen im lokalen Menü drei andere Befehle zur Verfügung, mit denen Sie die Tabelle löschen und umbenennen sowie die *FieldDefs*-Werte neu aus der Tabelle auslesen können (TABELLENDEFINITION AKTUALISIEREN). Letzterer Punkt und der Befehl TABELLE ERSTELLEN beziehen sich auch auf das Property *IndexDefs*, das einen ähnlichen Property-Editor sein eigen nennt. Zu den Indexdefinitionen kommen wir in Kapitel 7.4.2.

Hinweis: Um die Struktur einer bestehenden dBase/Paradox-Tabelle zu ändern, ohne sie vorher löschen zu müssen, können Sie die Datenbankoberfläche verwenden.

7.3.4 Ein Beispielprogramm mit dynamischen Feldern

Wir setzen an dieser Stelle das in Kapitel 7.2.5 begonnene Beispielprogramm fort. Ein Nachteil des Datenbank-Browsers war bisher, dass er die Daten zwar sehr übersichtlich in einer *DBGrid*-Tabelle anzeigt, diese aber nicht alle Datenelemente darstellen kann: Grafiken, Memos und andere BLOBs erhalten zwar eine eigene Spalte, diese enthält jedoch immer nur einen ersatzweisen Text wie »(Memo)« oder »(Blob)« .

Mit der hier beschriebenen Erweiterung durchsucht der Browser die Struktur einer beliebigen Tabelle zur Laufzeit nach einem Grafik- und einem Memofeld. Das letzte gefundene Feld jedes dieser Typen stellt er jeweils in einem statisch im Formular eingebauten Datensteuerelement dar (einer *TDBImage*- bzw. einer *TDBMemo*-Komponente).

Einbau der Komponenten

Zu den schon in Kapitel 7.2.5 erwähnten Komponenten kommen in der vollständigen Version des Programms die folgenden hinzu:

- *cbFields* ist eine Kombobox, in der das Programm alle Felder der gerade gewählten Datenbank auflistet: Für jedes in *FieldDefs* angegebene Feld erweitert das Programm die Stringliste (*Items*) dieser Kombobox um den Namen dieses Felds.
- *DBMemo* ist die statische Memo-Komponente, die mit dem letzten Memo-Feld der Feldliste verbunden wird
- *MemoLabel* ist die Beschriftung für dieses Feld (sie gibt den Feldnamen an).

- *DBImage* zeigt dementsprechend das zuletzt gefundene Feld der Datenbank an, das Bilddaten enthält
- *GraphicLabel* zeigt den Namen dieses Feldes an.

Zunächst zerlegen wir die Aufgabe, die bei jeder neuen Dateiauswahl durchgeführt wird, in drei Teile: die gegenüber Kapitel 7.2.5 kaum veränderte Methode für das *OnChange*-Ereignis der Dateiliste und die Methoden *Activate* und *Deactivate*:

```
procedure TForm1.FileListBox1Change(Sender: TObject);
begin
  if FileListBox1.FileName <> '' then begin
    Deactivate;
    Table1.DatabaseName := DirectoryListBox1.Directory;
    Table1.TableName := FileListBox1.FileName;
    Activate;
  end;
end;
```

Dynamische Abfrage *R185*

Activate ist die wichtigste Methode, denn sie greift intensiv auf die *TFieldDefs*-Daten zu, die sofort nach dem Setzen des *Active*-Properties auf *True* zur Verfügung stehen und die Struktur der in *FileListBox1Change* gesetzten Tabelle beschreiben.

Activate durchläuft alle Felddefinitionen und hängt in jedem Fall den Namen des Felds an die Kombobox an. Dann überprüft Sie den Feldtyp und verbindet das Feld mit dem *DBImage* bzw. dem *DBMemo*, falls es den passenden Typ aufweist (gibt es mehrere Felder desselben Typs, so überschreibt die Methode die bisher aufgebaute Feldverknüpfung einfach mit dem jeweils letzten Feld des Typs):

```
procedure TForm1.Activate;
var
  i: Integer;
begin
  Table1.Open;
  with Table1 do
    for i := 0 to FieldDefs.Count-1 do begin { für jedes Feld: }
      cbFields.Items.Add(FieldDefs[i].Name);   { Liste erweitern }
      if FieldDefs[i].DataType in [ftGraphic, ftBlob, ftTypedBinary] then
      begin
        { Graphics und Blobs als Grafik darstellen }
        DBImage.DataField := FieldDefs[i].Name;
        GraphicLabel.Caption := FieldDefs[i].Name;
      end
      else if FieldDefs[i].DataType = ftMemo then begin
        { Memofeld gefunden }
        DBMemo.DataField := FieldDefs[i].Name;
        MemoLabel.Caption := FieldDefs[i].Name;
      end;
```

```
    end;
end;
```

Hinweis: Die ClientDataSet-Version des Programms unterscheidet sich vom hier gedruckten Code nur dadurch, dass statt *Table1* die Komponente *ClientDataSet1* verwendet wird.

Lösen der Verbindung

Das vollständige Lösen der in *Activate* aufgebauten Verbindungen besteht nun nicht mehr nur im Abschalten von *Active*, sondern auch im Löschen der *DataField*-Properties der Datensteuerelemente. Dies ist notwendig, damit es beim Öffnen einer anderen Tabelle nicht sofort zu einem Fehler kommt:

```
procedure TForm1.Deactivate;
begin
  Table1.Active := False;
  cbFields.Clear;
  DBImage.DataField := '';
  DBMemo.DataField := '';
  GraphicLabel.Caption := 'kein Grafikfeld gefunden';
  MemoLabel.Caption := 'kein Memofeld gefunden';
end;
```

TDBImage-Methoden

Die *TDBImage*-Komponente erhält im Beispielprogramm noch ein kleines Popup-Menü, in dem Sie zur Laufzeit mit der Zwischenablage arbeiten und das *Stretch*-Flag der Grafik umschalten können (*DBImage.Stretch*). Die Menüpunkte für die Zwischenablage erfordern jeweils nur den Aufruf einer der *DBImage*-Methoden *CopyToClipboard*, *PasteFromClipboard* und *CutToClipboard*, so dass sich ein Listing erübrigen dürfte.

7.3.5 Die Daten des aktuellen Datensatzes

In Kapitel 7.2.3 wurden zwar schon einige Methoden vorgestellt, die den aktuellen Datensatz auswählen, für den eigentlichen Zugriff auf den Datensatz waren jedoch die Datenbank-Komponenten zuständig: Sie lesen den Inhalt des aktuellen Datensatzes automatisch aus und zeigen ihn an. Auch wenn der Benutzer Daten ändert, werden die Änderungen automatisch verarbeitet, sobald er das Feld verlässt oder den *Post*-Schalter drückt. Sie müssen unter Umständen (bei der Verwendung von dbExpress statt BDE) lediglich sicherstellen, dass die Änderungen aus dem Änderungsprotokoll auch in die Datenbank geschrieben werden, brauchen die Daten aber oft nicht durch eigenen Programm-Code zu ändern.

Wenn Sie einzelne Daten durch das Programm auslesen lassen oder Datensätze programmgesteuert erzeugen wollen, benötigen Sie jedoch Zugriff auf den Inhalt des aktuellen Datensatzes. Wie bereits erwähnt, ist dieser auch schon der einzige Datensatz, den Sie gleichzeitig ansprechen können. Zum Ansprechen von mehreren Datensätzen müssen Sie ähnlich vorgehen wie der Benutzer einer Datenbankanwendung: Datenzugriff, einen anderen Datensatz ansteuern und dann auf dessen Daten zugreifen. Allerdings haben Sie im Programm einige zusätzliche Möglichkeiten, wie Lesezeichen zu setzen und die Aktualisierung der Anzeigen zeitweise auszuschalten.

Für den Zugriff auf den Inhalt des aktuellen Datensatzes benötigen wir ein weiteres Bündel von *TField*-Properties.

Value

Zunächst einmal finden Sie in den verschiedenen von *TField* abgeleiteten Typen jeweils ein Property namens *Value*, dessen Typ dem Datentyp des Felds entspricht. Falls Sie z.B. ein Feld des Typs *ftInteger* beschreiben wollen und dementsprechend ein Feld des Typs *TIntegerField* vorliegen haben, hat dessen Property *Value* den Typ Integer.

Wenn Sie mit persistenten Feldern arbeiten, die Sie im Felder-Editor erzeugt haben, können Sie diese besonders einfach über ihre Namen ansprechen, so dass das folgende Beispiel den Wert 16 in ein Feld schreibt:

```
DatenbanknameFeldname.Value := 16;
```

Dabei soll *DatenbanknameFeldname* für den von Delphi automatisch erzeugten Namen des persistenten Felds stehen, den Sie im Objektinspektor ändern können, wie bei jeder anderen Komponente auch.

Auch die weitere Vorgehensweise nach dem Schreiben der Daten entspricht der des Benutzers. Sie können also die *DataSet*-Methode *Post* aufrufen, um die Änderungen sofort zu speichern, oder darauf warten, dass die *DataSet*-Komponente das automatisch tut, wenn der aktuelle Datensatz gewechselt wird.

Value als Variante

Um auf *Value* zugreifen zu können, benötigen Sie kein Objekt einer speziellen *TField*-Klasse, denn auch die abstrakte Klasse *TField* verfügt bereits über ein *Value*-Property, das jedoch keinen speziellen Datentyp hat, sondern eine Variante darstellt. Sie können diesem Property also nahezu beliebige Datentypen zuweisen, und die VCL versucht dann, falls notwendig, diese in den passenden Typ umzuwandeln (zu Varianten siehe auch COM-Automation in Kapitel 8.6.1).

Da das *Fields*-Array einer *DataSet*-Komponente die einzelnen *TField*-Komponenten als *TField*-Objekt zurückgibt, können Sie wie folgt (ohne eine explizite Typenumwandlung) auf das *Value*-Property eines solchen Objekts zugreifen:

```
Table.Fields[i].Value := 16;
```

Formatumwandlungs-Properties

Jedes *TField*-Objekt verfügt über einen Satz von Properties, die es Ihnen ermöglichen, den Feldwert in verschiedenen Datentypen anzugeben und abzufragen: *AsBoolean*, *AsDateTime*, *AsFloat*, *AsInteger* und *AsString*. Die Feldtypen, die beispielsweise in einem *DBGrid* vollständig als Text dargestellt werden können, können auch über *AsString* geändert und gelesen werden. Die meisten Feldtypen können sehr flexibel ineinander umgewandelt werden. Nur in wenigen Fällen müssen Sie mit Problemen rechnen, so etwa wenn Sie versuchen, das Property *AsDateTime* bei einem Feld zu verwenden, das weder ein String- noch ein Datums- oder Zeitfeld ist.

Um auf das obige Beispiel zurückzukommen: Mit *AsInteger* können Sie den Feldwert auch ohne Verwendung einer persistenten *TField*-Komponente und ohne explizite Typenumwandlung und Verwendung des Properties *Value* ansprechen:

```
Table.Fields[Feldindex].AsInteger := 16;
```

Das *FileDB*-Beispiel aus dem nächsten Kapitel gibt Beispiele für die praktische Verwendung der Properties *AsInteger* und *AsString*.

Rufen der dynamischen Felder beim Namen

Die letzte Beispielzeile kann noch etwas lesbarer gestaltet werden. Schwierig zu verstehen ist besonders der *Feldindex*, denn normalerweise werden die Felder bei ihrem Namen genannt.

Tatsächlich können Sie die Felder auch in dieser Situation beim Namen nennen, auch ohne sie persistent im Felder-Editor definiert zu haben. *TDataSet* definiert zu diesem Zweck die Methode *FieldByName*, der Sie den Namen des gewünschten Felds als String übergeben und die das gefundene Feld als Ergebnis des Typs *TField* zurückliefert:

```
Table.FieldByName('Feldname').AsInteger := 16;
```

FieldByName erzeugt eine Exception, wenn der angegebene Feldname nicht existiert. Wenn Sie in diesem Fall lieber das Ergebnis *nil* empfangen wollen, verwenden Sie statt dessen die genauso aufrufbare *TDataSet*-Methode *FindField*.

Es ist noch eine weitere Vereinfachung möglich: Sie können den Feldnamen quasi als Index in eckigen Klammern direkt hinter den Tabellennamen schreiben (damit sprechen Sie *FieldValues* an, das Standard-Array-Property von *TDataSet*). Wie schon bei Ver-

wendung des Properties *TField.Value* erhalten Sie auch hier eine Variante. Zuweisung und Auslesen eines Wertes sehen dann wie folgt aus:

```
Table['Feldname'] := 16;
AusgelesenerWert := Table['Feldname'];
```

Arbeiten in der Tabelle

Wenn die Delphi-Anwendung selbst mehrere Datensätze verändert, zeigt sich ein Nebeneffekt der Tatsache, dass es immer nur einen Datensatz gibt, auf den Programm und Anwender gleichzeitig zugreifen können. Wenn das Programm einen anderen Datensatz auswählt, um in diesem einige Änderungen vorzunehmen, verschwindet auch der für den Benutzer aktuelle Datensatz aus dem Blickfeld, statt dessen zeigen die Datensteuerelemente den vom Programm angesprochenen Datensatz an.

Oft ist es jedoch für den Benutzer angenehmer, von diesen Hintergrundaktivitäten des Programms nicht allzuviel mitzubekommen, und so empfiehlt es sich, bei längeren Arbeiten in der Tabelle die Aktualisierung der Datenanzeige-Komponenten zu unterbinden. Hierzu sieht *TDataSet* die Methoden *DisableControls* und *EnableControls* vor. Mit *DisableControls* trennen Sie alle visuellen Elemente, die mit der Tabelle verbunden sind, kurzfristig von dieser ab, so dass das Programm seinen Auftrag hinter den Kulissen ausführen kann. Danach schalten Sie die Komponenten mit *EnableControls* wieder an.

Ein wichtiger Vorteil dabei ist natürlich auch, dass die Aktualisierung einer Vielzahl von Datensätzen spürbar schneller abläuft, wenn die Tabellen im Formular nicht bei jedem Datensatz-Wechsel aktualisiert werden müssen. Im Beispielprogramm *FileDB* (nächstes Kapitel) können Sie diesen Geschwindigkeitsunterschied bei laufender Operation testen, indem Sie die Aktualisierung der Komponenten über einen Markierungsschalter manuell an- und abschalten.

7.3.6 Eine Dateidatenbank

Ein Beispiel für eine Datenbankanwendung, die ausgiebig selbst Datensätze erstellt, ist *FileDB* (Abbildung 7.15). Sie liest komplette Verzeichnisbäume mitsamt den enthaltenen Dateien ein und legt diese in einer Datenbank ab, die aus zwei über eine Relation verknüpften Tabellen besteht:

In der Dateitabelle sollen zu jeder Datei neben dem Namen (Feld *Dateiname*) auch die *Größe* und das *Datum* (der Zeitstempel des Betriebssystems) gespeichert werden. Wichtig ist natürlich auch das Verzeichnis, in dem sich die Datei befindet. Da Verzeichnisse über hundert Dateien haben und Verzeichnispfade sehr lang werden können, wäre es eine große Platzverschwendung, für jede Datei ein eigenes, 80 Zeichen langes Feld für

den Verzeichnispfad zu reservieren. Besser ist es, eine zweite Tabelle zu definieren, die jedes Verzeichnis genau einmal aufführt und mit einer Nummer versieht.

Die Verzeichnistabelle enthält nur zwei Felder: Verzeichnisnummer (Integer) und Verzeichnispfad (vollständige Verzeichnisangabe als String). Da der Pfad jedes Verzeichnisses in dieser Tabelle nachgeschlagen werden kann, beschränkt sich die Angabe des Verzeichnisses in der Dateitabelle einfach auf die Nennung der Verzeichnisnummer.

Die folgende Tabelle listet die Namen der Felder auf, wie sie auf Datenbankebene definiert wurden, und durch welche persistenten Feldkomponenten sie in der Delphi-Anwendung repräsentiert werden:

Tabelle Dirs	**Tabelle Files**
DirID (DirsDirID: TIntegerField)	**DirID** (FilesDirID: TIntegerField)
Path (DirsPath: TStringField)	**FileName** (FilesFileName: TStringField)
	FileTime (FilesFileTime: TSQLTimeStampField)
	FileSize (FilesFileSize: TIntegerField)

Auf der CD finden Sie zwei Versionen dieses Programms: Das Projekt *FileDB* greift mit Hilfe der BDE auf zwei im Programmverzeichnis gespeicherte Paradox-Tabellen zu, das Projekt *IBFileDB* verwendet dbExpress für den Zugriff auf eine ebenfalls im Programmverzeichnis gespeicherte Interbase-Datenbank. Wir werden im Folgenden nur die BDE-Version der Anwendung (*FileDB*) besprechen. Die Unterschiede in *IBFileDB* halten sich in Grenzen. Wichtig ist hier vor allem, dass in *IBFileDB* das Änderungsprotokoll auf die Datenbank übertragen werden muss. Hierfür enthält die unten abgedruckte Methode *AddFilesClick* in *IBFileDB* je einen *ApplyUpdates*-Aufruf für die beiden Tabellen. Die Funktion der beiden Anwendungsvariationen ist ansonsten gleich. In *IBFileDB* ist als Extra noch die Option enthalten, die Interbase-Datenbank in einem beliebigen Verzeichnis zu erzeugen (siehe hierzu auch die Erzeugung der Interbase-Datei in Kapitel 7.1.5.

Um die beiden benötigten Paradox-Tabellen manuell mit der Datenbankoberfläche zu erzeugen, wären die folgenden Schritte notwendig: Geben Sie für die erste Beispieltabelle die in Abbildung 7.1 gezeigten Daten ein und markieren Sie dabei in der Spalte *Schlüssel* auch die ersten beiden Felder. Drücken Sie danach den Schalter *Speichern unter...* und speichern Sie die Datei unter dem Namen *files.db*. Für die zweite Tabelle *dirs.db* tun Sie das Gleiche, wobei Sie diesmal ein Integer-Feld mit dem Namen *DirID* und ein Stringfeld mit dem Namen *Path* definieren; *DirID* dient als Schlüssel. Für das Beispielprogramm müssten Sie die Tabellen in einem weiteren Schritt noch indizieren, dies ist jedoch erst Thema in Kapitel 7.4.2. Das vollständige Beispielprogramm *FileDB* kann die Verzeichnis- und die Dateitabellen selbst anlegen (Menüpunkt DATENBANK | NEU).

Version 1

Die erste Version der Methoden, die hier abgedruckt ist, unterscheidet sich allerdings noch deutlich von der Endversion. Ohne die in Kapitel 7.4.4 hinzugefügten Verbesserungen eignet sich die Datenbank nur zum Einlesen der Verzeichnis- und Dateidaten in eine völlig neue Tabelle, denn sie berücksichtigt in keiner Weise schon vorhandene Einträge, die unter Umständen nur aktualisiert zu werden bräuchten. Wenn Sie mit dieser Version versuchen würden, eine bestehende Datenbank zu aktualisieren, käme es häufig zu Indexfehler-Exceptions, denn das Programm würde versuchen, schon existierende Schlüssel (bestehend aus Verzeichnisnummer und Dateiname) zu verdoppeln. Die notwendigen Erweiterungen können erst hinzugefügt werden, nachdem Kapitel 7.4.2 das Thema der Indizes erläutert hat.

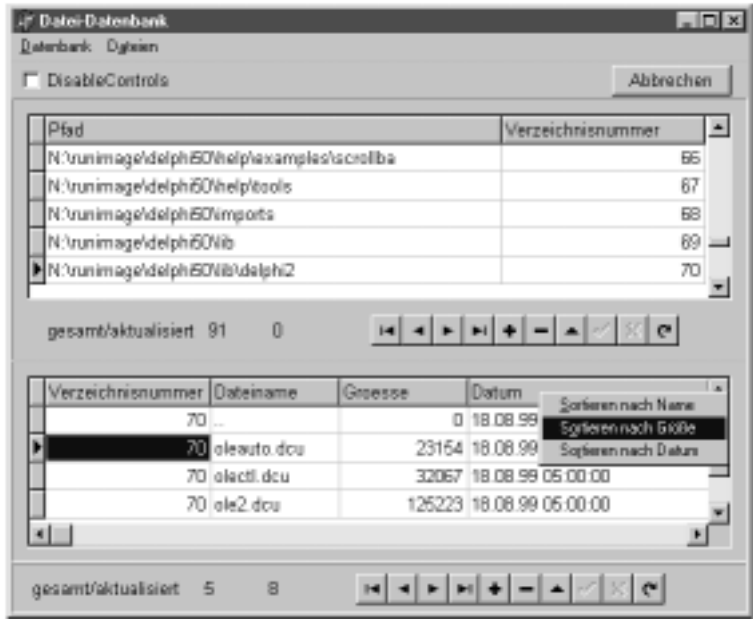


Abbildung 7.15: Das Beispielprogramm FileDB

Das Formular

Das Hauptformular von *FileDB* wird wie folgt entworfen:

- Es wird von vier Panels ausgefüllt, von denen die beiden ersten mit *alTop*, die letzte mit *alBottom* und die vorletzte mit *alClient* ausgerichtet ist. Dies hat zur Folge, dass Größenänderungen des Fensters immer zu Gunsten oder zu Lasten des zweiten *DBGrids* gehen, das sich in diesem dritten Panel befindet.

- Je zwei Exemplare von *TTable*, *TDataSource*, *TDBGrid* und *TDBNavigator* sorgen dafür, dass die beiden Tabellen unabhängig voneinander angesprochen werden können.
- Die Properties der *TTable*-Komponenten *Files* und *Dirs* werden auf die beiden Tabellen `files.db` und `dirs.db` gesetzt, so dass zur Laufzeit nur noch das *Active*-Property eingeschaltet werden muss, um die Tabellen zu aktivieren. Falls die Tabellen noch nicht existieren, können Sie sie zur Laufzeit mit dem Menüpunkt DATENBANK | NEU erzeugen. Die beiden Tabellen sind zu einem Haupt-/Detailgespann verbunden, das jedoch erst in Kapitel 7.4.5 Bedeutung erlangt.
- Im Felder-Editor werden alle Felder der Tabellen als persistente Felder definiert, so dass wir diese im Quelltext über einen festen Namen ansprechen können (die im Beispielprogramm gültigen Namen wurden in der Einleitung bereits in einer Tabelle aufgelistet).
- Die weiteren Bestandteile und Eigenschaften des Formulars erschließen sich von selbst, bis auf einige Details, die in späteren Kapiteln besprochen werden (im Zusammenhang mit der Aufteilung in Haupt- und Detail-Tabelle).

Wir beginnen mit den wichtigsten Methoden des Formulars, und zwar denen, die die eigentliche Erweiterung der Datenbank übernehmen (*StoreFiles* und *StoreDirs*), und sehen uns erst danach an, wie die Suche der Dateien abläuft und wie die beiden Methoden aufgerufen werden.

Programmgesteuertes Erzeugen von Datensätzen *R186*

StoreFile erhält als Parameter den Namen, die Größe und den Zeitstempel einer Datei sowie die Nummer des Verzeichnisses, in dem sie gespeichert ist. Sie fügt einen neuen Datensatz in die Tabelle *Files* ein, setzt die einzelnen Felder über die in Kapitel 7.3.5 besprochenen *As...*-Properties auf die in den Parametern erhaltenen Werte und speichert den Datensatz mit *Post*:

```
procedure TMainForm.StoreFile(Dir: LongInt; Name: string;
  Size: LongInt; Date: TDateTime);
{ gekürzte Version }
begin
  Files.Insert;
  FilesFileTime.AsDateTime := Date;
  FilesFileName.AsString := Name;
  FilesDirID.AsInteger := Dir;
  FilesFileSize.AsInteger := Size;
  Files.Post;
end;
```

StoreDir erhält einen Verzeichnispfad als Parameter, gibt diesem eine Nummer, speichert Nummer und Pfad in der Tabelle *Dirs* und liefert die Nummer als Ergebnis zurück. (Zur Vergabe der Verzeichnisnummer kommen wir in Kapitel 7.4.4, für diese gekürzte Version würde es genügen, die bisher letzte Nummer in einer globalen Variablen zu speichern und in jedem *StoreDir*-Aufruf um eins zu erhöhen.)

```
function TMainForm.StoreDir(Path: string): LongInt;
{ gekürzte Version }
begin
  Dirs.Insert;
{    DirIndex := MaxDirIndex;   siehe vollständige Version }
  DirsDirID.AsInteger := DirIndex;
  inc(DirIndex)
  DirsPath.AsString := Path;
  Dirs.Post;
end;
```

Einlesen der Verzeichnisse

Zum Starten des Verzeichnis-Einlesevorgangs benötigen wir zuerst einmal ein Ereignis. *FileDB* stellt dazu den Menüpunkt DATEIEN | HINZUFÜGEN bereit, der in der folgenden Methode bearbeitet wird:

```
procedure TMainForm.AddFilesClick(Sender: TObject);
{ vollständige Version }
begin
  if DirSelect.ShowModal = mrOK then begin
    AddFromDir(DirSelect.DirectoryListBox1.Directory);
  end;
end;
```

Diese Methode ruft eine hier nicht weiter interessante Dialogbox auf, aus der sie das vom Benutzer ausgewählte Verzeichnis ausliest und es an die Methode weitergibt, die die eigentliche Arbeit erledigt: *AddFromDir*. Diese führt einige Verwaltungsaufgaben durch, die später in Kapitel 7.3.7 beschrieben werden, und delegiert das gesamte Einlesen an die Methode *InsertDir*, die den Pfad des Verzeichnisses ebenfalls als Parameter erhält:

```
procedure TMainForm.InsertDir(Path: string);
{ vollständige Version }
var
  SearchRec: TSearchRec;
  Error: Integer;
  DirIndex: LongInt;
  DateTime: TDateTime;
begin
  Error := FindFirst(IncludeTrailingPathDelimiter(Path)+Mask,
                     faAnyFile, SearchRec);
  SearchRec.Name := LowerCase(SearchRec.Name); { nur Kleinbuchstaben }
```

```
    DirIndex := StoreDir(Path); { * Neuer Datensatz * }
    Count.Caption := IntToStr(DirIndex);
    while (Error = 0) and not CancelRequest do begin
      UpdateLabels; { Statusanzeige über den Einlesevorgang erneuern }
         { UpdateLabels: siehe vollständige Datei auf CD-ROM }
      { nur Kleinbuchstaben: }
      SearchRec.Name := LowerCase(SearchRec.Name);
      DateTime := 0; { manche Dateien (z.B. verschiedene ".."-Einträge }
      try            { haben kein Datum. SearchRec.Time ist dann 0 }
                     { und FileDateToTime erzeugt eine Exception }
        DateTime := FileDateToDateTime(SearchRec.Time)
      except       { ... diese Exception einfach ignorieren. }
      end;
      { * Neuer Datensatz: * }
      StoreFile(DirIndex, SearchRec.Name, SearchRec.Size, DateTime);
      if (SearchRec.Name <> '.') and (SearchRec.Name <> '..') then
         { Die Verzeichnisse "." und ".." nicht beachten }
        if (SearchRec.Attr and faDirectory) <> 0 then
          InsertDir(IncludeTrailingPathDelimiter(Path)+SearchRec.Name);
      Error := FindNext(SearchRec);
    end;
    FindClose(SearchRec);
  end;
```

InsertDir speichert das gewählte Verzeichnis mit der Methode *StoreDir*, die die Nummer des Verzeichnisses zurückliefert. Diese Nummer kann dann für jede der im Verzeichnis enthaltenen Dateien an die Methode *StoreFile* übergeben werden. *InsertDir* findet diese Dateien mit Hilfe der Standardfunktionen *FindFirst* und *FindNext* (siehe Kapitel 2.8.1).

Wenn es sich bei der mit *FindFirst* oder *FindNext* gefundenen Datei um ein Verzeichnis handelt (*SearchRec.Attr* enthält dann das Bit *faDirectory*), ruft *InsertDir* sich selbst rekursiv auf, damit auch dieses Unterverzeichnis vollständig in die Datenbank aufgenommen wird.

7.3.7 Zusatzfunktionen für die Anwendung

Die Anwendung *FileDB* soll, während sie einen Verzeichnisbaum einliest, auch Ereignisse des Benutzers verarbeiten können. Zwar soll der Benutzer während dieses Vorgangs keine der Tabellen verändern können, aber zwei Dinge sollen ihm erlaubt sein:

- An- und Abschalten der laufenden Aktualisierung der *DBGrid*-Komponenten über den Markierungsschalter *DisableControls*.
- Und, was noch wichtiger ist, er soll den Einlesevorgang abbrechen können.

Für beides muss der Benutzer einen (Markierungs-)Schalter drücken. Das Programm kann dieses Drücken des Schalters jedoch nicht bearbeiten, wenn es gleichzeitig bereits die Dateien einliest (dieses Einlesen der Dateien ist ja bereits die Bearbeitung eines Ereignisses – und zwar des *OnClick*-Ereignisses für den Menüpunkt DATEIEN | HINZUFÜGEN...).

ProcessMessages *R187*

Damit der Benutzer die Einlesefunktion dennoch jederzeit abbrechen kann und auch das Drücken des *DisableControls*-Schalter sofort Wirkung zeitigt, muss das Programm quasi mit sich selbst Multitasking durchführen – es wird den Einlesevorgang in kurzen Intervallen unterbrechen, indem es die Methode *Application.ProcessMessages* aufruft, während der alle inzwischen angefallenen Nachrichten bearbeitet werden.

Ausführliche Informationen zu *ProcessMessages* und zum Multitasking allgemein gibt Kapitel 4.7. Wir verwenden hier keine Threads, da sonst wieder das Problem der Synchronisierung mit der VCL auftauchen würde (die ständige Verwendung der *Synchronize*-Methode würde auf das hinauslaufen, was wir auch mit *ProcessMessages* erreichen können: dass der Haupt-Thread abwechselnd Nachrichten bearbeitet und den Verzeichniseinlesevorgang fortsetzt).

Der Aufruf von *Application.ProcessMessages* findet in der vollständigen Methode *StoreFiles* (Kapitel 7.4.4) statt, also je einmal pro Datei, die der Datenbank hinzugefügt wird. Diese Häufigkeit stellt sicher, dass die Anwendung eingehende Nachrichten quasi sofort bearbeitet kann.

Absichern des Einlesevorgangs vor störenden Ereignissen

Der Benutzer kann nun also, während die Verzeichnisse eingelesen werden, Eingaben an das Formular senden. Der Aufruf bestimmter Menüpunkte würde jedoch das Einlesen massiv stören (z.B. der Befehl DATENBANK | NEU). Alle Kontrollelemente und Menüpunkte, die beim Einlesen nicht verwendet werden dürfen, sollten daher vorher deaktiviert werden. Dies ist Aufgabe der Methode *AddFromDir*.

Eine Möglichkeit, jegliche Eingabe in das Formular zu verhindern, wäre, es vollständig zu deaktivieren (Property *TForm.Enabled*). Dann könnten Sie jedoch auch die Schalter ABBRUCH und DISABLECONTROLS nicht mehr drücken.

Abbrechen des Vorgangs

Ein langwieriger Vorgang wie das Einlesen mehrerer tausend Dateien in eine Datenbank wird zwar oft als Hintergrundoperation implementiert, so dass er das System nicht mehr blockiert, trotzdem sollte er auch abgebrochen werden können. *FileDB* stellt hierzu den Schalter ABBRECHEN zur Verfügung, dessen *OnClick*-Methode die interne

Formularvariable *CancelRequest* auf *True* setzt. Die oben gezeigte Methode *InsertDir* fragt diese Variable nach jeder einzelnen eingefügten Datei ab und bricht den Einlesevorgang sofort ab, wenn sie den Wert *True* annimmt.

Die Methode AddFromDir

Die Methode *AddFromDir* sorgt also dafür, dass vor dem Einlesen der Verzeichnisse alle kritischen Komponenten deaktiviert werden. Der *try...finally*-Block stellt sicher, dass sie selbst dann wieder aktiviert werden, wenn es zu einer Exception kommt, durch die die Ausführung der Methode abgebrochen wird:

```
procedure TMainForm.AddFromDir(Dir: string);
{ vollständige Version }
var
  i: Integer;
begin
  { Datenbankverbindung einschalten }
  Files.Active := True;
  Dirs.Active := True;
  { Menüpunkte deaktivieren }
  ClearDB.Enabled := False;
  CreateDB.Enabled := False;
  AddFiles.Enabled := False;

  Count.Visible := True;   { Zähler anzeigen }
  { internes Flag und Zähler setzen }
  CancelRequest := False;
  FileUpdates := 0;   DirUpdates := 0;
  { Synchronisation von Haupt-/Detailtabelle abschalten }
  Files.MasterSource := nil; { MasterSource siehe 7.4.5 }
  try
    LowerCase(Dir);
    InsertDir(Dir);
  finally
    CancelRequest := False;
    ClearDB.Enabled := True;
    CreateDB.Enabled := True;
    AddFiles.Enabled := True;
    Count.Visible := False;
    Files.MasterSource := DirSource;
  end;
end;
```

Ausschalten der Aktualisierung

Das Beispielprogramm führt das Erweitern der beiden Tabellen für den Benutzer sichtbar durch, das heißt, dass Sie in der Tabelle immer den gerade hinzugefügten Datensatz sehen könnten, wenn dieser nicht so schnell von immer neuen Datensätzen wieder außer Sichtweite geschoben werden würde.

Wie schon erwähnt, können Sie diese Aktualisierung im Programm über die Methoden *DisableControls* und *EnableControls* ein- und ausschalten. *FileDB* gibt diese Aufgabe an den Benutzer weiter und stellt ihm den Markierungsschalter *DisableControls* zur Verfügung, mit dem Sie die Aktualisierung beim laufenden Einlesevorgang beliebig ein- und ausschalten können:

```
procedure TMainForm.DisableControlsClick(Sender: TObject);
begin
  { Da DisableControls ein Markierungsschalter ist (und kein Menüpunkt),
    muss das Property Checked nicht von Hand gesetzt werden. }
  if DisableControls.Checked then begin
    Files.DisableControls;
    Dirs.DisableControls;
  end else begin
    Files.EnableControls;
    Dirs.EnableControls;
  end;
end;
```

Während des Einlesevorgangs zeigt das Programm die Zahl der hinzugefügten Dateien und Verzeichnisse an (Aufruf von *UpdateLabels* in der Methode *InsertDir*), daher können Sie den Geschwindigkeitsunterschied zwischen beiden Modi leicht erkennen.

7.4 Sortieren, Suchen und Filtern

Während das Kapitel 7.3 ganz im Zeichen der Felder stand, bringt dieses Kapitel als wesentliche Neuerung die Indizes, die dazu dienen, die elementaren Datenbankoperationen wie Suchen, Sortieren und Filtern zu beschleunigen. In Kapitel 7.4.1 verlassen wir uns zunächst auf die automatische Handhabung der Indizes durch die VCL, die BDE bzw. durch *TClientDataSet*, bevor wir in Kapitel 7.4.2 explizit Gebrauch von Indizes machen, um Tabellen zu sortieren. Auch die in Kapitel 7.4.3 vorgestellten Methoden erfordern, dass Indizes explizit ausgewählt werden.

Gültigkeitsbereich der Methoden bei Client-Datenmengen

Zuerst muss aber auf eine generelle Einschränkung der in diesem Kapitel erwähnten Such- und Filtermethoden hingewiesen werden, falls sie auf Client-Datenmengen angewendet werden, wie sie etwa bei dbExpress eingesetzt werden (wenn Sie nur die BDE-Komponenten *TTable* und *TQuery* verwenden, können Sie diesen Abschnitt überspringen!): Sie beziehen sich nicht auf die Datenbank, wie sie vom Server verwaltet wird, sondern auf die Datenmenge, wie sie der Delphi-Anwendung erscheint. In dbExpress werden diese Methoden nur von den Client-Datenmengen wie *TClientDataSet*,

TSQLClientDataSet unterstützt, die grundsätzlich die gesamte Datenmenge, auf die die genannten Methoden wirken, im Hauptspeicher halten.

Während in den Browser-Beispielprogrammen von Kapitel 7.2 und in der Dateidatenbank aus Kapitel 7.3.6 jeweils ganze Tabellen vollständig in eine Client-Datenmenge (also in den Hauptspeicher) geladen werden, enthalten Client-Datenmengen in der Praxis häufig nur einen Ausschnitt aus der Datenbanktabelle (Abbildung 7.16).

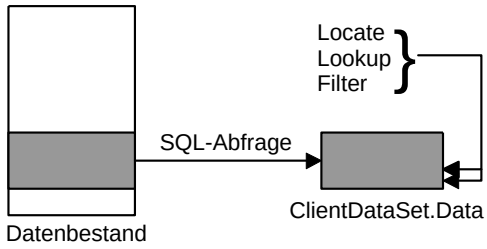


Abbildung 7.16: Wirkungsbereich der Such- und Filtermethoden in einem ClientDataSet

Dieser in einer Client-Datenmenge sichtbare Ausschnitt kann

- das Ergebnis einer expliziten Abfrage sein (in Kapitel 7.5.2 finden Sie Beispiele, bei denen zwar gleich Daten aus mehreren Tabellen kombiniert werden, im Prinzip handelt es sich aber auch dabei nur um einen Ausschnitt, und zwar um einen Ausschnitt des Gesamt-Datenbestandes),
- oder implizit beispielsweise durch die Definition einer *MasterSource* bestimmt werden (siehe Kapitel 7.4.5).

Bei allen in diesem Kapitel erwähnten Methoden zum Suchen und Filtern sollten Sie im Hinterkopf behalten, dass sich diese nur auf diesen in der Client-Datenmenge dargestellten Ausschnitt beziehen. Wenn Sie also etwa die gesamte Datenbanktabelle nach einem bestimmten Datensatz durchsuchen wollen, müssen Sie dies entweder mit einer SQL-Abfrage machen oder Sie müssen ein ClientDataSet verwenden, in dem die gesamte Tabelle geladen ist (für MyBase-Dateien: diese werden immer komplett in das ClientDataSet geladen; für SQL-Tabellen: setzen Sie *CommandType* auf *ctTable* und *CommandText* auf den Namen der Tabelle, um die gesamte Tabelle zu laden).

7.4.1 Methoden zum Suchen, Filtern und für Lesezeichen

Dieses Kapitel beschreibt einige Methoden, die bereits in der Klasse *TDataSet* definiert sind und die auch in den meisten konkreten Datenmengen-Komponenten implementiert werden (z.B. den BDE-Datenmengen *TTable* und *TQuery* sowie den Client-Datenmengen). In den neuen unidirektionalen Datenmengen von dbExpress können diese

Methoden allerdings nicht verwendet werden (der Aufruf dieser Methoden führt bei diesen Datenmengen zu einer Exception oder im Falle von *GetBookmark* zu einem undefinierten Ergebnis).

Suchen mit Locate

Die Methode *Locate* ist eine sehr flexible Suchfunktion, die mit nur drei Parametern auskommt:

- Im ersten Parameter geben Sie die Namen der Felder an, in denen gesucht werden soll. Wenn Sie mehrere Felder angeben, trennen Sie diese durch Semikola.
- Im zweiten Parameter geben Sie die Werte an, nach denen gesucht werden soll. Dieser Parameter hat den Typ *Variant*. Das bedeutet, dass Sie, wenn Sie nur in einem Feld suchen, einen Wert eines beliebigen Typs angeben können, der zum Suchfeld passt (für ein numerisches Feld können Sie z.B. einen String angeben, der eine Zahl enthält). Wenn Sie in mehreren Feldern suchen, übergeben Sie diese in einem Varianten-Array (siehe Beispiel unten).
- Im dritten Parameter geben Sie eine Menge an, die zwei Optionen enthalten kann: Mit *loCaseSensitive* beachtet die Suchfunktion Groß- und Kleinschreibung in Strings, mit *loPartialKey* interpretiert sie die Suchwerte nur als Teilschlüssel, findet also alle Datensätze, die mit dem angegebenen Schlüssel beginnen.

Locate gibt *True* zurück, wenn sie einen Datensatz mit den angegebenen Kriterien finden konnte. In diesem Fall setzt sie die aktuelle Position auf diesen Datensatz, so dass Sie seine anderen Felder sofort auslesen können. Zwei Beispiele für *Locate*-Aufrufe sind:

```
Table.Locate('Nummer', 100, []);
Files.Locate('DirID;FileName', VarArrayOf([3, 'A']), [loPartialKey]);
```

Die erste Anweisung sucht im Feld *Nummer* nach dem konstanten Wert 100 (dieser Wert wird automatisch in eine Variante umgewandelt), die zweite Anweisung sucht einen Datensatz, dessen Feld *Vznr* den Wert 3 aufweist und dessen Feld *Dateiname* mit dem Buchstaben »A« beginnt (da *loPartialKey* gesetzt ist). Ein Praxisbeispiel zu *Locate* finden Sie in Kapitel 7.4.4, Informationen zur Funktionsweise von Varianten in Kapitel 8.6.1.

Die Vorteile von *Locate* sind, dass sie sich selbst um die Aktivierung der notwendigen Indizes kümmert und dass sie auch funktioniert, wenn keine passenden Indizes verfügbar sind (wenn sie dann auch nicht so schnell ist ...).

Hinweis: Die *Locate*-Funktion einer *TTable* sucht nur in dem aktuellen Bereich der Tabelle. Wenn Sie die Tabelle nach bestimmten Kriterien filtern oder wenn es sich um eine Detail-Tabelle handelt, die mit einer Master-Tabelle verbunden ist, werden nur die Datensätze durchsucht, die auch für den Benutzer noch sichtbar wären. Dieses Verhalten gleicht damit dem eines ClientDataSets, in dem sich die Suchfunktion nur auf den im ClientDataSet geladenen Ausschnitt der Daten bezieht.

Lookup

Die Methode *Lookup* sucht ebenso flexibel in einer oder mehreren Spalten, allerdings verändert sie nicht die aktuelle Position der Tabelle, sondern liest nur Teile des gefundenen Datensatzes aus und gibt sie zurück. Welche Teile (Felder) des gefundenen Datensatzes Sie erhalten wollen, geben Sie im letzten Parameter an. *Lookup* liefert das Ergebnis in Form einer Varianten, bei mehreren Rückgabefeldern in einem Varianten-Array:

```
Pfad := Dirs.Lookup('DirID', GesuchteNummer, 'Path');
```

Dieser Aufruf sucht die *GesuchteNummer* in der Spalte *DirID* der Tabelle *Dirs* und gibt den Inhalt des Feldes *Path* aus dem gefundenen Datensatz zurück. *Lookup* führt damit prinzipiell genau die Funktion durch, die ein Lookup-Feld (siehe Kapitel 7.3.2) bei jeder Änderung des aktuellen Datensatzes durchführen muss. Allerdings haben Sie bei der Methode *Lookup* die Möglichkeit, mehrere Rückgabefelder anzugeben (siehe hierzu die Online-Hilfe).

Filtern

Beim Filtern einer Tabelle werden *mehrere* Datensätze gesucht, die bestimmten Kriterien genügen. Filterkriterien können in der Praxis erheblich komplexer sein als die oben besprochenen Suchkriterien (beispielsweise können sie »<=« - und »>=« -Bedingungen enthalten). Dieser Komplexität wäre selbst ein Variantenparameter kaum gewachsen, daher gibt es zum Filtern keinen einfach zu handhabenden Methodenaufruf wie für das Suchen. Statt dessen müssen Sie selbst eine Methode für das Ereignis *OnFilterRecord* schreiben und in dieser den aktuellen Datensatz der im Parameter erhaltenen Tabelle auf die Kriterien hin untersuchen. Die folgende Methode könnte beispielsweise alle Dateien herausfiltern, die größer als 200000 Byte sind:

```
procedure TForm1.Table1FilterRec(DataSet: TDataSet; var Accept: Boolean);
begin
  Accept := DataSet['Groesse'] > 200000;
end;
```

Wenn sie eine solche Methode mit Ihrer *DataSet*-Komponente verknüpft haben, müssen Sie die Filterung noch einschalten, indem Sie das *DataSet*-Property *Filtered* auf *True* setzen. Dadurch werden so lange nur noch die Datensätze sichtbar und ansprechbar,

die den Kriterien entsprechen, bis Sie *Filtered* wieder abschalten. Sie können also alle gefundenen Datensätze durchlaufen, indem Sie mit *DataSet.First* an den Anfang der Datenmenge gehen und dann mit der Methode *Next* jedes einzelne Element besuchen, bis *EOF=True*.

Lesezeichen

Ein Effekt der Methode *Lookup* ist, dass sie die im Formular angezeigte Position in der Tabelle ändert. Auch in anderen Situationen kann ein kurzzeitiger Wechsel zu einem anderen Datensatz erforderlich sein, nach dem der vorherige Datensatz wieder aktiviert werden soll.

Da Datensätze nicht über eine absolute Position angesprochen werden sollten, gibt es die Möglichkeit, die aktuelle Position mit Hilfe der Lesezeichen-Methoden wiederherzustellen. Sie können ein Lesezeichen immer nur an der aktuellen Position unterbringen, daher fällt der Aufruf der Methode *GetBookMark* recht kurz aus:

```
var
  BookMark: TBookMark;
begin
  BookMark := Tabelle.GetBookMark;
```

GetBookmark liefert eine Variable des Typs *TBookmark* zurück. Was es mit diesem auf sich hat, brauchen Sie nicht zu wissen, denn der einzige Zweck der *TBookmark*-Variable ist es, später an die Funktion *GoToBookmark* übergeben zu werden, um an die Position des Lesezeichens zurückzukehren:

```
VeraendereTabelle(Tabelle);
Tabelle.GotoBookmark(BookMark);
```

Da *GoToBookmark* das Lesezeichen nicht wieder auflöst, können Sie es beliebig oft verwenden, um später erneut an dieselbe Position zurückzukehren. Falls Sie das Lesezeichen aber nur temporär benötigen, sollten Sie den Speicher, den die VCL dafür reservieren musste, mit *FreeBookmark* wieder freigeben:

```
Tabelle.FreeBookmark;
```

7.4.2 Sortieren mit Indizes

Ein Index sortiert die Datensätze einer Tabelle nach einem oder mehreren Feldern in aufsteigender oder absteigender Reihenfolge. Der Index enthält nicht die kompletten Datensätze, sondern gibt nur an, an welcher Position der Tabelle sie zu finden sind und in welcher Reihenfolge sie innerhalb des Indexes angeordnet sind. Durch einen Index ist es möglich, einen gewünschten Wert viel schneller zu finden als ohne Index, wobei der Vorteil des Indexes mit der Anzahl der Datensätze wächst (normalerweise überproportional stark). Während die Datenbank-Software einen gesuchten Feldwert in

einem Index z.B. mit binärer Suche in wenigen Schritten finden kann, müsste sie ohne den Index jeden Datensatz einzeln vergleichen, was schon bei kleinen Tabellen um ein Vielfaches länger dauern kann.

Zusammengesetzte Indizes

Ein Index muss sich nicht nur nach einem einzelnen Feld richten, er kann sich auch aus mehreren Feldern zusammensetzen. Dabei werden die Datensätze zuerst nach dem ersten Indexfeld sortiert. Gibt es in dieser Spalte doppelte Werte, werden die Datensätze, die hier den gleichen Wert aufweisen, nach dem zweiten Indexfeld sortiert. Dieser Vorgang setzt sich bis zum letzten Indexfeld fort.

Wenn Sie beispielsweise den Index »Wohnort;Nachname;Name« in einer Adressdatenbank definieren, wird dadurch die gesamte Tabelle nach dem Wohnort sortiert. Bei dieser Sortierung lässt sich eine aufsteigende Reihenfolge der Nachnamen nur innerhalb desselben Orts finden. Wenn dann derselbe Nachname in einem Ort mehrmals vorkommt, werden selbst diese Datensätze noch nach Vornamen sortiert.

Indizes bei verschiedenen Tabellentypen

Die Indizes für die in diesem Buch berücksichtigten Tabellentypen werden zwar innerhalb einer Delphi-Anwendung auf die gleiche Weise angesprochen (z.B. über die weiter unten besprochenen Properties *IndexName* und *IndexFieldNames*), jedoch intern unterschiedlich verwaltet:

Bei dBase- und Paradox-Tabellen liegt ein Index immer in Form einer eigenen Datei vor. Die von Delphi unterstützten Indizes werden immer auf dem neuesten Stand gehalten, was bedeutet, dass der Index bei jeder Änderung an der Tabelle, die die Sortierung verändert, aktualisiert wird. Interbase speichert seine Indizes dagegen zusammen mit allen Tabellen der Datenbank in einer einzigen Datei. Auch ClientDataSets arbeiten mit Indizes, allerdings verwalten sie diese wie die eigentlichen Daten auch im Hauptspeicher; beim Speichern in eine MyBase-Datei werden für die Indizes keine zusätzlichen Dateien angelegt.

Während Sie für dBase-, Paradox- und Interbase-Tabellen Indizes selbst anlegen müssen, falls Sie effizient in diesen Tabellen suchen und filtern wollen, legen Client-Datenmengen ihre Indizes bei Bedarf automatisch im Hauptspeicher an. Während die Definition von Interbase-Indizes bereits im SQL-Skript von Kapitel 7.1.3 demonstriert wurde, soll im Folgenden kurz skizziert werden, wie Sie mit der Datenbankoberfläche Indizes für Paradox- und dBase-Tabellen erzeugen können.

Paradox- und dBase-Tabellen haben unterschiedliche Indexeigenschaften. So ist es bei Paradox möglich, einen Primärindex zu definieren. Dieser Primärindex sortiert die Datensätze direkt innerhalb der Tabelle, benötigt also keine eigene Datei. Der Primär-

index muss immer aus der ersten oder den ersten Spalten der Tabelle bestehen. Schließlich haben Primärindizes keinen Namen, weshalb sie im Delphi-Programm mit einem leeren String bezeichnet werden. Die normalen, in getrennten Dateien vorliegenden Indizes heißen bei Paradox *Sekundärindizes*.

Ein weiteres Unterscheidungsmerkmal der Indizes ist, ob es sich um einen gepflegten, das heißt bei jeder Tabellenänderung aktualisierten, oder um einen ungepflegten Index handelt. Delphi unterstützt nur die gepflegten dBase- und Paradox-Indizes.

Erzeugen eines Indexes mit der Datenbankoberfläche

In der Datenbankoberfläche können Sie Indizes dialoggesteuert erzeugen, ausgehend von einer Dialogbox der schon in Abbildung 7.1 gezeigten Art. Sie können die Indizes bereits beim Erstellen der Tabelle oder nachträglich festlegen. Im letzteren Fall wählen Sie den Menüpunkt TABELLE | UMSTRUKTURIEREN... und gelangen in eine geringfügig modifizierte Dialogbox.

Bei einer Paradox-Tabelle gehen Sie dann in beiden Fällen wie folgt vor: Wählen Sie im Feld *Tabelleneigenschaften* die Option *Sekundärindizes* und drücken Sie den darunter erscheinenden Schalter DEFINIEREN. Dieser führt Sie zu der in Abbildung 7.17 gezeigten Dialogbox, in der Sie die zu indizierenden Felder von der rechten in die linke Liste befördern und eventuell umordnen. Für jeden neu erstellten Index drücken Sie *OK*, woraufhin Sie die Datenbankoberfläche nach einem Namen für den Index fragt.

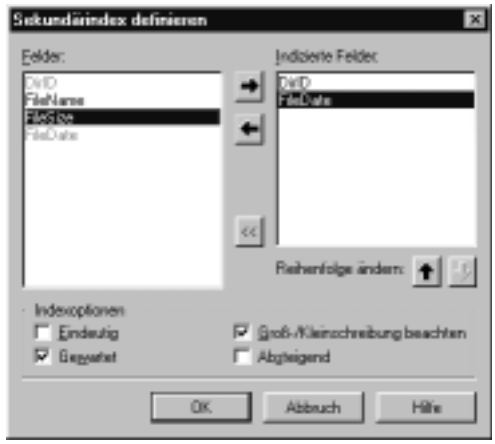


Abbildung 7.17: Definieren eines Indexes in der Datenbankoberfläche

Der in der Abbildung gezeigte Beispielindex sortiert die Tabelle zuerst nach der Verzeichnisnummer und dann nach dem Datum der Datei; er wird in Kapitel 7.4.5 verwendet. Wie Sie einen Index im Programm erzeugen können, können Sie dem

Beispielprogramm *FileDB*-Projekts entnehmen. In der Methode für den Menüpunkt DATENBANK | NEU werden mit der neuen Tabelle auch die oben erwähnten Indizes definiert. Diese Methode wurde bereits in Kapitel 7.1.1 abgedruckt; sie verwendet das Property *IndexDefs* der *TTable*-Komponente. Dieses ist auch schon zur Entwurfszeit (im Objektinspektor oder im Objekthierarchie-Fenster) auswählbar und kann dann analog zum Property *FieldDefs* editiert werden. Wie schon beim Property *FieldDefs* werden auch die Indexdefinitionen erst beim Neu-Erzeugen der Tabelle in die Realität umgesetzt. Rufen Sie dazu TABELLE ERSTELLEN aus dem lokalen Menü der *TTable*-Komponente auf, wie schon unter 7.3.3 beschrieben.

Suchen mit einem Index

Das Vorhandensein eines passenden Indexes ist eine wichtige Voraussetzung für das Suchen und Filtern von Daten in großen Tabellen. Delphi erlaubt zwar schon seit der Version 2 auch Such- und Filteroperationen ohne Index, diese sind jedoch nur bei kleinen Tabellen empfehlenswert. Wichtig ist vor allem, dass die Suchmethoden automatisch einen passenden Index auswählen, wenn dieser erst einmal existiert.

Wenn Sie beispielsweise in der Dateidatenbank aus Kapitel 7.4.4 einen bestimmten Dateinamen suchen wollen, so sollte das Feld *FileName* indiziert sein. Der Primärindex von `files.db` besteht zwar aus den Feldern *DirID* und *FileName*, eignet sich aber dennoch nicht zur Suche *nur* nach einem Dateinamen, denn bei zusammengesetzten Indizes wird zuerst im ersten Feld nach Übereinstimmung gesucht. Sie können also mit dem voreingestellten Index dieser Tabelle entweder nur nach der Verzeichnisnummer oder nach Verzeichnisnummer *und* Dateiname gleichzeitig suchen. Um den Dateinamen alleine per Index zu suchen, benötigen Sie einen Index, der das Feld *FileName* an erster Stelle enthält.

Den Index auswählen

Da die in Kapitel 7.4.1 beschriebenen Methoden selbst einen passenden Index auswählen, müssen Sie sich oft nicht selbst um die Auswahl eines Indizes bemühen. Wenn die Daten allerdings sortiert angezeigt werden sollen, bieten sich die im Folgenden beschriebenen Vorgehensweisen an, um einen Index auszuwählen.

Zunächst ist interessant, wie überhaupt die voreingestellte Sortierreihenfolge einer Datenmenge aussieht: Paradox-Tabellen werden standardmäßig nach ihrem Primärindex sortiert. Die standardmäßige Sortierreihenfolge in einer *TQuery*-Komponente oder in einem SQL-ClientDataSet von dbExpress wird durch die Reihenfolge bestimmt, in der die Datensätze von dem Server geliefert werden. Auf dem Server kann die Reihenfolge beispielsweise durch einen voreingestellten Primärindex bestimmt werden, falls nicht explizit ein anderer Index gewählt wird, oder durch eine *Order-By*-Klausel in einer SQL-Abfrage. In einer von einem Server unabhängigen Cli-

ent-Datenmenge ist es auch möglich, die Daten ohne jegliche Sortierung einzugeben und anzeigen zu lassen.

Um einen anderen Index zu aktivieren, können Sie die Properties *IndexName* und *IndexFieldNames* verwenden, sie stehen sowohl für die BDE in *TTable* als auch für dbExpress in *TSQLClientDataSet* und ganz allgemein in allen *TClientDataSets* zur Verfügung:

- In *IndexName* nennen Sie den Index bei dem Namen, den Sie ihm beispielsweise in der Datenbankoberfläche gegeben haben, oder Sie geben den Namen *DEFAULT_INDEX* an, der wieder auf die voreingestellte Sortierreihenfolge zurückschaltet (evtl. auch ohne Sortierung). Wenn ein Index aus mehreren Feldern zusammengesetzt ist, wirkt dieser Name wie ein Kürzel für diese Felder.
- Alternativ dazu können Sie in *IndexFieldNames* die einzelnen Felder aufzählen, die im Index vorhanden sein sollen (wobei Sie die Felder mit Semikolons trennen). Die VCL versucht dann, einen entsprechenden Index zu finden. Bei einer *TClientDataSet*-Komponente können Sie hier beliebige Felder der Datenmenge angeben, auch wenn dafür noch kein Index existiert, denn *TClientDataSet* legt bei Bedarf einfach einen neuen Index im Hauptspeicher an.

Wenn Sie eines der beiden Properties setzen, wird jeweils das andere gelöscht. Um auch dann die Felder des aktuellen Indexes zu erfahren, wenn Sie *IndexName* gesetzt haben, können Sie die *DataSet*-Properties *IndexFields* und *IndexFieldCount* verwenden. Eine Übersicht über alle definierten Indizes gibt das Property *IndexDefs*. In der Handhabung sind diese drei Properties den Properties *Fields*, *FieldCount* und *FieldDefs* sehr ähnlich, falls dazu noch Fragen offen sind, sollten diese anhand der Online-Hilfe geklärt werden.

Hinweis: Wenn Sie für eine Datenbanktabelle verschiedene Indizes haben, bietet es sich eventuell an, mehrere *TTable*-Objekte für diese Tabelle zu erzeugen und in jedem dieser Objekte einen anderen Index zu wählen. Sie können dann gleichzeitig verschiedene Sortierungen der ursprünglichen Datenbanktabelle verwenden, ohne zwischen den Indizes umherschalten zu müssen.

7.4.3 Modusabhängige Filter- und Suchfunktionen

Die in diesem Kapitel vorgestellten Methoden sind – im Gegensatz zu den in Kapitel 7.4.1 besprochenen – schon seit der ersten Delphi-Version in *TTable* vorhanden. Sie werden aber auch von Client-Datenmengen implementiert, stehen also auch unter dbExpress zur Verfügung. Vor allem die nachfolgend besprochenen Methoden zum Filtern zwar weniger flexibel, aber einfacher zu handhaben als die in Kapitel 7.4.1 erläuterte Filterungsmöglichkeit.

Nach Bereichen filtern

Die Methoden zum Filtern von Bereichen geben Ihnen eine einfache Möglichkeit, die Datensätze einer Datenmenge abzufragen, die einfache Bereichskriterien erfüllen. Für jede Tabellenspalte können Sie einen eigenen Bereich definieren, in dem die Feldwerte der Datensätze liegen müssen. Wenn Sie für mehrere Spalten Bereiche definieren, müssen alle Einzel-Bereichskriterien erfüllt sein, damit ein Datensatz ins Filterergebnis aufgenommen wird.

Bei der Verwendung der Methode *SetRange* müssen Sie (im Gegensatz etwa zu den in Kapitel 7.4.1 genannten Methoden) selbst einen Index setzen, denn nach dessen Spalten entscheidet sich erst, wo überhaupt gefiltert werden soll. Mit *SetRange* können Sie alle Bereiche auf einmal setzen. Die Methode erwartet sowohl für den Bereichsbeginn als auch für das Ende einen Array-Parameter, innerhalb dessen Sie die einzelnen Werte in der Reihenfolge des Felder des aktuellen Indexes angeben. Wenn beispielsweise zwei numerische Felder indiziert sind, beschränken Sie die Anzeige der Datensätze mit der folgenden Anweisung auf die Datensätze, die in beiden Feldern einen Wert zwischen 0 und 2048 haben:

```
// Der Index muss vorher eingestellt werden, z.B.:
// Table.IndexFieldNames := 'NumField1;NumField2';
Table.SetRange([0, 0], [2048, 2048]);
```

Dasselbe Ergebnis wie mit diesem einzelnen Aufruf können Sie wie folgt auch in kleineren Schritten erreichen. Sie müssen sich dann nicht um die Reihenfolge der Werte im Array kümmern.

Als Erstes rufen Sie die Methode *SetRangeStart* auf, die die Tabelle in einen speziellen Modus setzt, in dem das Beschreiben der Feldwerte nicht den aktuellen Datensatz verändert, sondern den Beginn des Bereichs definiert. Sie können die Felder dann mit den Werten beschreiben, die die jeweilige Spalte mindestens haben muss, um in den Ergebnisbereich aufgenommen zu werden:

```
Files.SetRangeStart;
FilesGroesse.AsInteger := 0; { Bereichsbeginn im Feld FilesGroesse: 0 }
```

(Dieses Beispiel nimmt an, dass Sie mit dem Felder-Editor eine persistente *TField*-Komponente namens *FilesGroesse* für eine numerische Spalte der Tabelle angelegt haben.)

Danach rufen Sie *SetRangeEnd* auf und beschreiben die Feldwerte mit dem Ende des Bereichs. Zum Abschluss aktivieren Sie den Bereich mit einem Aufruf von *ApplyRange*:

```
Files.SetRangeEnd;
FilesGroesse.AsInteger := 2048;{ Bereichsende im Feld FilesGroesse: 2048 }
ApplyRange;
```

Um die Beschränkung irgendwann wieder aufzuheben, wenden Sie sich an die Methode *CancelRange*. Sie können außerdem mit dem Property *KeyExclusive* steuern, ob die Bereichsgrenzen zum Filterergebnis dazugehören, was per Voreinstellung der Fall ist (*KeyExclusive=False*). Außerdem können Sie das Bereichsende auch offen lassen, beispielsweise wenn Sie alle Dateien finden wollen, deren Name mit einer bestimmten Zeichenkette beginnt:

```
Files.SetRangeStart;
FilesDateiname.AsString := 'VERY';
Files.SetRangeEnd; { Alle Strings finden, die mit VERY beginnen }
Files.ApplyRange;
```

Suchen

Da Kapitel 7.4.1 bereits die leistungsfähige *Locate*-Funktion besprochen hat, sollen die alternativen Suchfunktionen hier nur am Rande erwähnt werden. Sie unterscheiden sich von *Locate* dadurch, dass die Angabe dessen, was gesucht werden soll, nicht in einem, sondern in mehreren Schritten erfolgt. Je nach Situation kann eine solche ausführliche Schreibweise übersichtlicher, umständlicher oder einfach eine Geschmacksfrage sein.

Zuerst geben Sie die Spalten an, in denen gesucht wird, indem Sie eine passende Indizierung in *IndexFieldNames* oder *IndexName* wählen, dann folgt eine Sequenz, die dem Filtern mit *SetRangeStart/End* und *ApplyRange* aus dem vorherigen Abschnitt stark ähnelt: Statt der Bereichsgrenzen schreiben Sie die zu suchenden Daten in die Felder, zur Einleitung rufen Sie die Methode *SetKey*, zum Abschluss *GotoKey* auf. Wenn Sie nach Teilschlüsseln suchen wollen, verwenden Sie *GotoNearest* statt *GotoKey*.

Hinweis: Die Verwendung von *SetKey/GotoKey* kann sogar effizienter sein als der Aufruf von *Locate*, nämlich wenn mehrere geringfügig unterschiedliche Suchen aufeinander folgen. Während Sie bei *Locate* immer die gesamte Parameterliste angeben müssen – von den Suchspalten bis zu den Werten für jede einzelne Spalte – brauchen Sie vor einem erneuten Aufruf von *GotoKey* nur die Spalten zu ändern, deren Suchwerte sich verändert haben.

In einem Index, der mehr Felder beinhaltet, als Sie für die derzeitige Suche bzw. Filterung benötigen, können Sie die Suche in den nicht benötigten Spalten abschalten, indem Sie das Property *KeyFieldCount* setzen. Wollen Sie beispielsweise mit einem Index »Nachname;Vorname« nur nach dem Nachnamen suchen, etwa um den Vornamen herauszufinden, müssen Sie *KeyFieldCount* auf eins setzen. Wenn Sie keinen Vornamen für die Suche angeben, *KeyFieldCount* aber größer als eins ist, wird die Datenbank-Engine nach einem Nachnamen mit leeren Vornamen suchen, den sie wahrscheinlich nicht findet.

TDataSet-Zustände

Die Zustände einer *TDataSet*-Komponente dienen unter anderem dazu, bei den Zugriffen auf die aktuellen Feldwerte zu unterscheiden, ob dieser Zugriff die Daten des Datensatzes verändert oder ob er zur Angabe einer Bereichsgrenze oder von Suchparametern dient. Nachdem wir die Tabelle in den letzten Abschnitten in den *dsSetKey*-Modus geschaltet haben, fasst die folgende Tabelle die wichtigsten Zustände einer *DataSet*-Komponente zusammen.

Zustand	**Beschreibung**	**herbeigeführt durch**	**beendet durch**
dsInactive	Die Tabelle ist noch geschlossen.	Close	Open
dsBrowse	Anfangsmodus nach der Aktivierung, kein Editieren möglich.	Beenden jedes anderen Zustandes	Wechsel in einen beliebigen anderen Zustand
dsEdit	Normaler Editiermodus: Eine vorhandene Zeile wird editiert. Änderungen der Daten der Feldkomponenten werden bei *Post*-Aufruf in die Datenbank übernommen.	Edit	Cancel, Delete, Post
dsInsert	Eine neue, noch nicht gespeicherte Zeile wird editiert, sonst wie *dsEdit*.	Insert, Append	Post, Delete
dsSetKey	Suchparameter werden spezifiziert, nur bei *TTable* gültig.	SetKey, EditKey, SetRangeStart, SetRangeEnd	Cancel, GotoKey, FindKey, ApplyRange
dsCalcFields	Nur berechnete Felder können verändert werden (während der Bearbeitung des *OnCalcField*-Events).	automatisches Ereignis	Ende der *OnCalcField*-Methode
dsFilter	Eine mit dem *OnFilterRecord*-Ereignis verknüpfte Methode prüft gerade das Filter kriterium.	automatisches Ereignis, falls *Filtered=True*	Ende der *OnFilter Record*-Methode

Sie können den aktuellen Zustand jederzeit über das Property *TDataSet.State* abfragen, allerdings nicht über dieses Property ändern. Hierzu müssen Sie die in der Tabelle gezeigten Methoden aufrufen.

7.4.4 Aktualisieren der Dateidatenbank

In diesem Kapitel geht es darum, die Beschränkungen der Dateidatenbank aus Kapitel 7.3.6 zu beseitigen: Da diese die Datensätze immer nur hinzufügt, aber niemals bestehende ersetzt, kann es schnell zu doppelten Einträgen, folglich zu doppelten Schlüs-

seln und somit zu einer Indexfehler-Exception kommen, sobald die Anwendung versucht, die geschriebenen Datensätze mit *Post* in das Änderungsprotokoll zu übernehmen.

Die Indizes der beiden Tabellen

Beim Schreiben eines neuen Datei-Datensatzes genügt es, nach den Feldern des Primärindexes zu suchen, um eine eventuell schon bestehende Datei zu finden: Jede Datei ist durch ihre Verzeichnisnummer und den Dateinamen eindeutig bestimmt (sofern das Betriebssystem keine doppelten Dateinamen erlaubt).

Anders verhält es sich bei der Frage, ob ein Verzeichnis schon vorhanden ist. Die Verzeichnistabelle ist im Primärindex nach der Verzeichnisnummer sortiert, muss aber in diesem Fall nach dem Pfadnamen durchsucht werden. Hierfür erhält die Tabelle einen neuen Index, der sie nach dem Feld *Path* sortiert.

Die folgende Tabelle zeigt alle in *FileDB* verwendeten Indizes, von denen einige erst im nächsten Kapitel eingesetzt werden:

Tabelle	Indexname	Indizierte Felder
Dirs	– (Primärindex)	DirID
Files	– (Primärindex)	DirID;FileName
Dirs	Path	Path
Files	ByDate	DirID;FileTime
Files	BySize	DirID;FileSize

Hinweis: Die Primärindizes von Paradox haben grundsätzlich keine Namen, in der Interbase-Version der Datenbank (siehe SQL-Skript in Kapitel 7.1.3) wurden die beiden Primärindizes mit den Namen *Directories0* und *Files0* versehen (dies war der bei der Konvertierung der Datenbank automatisch vom Data-Pump-Tool vorgeschlagene Name).

Suchen, um zu aktualisieren

Die neue Version von *StoreFile* sucht nun, bevor sie die in den Parametern erhaltenen Dateidaten in einen Datensatz schreibt, in den Spalten *DirID* und *FileName*, ob bereits eine Datei des gleichen Namens und mit gleicher Verzeichnisnummer in der Datenbank enthalten ist. Das Ergebnis der Suchfunktion *Locate* entscheidet dann darüber, ob mit *Insert* ein neuer Eintrag eingefügt wird oder der gefundene Eintrag mit *Edit* ange-

passt wird (schließlich könnten sich die Größe und das Datum einer schon früher eingelesenen Datei geändert haben):

```
procedure TMainForm.StoreFile(Dir: LongInt; Name: string;
                              Size: LongInt; Date: TDateTime);
begin
  Application.ProcessMessages; { "Multitasking", siehe Kapitel 7.3.7 }
  if not Files.Locate('DirID;FileName',
                      VarArrayOf([Dir, Name]), [])
  then
    Files.Insert
  else begin
    inc(FileUpdates); { für die Statusanzeige }
    Files.Edit;
  end;
  { Das Eintragen der Daten ist für Insert und Edit gleich: }
  FilesFileTime.AsDateTime := Date;
  FilesFileName.AsString := Name;
  FilesDirID.AsInteger := Dir;
  FilesFileSize.AsInteger := Size;
  Files.Post; { Änderungen schreiben }
end;
```

Eindeutige Verzeichnisnummerierung

FileDB verwendet eine sehr einfache Methode, jedes Verzeichnis mit einer eindeutigen Nummer zu versehen: Jedes neue Verzeichnis erhält eine Nummer, die um eins größer ist als die bisher maximale Nummer. Falls einmal Verzeichnisse gelöscht werden (eine in diesem Beispielprogramm nicht implementierte Funktion), füllt *FileDB* die Lücken in der Nummerierung nicht auf.

Nach dem Öffnen der Tabelle stellt *FormCreate* die bisher größte Verzeichnisnummer fest, indem sie mit der Methode *TTable.Last* an das Ende der Tabelle springt (die nach ihrem Primärindex nach der Verzeichnisnummer sortiert ist) und die Nummer des dort befindlichen Eintrags ausliest:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Files.Active := True;
  Dirs.Active := True;
  Dirs.Last;
  MaxDirIndex := DirsDirID.AsInteger;
  UpdateLabels;
end;
```

Die Methode *StoreDir* ist geringfügig anders aufgebaut, da sie keine Daten aktualisieren muss, wenn das Verzeichnis schon vorhanden ist:

```
function TMainForm.StoreDir(Path: string): LongInt;
begin
```

```
    if not Dirs.Locate('Path', Path, []) then begin
      Dirs.Insert;
      inc(MaxDirIndex);
      Result := MaxDirIndex;
      DirsDirID.AsInteger := MaxDirIndex;
      DirsPath.AsString := Path;
      Dirs.Post;
    end else begin
      inc(DirUpdates);
      Result := DirsDirID.AsInteger;
    end;
  end;
```

Neben dem Speichern der Datensätze wird hier die zweite Aufgabe von *StoreDir* besonders wichtig: die Rückgabe der Verzeichnisnummer an die Methode *InsertDir*, die diese Nummer an *StoreFile* weitergibt (zu *InsertDir* siehe Kapitel 7.3.6).

7.4.5 Haupt-/Detailformulare und Sortieren

Das Formular des Beispielprogramms *FileDB* (Abbildung 7.15) ist ein typisches Beispiel für ein Haupt-/Detail- (bzw. Master-/Detail-)Formular: Eine Haupttabelle gibt eine grobe Übersicht über alle zur Verfügung stehenden Daten – in diesem Fall über alle eingelesenen Verzeichnisse, während die Detailtabelle genauere Angaben zu dem Element gibt, das gerade in der Haupttabelle ausgewählt ist – in diesem Fall also alle in diesem Verzeichnis gefundenen Dateien.

Properties zur Verknüpfung *R180*

Sie können eine Haupt-/Detailkombination mit dem Datenbankformular-Experten automatisch erstellen oder von Hand, indem Sie zwei Tabellen, zwei *DataSource*-Komponenten und zwei *DBGrids* wie gewohnt in das Formular aufnehmen und dann zwei Properties der Detailtabelle verändern:

- In *MasterSource* wählen Sie die *TDataSource*-Komponente der Haupttabelle, nach der sich die Detailtabelle richten soll.
- In *MasterFields* geben Sie die Felder der Haupttabelle an, die die Auswahl der Details der Detailtabelle steuern sollen. Der Inhalt dieser Felder wird an den Index der Detailtabelle geleitet, und diese zeigt dann nur die Datensätze, die in den Indexfeldern mit den *MasterFields* der Haupttabelle übereinstimmen.

Für *FileDB* bedeutet das: Die *MasterSource* von *Files* ist *DirSource*, die mit der Verzeichnistabelle verbundene *TDataSource*-Komponente. Die *MasterFields* enthalten nur ein Feld, und zwar das Feld *DirID*.

Die Verknüpfung der beiden Tabellen wirkt sich wie folgt aus: Jedes Mal, wenn Sie in der Verzeichnisliste einen anderen Datensatz auswählen, wird die Nummer dieses Verzeichnisses im Index der Dateitabelle gesucht und nur die Dateien mit derselben Nummer werden aufgelistet.

Hinweis: Für das Einfügen neuer Datensätze beim Einlesen eines Verzeichnisses muss die Master-/Detail-Verknüpfung von FileDB wieder gelöst werden, damit per *Locate* alle Dateien der Dateitabelle gefunden werden können (sonst würden nur die Dateien des Verzeichnisses gesucht, das gerade in der Master-Tabelle gewählt ist). Zum Lösen der Verknüpfung muss *MasterSource* auf *nil* geschaltet werden. Dies wurde bereits in Kapitel 7.3.7 in der Methode *AddFromDir* gezeigt.

Haupt-/Detailverbindungen im Datendiagramm

Abbildung 7.18 zeigt die Verbindung der beiden Tabellen in der Diagramm-Ansicht. Die Linie zwischen den Tabellen wird mit dem Inhalt des *MasterFields*-Properties der Detail-Tabelle beschriftet. Wie in Kapitel 7.2.8 erwähnt, zeichnet Delphi die Verbindungslinie automatisch, sobald Sie die beiden Tabellen von der Objekthierarchie in das Diagramm ziehen.

Mit dem entsprechenden Symbol der Schalterleiste können Sie auch manuell solche Haupt-/Detailverbindungen zwischen zwei Tabellen ziehen. Sie erhalten dann einen Dialog, in dem Sie die zu verbindenden Felder auswählen können. Die Properties der Tabellenkomponenten werden daraufhin automatisch eingestellt.

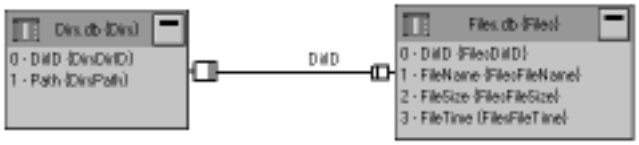


Abbildung 7.18: Die symbolische Darstellung der Haupt-/Detailverbindung des Beispielprogramms in der Diagramm-Ansicht

Sortieren der Details *R184*

Zum Abschluss erhält das *DBGrid* der Dateiliste noch ein kleines Popup-Menü, in dem Sie wählen können, ob die Dateien nach Größe, Datum oder nach dem Namen sortiert werden sollen. Die Sortierung kann prinzipiell einfach dadurch erreicht werden, dass die Spalten, nach denen sortiert werden soll, in das Property *IndexFieldNames* der Datenmenge geschrieben werden. In diesem Fall muss aber zusätzlich bedacht werden, dass *Files* eine Detail-Tabelle in einem Master-/Detailgespann ist und über die Verzeichnisnummer mit der Verzeichnistabelle verknüpft ist. Und die *MasterFields*

einer Detail-Datenmenge müssen immer die ersten Bestandteile des aktuellen Indexes sein, damit diese Verknüpfung funktioniert.

Um sowohl der Steuerung durch die Master-Tabelle als auch dem Sortierungswunsch des Benutzers gerecht werden zu können, benötigt *FileDB* also die beiden Indizes *DirID;FileSize* und *DirID;FileTime* (eine Tabelle in Kapitel 7.4.4 zeigt alle von *FileDB* verwendeten Indizes). Mit diesen Indizes können die drei lokalen Menüpunkte schnell implementiert werden:

```
procedure TMainForm.NameSortClick(Sender: TObject);
begin
  Files.IndexFieldNames := 'DirID;FileName';
end;

procedure TMainForm.SizeSortClick(Sender: TObject);
begin
  Files.IndexFieldNames := 'DirID;FileSize';
end;

procedure TMainForm.DateSortClick(Sender: TObject);
begin
  Files.IndexFieldNames := 'DirID;FileTime';
end;
```

7.5 Eine Beispielanwendung

Nachdem die Beispiele der vorangegangenen Kapitel ihre speziellen Demonstrationsaufgaben fernab der Alltagspraxis ausgeführt haben, zeigt dieses Kapitel zum Abschluss ein praxisnähere Anwendung: Eine Terminverwaltung, die auf einer Datenbank mit drei Tabellen basiert, in denen Termine sowie damit verknüpfte Adressen und Aktionen abgelegt werden, soll vom Benutzer auf verschiedene Weise bearbeitet und angesehen werden können (siehe Kapitel 7.5.3, Abbildungen 7.21 – 7.23).

Die eigentliche Funktion der Anwendung soll dabei offen bleiben – so enthält die Aktionstabelle etwa ein Feld, in dem sich ähnlich wie beim Beispielprogramm aus Kapitel 1.9.5 Programme eintragen lassen, die aber von dieser Anwendung nicht gestartet werden. Die Termintabelle enthält Felder, die eine Wiederholung des Termins im Tages-, Monats- oder Jahresrhythmus zulassen, doch die Anwendung erinnert nicht an diese Wiederholung. Statt dessen sollen in diesem Kapitel einige Funktionen implementiert werden, die uns Entwicklern Experimente mit der Funktionsweise von dbExpress erlauben: einen SQL-Monitor, eine Simulation von Mehrbenutzerzugriff auf die Datenbank und ein Formular zur Ansicht des Änderungsprotokolls (in einer modifizierten Form könnte ein solches Formular auch für normale Benutzer interessant sein).

Bei diesem Beispiel handelt es sich um eine echte Cross-Platform-Anwendung, es wird also nicht nur die Kylix-kompatible dbExpress-Schnittstelle verwendet, sondern statt der VCL kommt die auch unter Kylix verfügbare CLX zum Einsatz. Da die CLX nicht Thema dieses Buches ist, soll dieser Umstand im Folgenden nicht näher beachtet werden (die Anwendung funktioniert ebenso auch mit der VCL, so dass Sie sie im Folgenden auch einfach als VCL-Anwendung ansehen können).

Alternativer Zugriff über die BDE

Da ältere Delphi-Versionen das Projekt wegen fehlender dbExpress-Architektur nicht kompilieren können, finden Sie auf der CD unter der Projektbezeichnung *BDETermin-Viewer* auch noch eine Mini-Version der Anwendung für die BDE. Diese Mini-Version beschränkt sich auf den kompliziertesten Teil der Anwendung – die Abfrage der für einen bestimmten Tag eingetragenen Termine, wobei Informationen aus drei Tabellen in einer einzigen Datenmenge zusammengeführt werden (in den folgenden Kapiteln ist dies das ClientDataSet namens *cdsTagesTermine*), und das Update der Tabellen, wenn das Abfrageergebnis im DBGrid editiert wird (siehe Kapitel 7.5.6). Für die Abfrage der aktuellen Termine verwendet *BDETerminViewer* statt der *TSQLClientDataSet*- eine *TQuery*-Komponente. Deren *SQL*-Property enthält eine ähnliche SQL-Anweisung wie das Property *CommandText* des SQLClientDataSets. Hier im Buch wird lediglich die dbExpress-Version der Anwendung besprochen.

7.5.1 Definition der Datenbank

Da die Anwendung unabhängig von der BDE sein soll, kommt die Verwendung einer dBase- oder Paradox-Datenbank nicht in Frage. Auf der CD finden Sie sowohl eine fertige Datenbankdatei für Interbase als auch das SQL-Skript zur Erzeugung der Datei. Mit diesem Skript sollte es möglich sein, die Datenbank auch mit anderen SQL-Servern als Interbase zu verwenden. Das Beispielprogramm ist auf der CD unter dem Namen *TerminVerwaltung* zu finden.

Metadaten-Extraktion

Das in diesem Kapitel gezeigte Skript wurde nicht per Hand angefertigt, da die ursprüngliche Interbase-Datei von dem in Kapitel 7.1.5 beschriebenen Data Pump-Utility aus drei Paradox-Tabellen erzeugt wurde, sondern mit dem *isql*-Utility von Interbase. Auf eine bestehende Interbase-Datei angewendet, kann *isql* die Metadaten dieser Datei extrahieren und in Form von DDL-Anweisungen ausgeben[18]. Aus diesen Anwei-

18 In der Windows-Ausgabe von Interbase ist dies auch mit dem GUI-Tool *Interbase Windows ISQL* (kurz *wisql*) möglich, und zwar bei bestehender Datenbankverbindung über den Menüpunkt METADATA | EXTRACT DATABASE.

sungen lässt sich die Struktur der Datenbank genau reproduzieren, so dass die gesamte Ausgabe von *isql* sich wieder als Skript zum Erzeugen einer neuen Datenbank verwenden lässt. Der Aufruf der Extraktionsfunktion von *isql* geschieht wie folgt:

```
isql -extract TerminDB.gdb
```

Dadurch werden die DDL-Anweisungen auf der Konsole ausgegeben. Eine Umleitung in die Datei ist leicht mit der entsprechenden Shell-Syntax oder mit der *isql*-Option »-output Dateiname« möglich. Zu beachten ist dabei, dass diese Option *zwischen* die *extract*-Option und den Namen der Datenbank gehört:

```
isql -extract -output MyScript.sql TerminDB.gdb
```

Hinweis: Standardmäßig gibt *isql* alle DDL-Anweisungen und Namen von Tabellen, Feldern, Generatoren etc. in Großbuchstaben aus. SQL-Anweisungen und -Bezeichner sind wie Object-Pascal-Code jedoch nicht abhängig von Groß- und Kleinschreibung. Daher werden die Listings von *isql* hier zwar originalgetreu wiedergegeben, im Folgenden aber für die selbst definierten Bezeichner auch Kleinbuchstaben verwendet.

Die Tabellen der Termindatenbank

Die Erzeugung einer Datenbank mit *isql* wurde bereits in Kapitel 7.1.3 erläutert, so dass im Folgenden nur noch die DDL-Anweisungen gezeigt werden, mit denen die im Beispielprogramm verwendeten Tabellen erstellt werden können:

```
CREATE DATABASE "TERMINE.GDB" PAGE_SIZE 1024;

CREATE TABLE TERMINE (TERMINNR INTEGER,
        DATUM DATE,
        ZEIT DATE,
        BESCHREIBUNG VARCHAR(80),
        AKTIONNR INTEGER,
        ADRESSNR INTEGER,
        WDHOLENT INTEGER,
        WDHOLENM INTEGER,
        WDHOLENJ INTEGER);
```

Diese Tabelle versieht jeden Termin mit einer Nummer. Die »Kerndaten« eines Termins bestehen aus der *Beschreibung* und einer Zeitangabe, die in *Datum* und *Zeit* aufgeteilt wurde.

Die drei letzten Felder *WdHolenT/M/J* sind für sich wiederholende Termine gedacht und werden in der Programmversion auf der CD nur dadurch beachtet, dass alle wiederholten Termine in einer Tabelle des Hauptfensters aufgelistet werden (jedoch rech-

Hinweis: Effizienter wäre es, Datum und Zeit in einem gemeinsamen Feld zu speichern, denn der *Date*-Datentyp von Interbase enthält auch Zeitinformationen. Vor der Konvertierung hatten die beiden Felder in der Paradox-Tabelle noch spezielle Typen für Datum und Zeit. Für dieses Beispielprogramm wurde es bei den beiden Feldern belassen, weil dadurch eine einfache Möglichkeit für den Benutzer entsteht, entweder Datum oder Zeit offen zu lassen (das Feld bei der SQL-Konstanten *null* zu belassen).

net das Programm nicht aus, an welchen Tagen nun die Wiederholungen eines bestimmten Termins stattfinden).

Wichtig sind die beiden Felder *AktionNr* und *AdressNr*, die den Termin jeweils mit einer Aktion und einer Adresse aus den folgenden anderen beiden Tabellen verknüpfen:

```
CREATE TABLE ADRESSEN (ADRESSNR INTEGER,
        VORNAME VARCHAR(40),
        NACHNAME VARCHAR(40),
        POSTANSCHRIFT VARCHAR(100),
        EMAIL VARCHAR(100),
        NOTIZEN VARCHAR(100));

CREATE TABLE AKTIONEN (AKTIONNR INTEGER,
        AUSGABEOPTIONEN INTEGER,
        SIGNALZAHL INTEGER,
        KOMMANDO VARCHAR(180));
```

Für das Beispielprogramm sind die wichtigsten Felder dieser beiden Tabellen jeweils die erstgenannten, die die Adress- und Aktionsnummer angeben. Alle anderen Felder gehören lediglich zur »Kulisse« des Programms.

Hinweis: Was es bedeutet, dass ein Termin mit einer Adresse bzw. einer Aktion verknüpft ist, bleibt in diesem Beispielprogramm alleine dem Benutzer überlassen. Wie schon in der Einleitung erwähnt, führt das Programm selbst keine Interpretationen aus und beinhaltet auch keine automatische Ausführung irgendwelcher Aktionen. Die Bedeutung der Felder *Ausgabeoptionen*, *Signalzahl* und *Kommando* ist also nicht näher definiert.

Automatische Nummerierung von Datensätzen *R182*

Jede der drei Beispieltabellen verfügt über eine Spalte, durch die jeder Datensatz eine Nummer zugeordnet bekommt. Dass diese Nummer eindeutig sein sollte, versteht sich von selbst, da die Nummer in anderen Tabellen als Referenz auf den jeweiligen Datensatz verwendet werden soll. Eine Möglichkeit, diese Eindeutigkeit von der Datenbanksoftware sicherstellen zu lassen, besteht darin, auf jede Nummernspalte

einen eindeutigen Index zu definieren, also einen Index, in dem jeder Wert nur ein einziges Mal vorkommen darf. Einen solchen erhält man in SQL mit dem Schlüsselwort *UNIQUE*:

```
/*  Index definitions for all user tables */
CREATE UNIQUE INDEX ADRESSENO ON ADRESSEN(ADRESSNR);
CREATE UNIQUE INDEX AKTIONENO ON AKTIONEN(AKTIONNR);
CREATE UNIQUE INDEX TERMINEO ON TERMINE(TERMINNR);
```

Jeder Versuch, einen neuen Datensatz mit einer schon bestehenden Nummer zu versehen, wird von Interbase mit einem SQL-Fehler geahndet, der seine Laufbahn in einer Delphi-Anwendung als Exception fortsetzt. Die drei Indizes dienen in diesem Beispielprogramm hauptsächlich dieser Absicherung der Eindeutigkeit und werden ansonsten nicht explizit verwendet, da das Programm auf Sortieroptionen verzichtet.

Interbase verfügt über ein sehr robustes Konzept für die automatische Zuweisung eindeutiger Schlüssel zu neuen Datensätzen, mit dessen Hilfe es der Anwendung leicht gemacht wird, die Eindeutigkeitsregel für die Nummern zu beachten. Dieses Konzept setzt sich zusammen aus

- Generatoren – das sind Variablen, deren Werte von Interbase automatisch bei jedem Abruf in festgelegten Schritten erhöht werden,
- Triggern – das sind Anweisungen, die als Reaktion auf bestimmte Ereignisse auf dem Server selbst ausgeführt werden.

Das Ereignis, das bei der automatischen Nummerierung entscheidend ist, ist das der Erzeugung eines neuen Datensatzes; in SQL heißt dieses Ereignis *INSERT* und Sie können eigenen Code sowohl direkt vor diesem Ereignis als auch danach ausführen lassen (hierfür geben Sie vor *INSERT* noch den Zusatz *BEFORE* oder *AFTER* an).

Um beispielsweise das Feld *TerminNr* in der Termintabelle bei jeder Einfügung eines neuen Datensatzes mit einem neuen Wert des Generators zu belegen, wird folgender Trigger definiert:

```
SET TERM ^ ;
CREATE TRIGGER CREATE_TERMINNR FOR TERMINE
ACTIVE BEFORE INSERT POSITION 0
AS BEGIN
    NEW.TERMINNR = GEN_ID(TerminNr_Gen, 1);
END ^
SET TERM ; ^
```

Sie können auf diese Weise auch *mehrere* Ereignis-Handler mit demselben Ereignis verknüpfen. Die Angabe von *POSITION 0* besagt, an welcher Stelle dieser Handler in einer Kette von mehreren Handlern ausgeführt werden soll.

Wer die Arbeit mit einem »intelligenten« Compiler wie dem Object-Pascal-Compiler von Delphi gewohnt ist, wird sich bei der ersten Begegnung mit einem solchen Skript über eine weitere Besonderheit wundern: Die gesamte Trigger-Definition soll von SQL als eine Anweisung verstanden werden und muss damit mit einem »Terminator« -Zeichen beendet werden, welches standardmäßig das Semikolon ist. Dieses Semikolon wird jedoch auch zum Abschluss des geschachtelten Codes, der beim *INSERT*-Ereignis ausgeführt werden soll, benötigt. Damit der ahnungslose SQL-Interpreter dieses geschachtelte Semikolon nicht als Ende der *CREATE-TRIGGER*-Anweisung missversteht, muss vor dieser SQL-Anweisung mit *SET TERM* ein anderes Terminatorzeichen eingestellt werden. Diese Einstellung wird am Schluss wieder rückgängig gemacht.

Der im obigen Skript genannte Generator muss schon vorher definiert worden sein, in der Beispieldatenbank geschah dies wie folgt:

```
CREATE GENERATOR TERMINNR_GEN;
```

Ausführliche Informationen zu *CREATE TRIGGER* und *CREATE GENERATOR* erhalten Sie beispielsweise in der mit Interbase mitgelieferten Dokumentation *langref.pdf*.

Verwendung des Triggers in der Praxis

Die Verwendung des Triggers sieht nun so aus, dass bei der Eingabe eines neuen Datensatzes die Nummernspalte einfach offen bleiben kann. Wenn der Datensatz aus dem Änderungsprotokoll mit *ApplyUpdates* an den Server geschickt wird, versieht dieser den Datensatz mit einer neuen Nummer. Damit der Benutzer nicht versehentlich trotzdem eine Nummer eingibt, die dann vom Server überschrieben wird, sollte das Nummernfeld sich in der Anwendung nicht editieren lassen. Damit die vom Server erzeugte Nummer später auch im Programm angezeigt wird, ist ein *Refresh* der Datenmenge notwendig (siehe *TdmTermine.cdsGeneralAfterPost* Kapitel 7.5.4).

7.5.2 Das Datenmodul

Im Gegensatz zu den bisherigen Beispielprogrammen drängt es sich für die Terminverwaltung geradezu auf, alle Datenzugriffskomponenten in einem eigenen Datenmodul unterzubringen, denn die Komponenten sollen von vielen verschiedenen Formularen aus angesprochen werden.

Abbildung 7.19 zeigt das Datenmodul, wie es von der Delphi-IDE präsentiert wird.

- Zunächst wird eine Verbindungskomponente (*TSQLConnection*) verwendet, über die die Verbindung zu Interbase hergestellt wird (siehe Kapitel 7.1.6). Das Property *LoginPrompt* wurde auf *False* eingestellt, weil die Datenbank sowieso nur das allgemein bekannte Passwort *masterkey* verwendet. Alle im Folgenden genannten Datenmengen sind über ihr *DBConnection*-Property mit dieser Verbindungskomponente verknüpft.

- Für jede der drei Tabellen wurde nun jeweils ein Exemplar von *TSQLClientDataSet* und eines von *TDataSource* in das Modul eingefügt (*cdsTermine*, *cdsAdressen*, *cdsAktionen*). Der *CommandType* der Datenmengen wurde auf *ctTable* gesetzt und in *CommandText* wurde die passende Tabelle ausgewählt. Damit entsprechen diese drei Client-Datenmengen übrigens einer direkten Portierung einer BDE-Anwendung, in der die drei Tabellen über die Komponente *TTable* angesprochen wurden. Für größere Datenmengen kann diese simple Übertragung problematisch werden, weil das ClientDataSet die gesamte Tabelle in den Hauptspeicher lädt.
- Anders verhält es sich bei den Datenmengen *cdsTagesTermine*, *cdsTermineZuAdresse* und *cdsWhTermine*. Diese enthalten keine Original-Tabellen der Datenbank, sondern das Ergebnis von SQL-Abfragen, die weiter unten gezeigt sind. Auch für diese drei Datenmengen steht im Datenmodul je eine *TDataSource*-Komponente zur Verfügung.
- Um die Beschriftung und die Spaltenbreite von Feldern in einem DBGrid anzupassen, wurden außerdem in den meisten ClientDataSets persistente Feldkomponenten erzeugt und deren Properties wie etwa *DisplayLabel*, *DisplayWidth* gesetzt. Für die Datenmenge *cdsTermine* wurden außerdem zwei Lookup-Felder erzeugt, die dem Benutzer später im Termineingabe-Dialog (siehe Kapitel 7.5.3) bei der Auswahl von verknüpfter Adresse und Aktion behilflich sein können.
- Außerdem finden Sie in der Abbildung noch die folgenden Komponenten, deren Zweck in den späteren Kapiteln noch genauer erläutert wird: *SQLMonitor1* (Kapitel 7.5.7), das Duo *cdsAdressen2* / *AdressenSrc2* (Kapitel 7.5.5) und die Datenmenge *qryModifyTagesTermine*, die nicht an eine *DataSource* angeschlossen zu werden braucht (Kapitel 7.5.6).

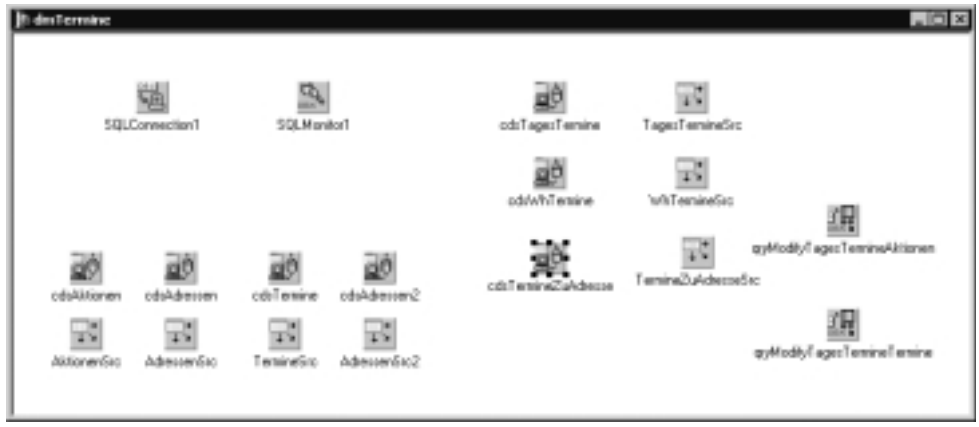


Abbildung 7.19: Das Datenmodul der Terminverwaltung

SQL-Abfragen

Mit SQL-Abfragen können Sie Teilmengen aus einer einzelnen Tabelle oder aus einer Verknüpfung mehrerer Tabellen vom Server abrufen. Die Datensätze, die das Ergebnis der Abfrage bilden, können als neue Datenmenge aufgefasst werden und werden daher in dbExpress auch als *TDataSet* angesehen. Eine generelle Beschreibung von SQL-Abfragen muss spezieller Datenbankliteratur überlassen bleiben; eine ausführliche Referenz finden Sie in der schon erwähnten *Language Reference*, die mit Interbase als PDF-Dokument mitgeliefert wird.

In dbExpress gibt es zwei Möglichkeiten, eine Datenmenge mit dem Ergebnis einer SQL-Abfrage zu verknüpfen:

- Wenn Sie *TSQLQuery* verwenden, können Sie in BDE-kompatibler Weise eine SQL-Abfrage im Property *SQL* angeben (die Kompatibilität bezieht sich auf die *TQuery*-Komponente). Allerdings handelt es sich bei *TSQLQuery* um eine unidirektionale Datenmenge mit allen in Kapitel 7.1.6 genannten Einschränkungen.
- In *TSQLClientDataSet* geben Sie den Abfrage-Text im Property *CommandText* ein, wenn Sie unter *CommandType* den Wert *ctQuery* eingestellt haben. Sobald Sie die so vorbereitete Komponente aktivieren (*Active*-Property oder *Open*-Methode), leitet diese die Abfrage an den SQL-Server und nimmt die von diesem zurückgelieferten Datensätze in ihr Datenpaket *Data* auf (die Datensätze befinden sich dann also alle im Speicher der Delphi-Anwendung).

Für beide Properties stellt Ihnen dbExpress einen *Anweisungstext-Editor* zur Verfügung, der Ihnen die Zusammenstellung der SQL-Anweisung dadurch erleichtert, dass er zwei Listen enthält, aus denen Sie die Namen von Tabellen und deren Feldern per Klick in den Abfragetext einfügen können (siehe Abbildung 7.20).

Hinweis: Es sei noch angemerkt, dass Sie in der Client/Server- bzw. Enterprise-Ausgabe von Delphi die SQL-Anweisung einer *TQuery*-Komponente mit einem leistungsfähigen *SQL-Builder* (in früheren Delphi-Versionen mit dem *Visual Query Builder*) interaktiv zusammenstellen können. Dieser *SQL*-Builder ist wesentlich leistungsfähiger als der SQL-Anweisungseditor von dbExpress. Zwar ist die *TQuery*-Komponente auf BDE-Anwendungen beschränkt, doch können Sie SQL-Anweisungen mit dieser Komponente im SQL-Builder entwerfen und sie danach mit einer *TSQLClientDataSet*-Komponente verwenden, indem Sie den kompletten Anweisungstext in deren *CommandText*-Property kopieren.

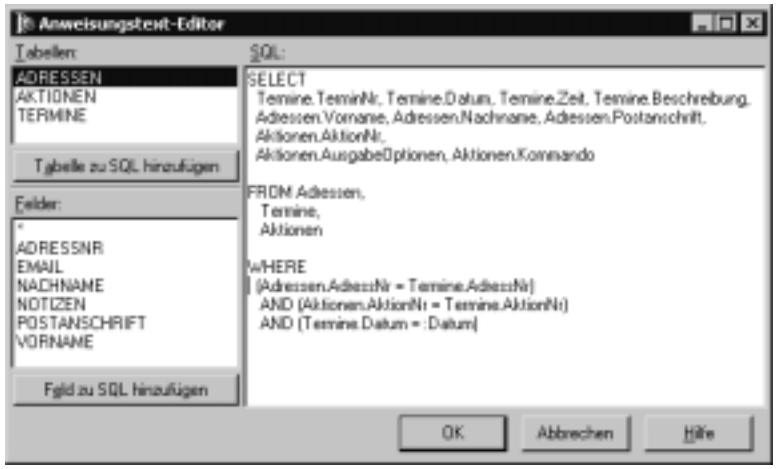


Abbildung 7.20: Eingabe von SQL-Anweisungen für dbExpress-Komponenten

Das Beispielprogramm gibt drei Beispiele für SQL-Abfragen. Zum Einstieg soll jedoch eine minimale Abfrage den grundsätzlichen Aufbau einer solchen Abfrage verdeutlichen:

```
select * from Adressen
```

Diese Abfrage wählt alle Spalten (*) der Adress-Tabelle und enthält keine weiteren Bedingungen zur Auswahl der Datensätze. Folge ist, dass das Ergebnis der Abfrage einfach aus dem gesamten Inhalt der Tabelle *Adressen* besteht. Bei sehr großen Tabellen sollte also genau diese Abfrage mit einem ClientDataSet vorsichtig verwendet werden, denn sie zwingt das ClientDataSet zur Verwaltung eines riesigen Datenpakets.

Die Datenmenge *cdsWhTermine* des Beispielprogramms soll nur die Termine enthalten, die mit einer Aktion verknüpft sind und die in bestimmten Abständen wiederholt werden – bei denen also eines der Felder *WdholenT/M/J* größer als Null ist (die Null bedeutet damit also »keine Wiederholung des Termins«, jedoch ist das eine reine Definitionsfrage). Für die Termine, auf die diese Bedingung zutrifft, sollen neben den Spalten der Termintabelle auch die Felder *Kommando* und *AusgabeOptionen* der zugehörigen Aktion aus der Aktionstabelle ausgegeben werden (und zwar ohne die Verwendung eines Lookup-Feldes wie in der Datenmenge *cdsTermine*). Die dafür erforderliche SQL-Abfrage sieht wie folgt aus:

```
SELECT Termine.Beschreibung, Termine.WdholenT, Termine.WdholenM,
    Termine.WdholenJ, Termine.AktionNr, Termine.AdressNr,
    Aktionen.Kommando, Aktionen.AusgabeOptionen
FROM Termine, Aktionen
WHERE (Termine.AktionNr = Aktionen.AktionNr)
```

```
AND  (    (Termine.WdholenT > 0)
       OR (Termine.WdholenM > 0)
       OR (Termine.WdholenJ > 0) )
```

Hier werden nun hinter *SELECT* alle Spalten aufgelistet, die in der Ergebnismenge enthalten sein sollen. Da in diesem Fall Spalten sowohl aus der Tabelle *Adressen* als auch aus *Aktionen* im Ergebnis vorkommen, ist die Ergebnismenge nicht mehr einer bestimmten Tabelle der Datenbank zuzuordnen, sondern sie stellt eine ganz neue Tabelle dar, in der Inhalte aller drei Tabellen kombiniert werden. Diese neue Tabelle wird nicht in der Datenbank gespeichert, sondern kann in diesem Beispielprogramm nur über eine Abfrage gewonnen werden.

Wenn Sie im ClientDataSet den Felder-Editor zu einer solchen Datenmenge aufrufen, werden Sie sehen, dass Sie für jede dieser Spalten eine eigene persistente Feldkomponente erzeugen können. Alles weitere in Kapitel 7.3 über persistente Felder Gesagte lässt sich somit auf die Ergebnisspalten einer SQL-Abfrage übertragen.

Joins

Im *From*-Teil der obigen Abfrage müssen die beiden Tabellen genannt werden, aus denen im vorherigen *Select*-Teil bereits Spalten genannt wurden. Wenn auf diese Weisen mehrere Tabellen in einer Ergebnismenge zusammengefasst werden, spricht man von einem *Join*. Ein solcher Join ist in der Praxis nur sinnvoll, wenn er durch eine Bedingung eingeschränkt wird, die im *Where*-Teil der Abfrage angegeben wird. Im obigen Listing ist `(Termine.AktionNr = Aktionen.AktionNr)` die Join-Bedingung. Ohne eine Join-Bedingung würde die *Select*-Abfrage alle Kombinationen aus Datensätzen der beiden Tabellen enthalten. Jeder einzelne Termin würde also mit allen in der Aktionstabelle definierten Aktionen kombiniert werden und für eine Datenbank mit 20 Terminen und fünf Aktionen würde eine Ergebnismenge mit 100 Datensätzen generiert werden. Wenn Sie eine formalere Beschreibung von Joins oder weitere Informationen über verschiedene Arten von Joins suchen, finden Sie diese sehr ausführlich in Büchern, die sich allgemein mit der Datenbanktheorie befassen (oben wird übrigens ein »inner join« durchgeführt, was zur Folge hat, dass Termine, die *nicht* mit einer Aktion verknüpft sind, gar nicht in das Ergebnis aufgenommen werden).

Filtern und Sortieren

Schließlich enthält die oben gezeigte Abfrage noch drei weitere Bedingungen, die mit *AND* mit der Join-Bedingung verknüpft werden. Hier werden einfach die *WdholenX*-Felder abgefragt, denn es sollen ja nur die wiederholten Termine abgefragt werden. Solche Bedingungen, die nur auf die Felder einer Tabelle Bezug nehmen, führen zu einer *Filterung*, die von der späteren Filterung in einem ClientDataSet – etwa mit dem *OnFilterRecord*-Ereignis (Kapitel 7.4.1) – zu unterscheiden ist: Während von der SQL-Anweisung ausgefilterte Records gar nicht an die dbExpress-Komponenten gesendet

werden, werden die von Client-Datenmengen ausgefilterten Datensätze nur verborgen, bleiben aber weiterhin im Datenpaket der Datenmenge enthalten.

Sie können außerdem noch eine Sortierung für SQL-Ergebnismengen festlegen, indem Sie eine *Order-By*-Klausel an den *Where*-Teil anfügen. Im Beispielprogramm wurde hiervon kein Gebrauch gemacht und auch unsortierte Ergebnismengen können Sie ja durch ein ClientDataSet beliebig sortieren, indem Sie *IndexFieldNames* auf die zu sortierenden Spalten setzen.

SQL-Abfragen mit Parametern

Die zweite Abfrage-Datenmenge des Beispielprogramms, *cdsTagesTermine*, soll alle Termine eines bestimmten Tages angeben. Dabei sollen statt der Nummer der mit dem Termin verknüpften Adresse die Adressdaten und statt der Aktionsnummer die Aktionsdaten im Ergebnis aufgeführt werden. Die folgende Abfrage verbindet also alle drei Tabellen in einem Join:

```
SELECT Termine.TerminNr, Termine.Datum, Termine.Zeit,
   Termine.Beschreibung, Adressen.Vorname, Adressen.Nachname,
   Aktionen.AktionNr, Aktionen.AusgabeOptionen, Aktionen.Kommando
FROM Adressen, Termine, Aktionen
WHERE
       (Adressen.AdressNr = Termine.AdressNr)
   AND (Aktionen.AktionNr = Termine.AktionNr)
   AND (Termine.Datum = :Datum)
```

Die letzte Bedingung des *Where*-Teils sorgt für die Filterung der Ergebnisliste auf die Termine eines gewählten Tages. Dieser Tag wird in der Abfrage nicht als festes Datum angegeben, da der Benutzer zur Laufzeit wählen soll, für welchen Tag die Termine angezeigt werden sollen. In der Abfrage wird der Tag daher als Parameter (»:Datum«) angegeben, der zur Laufzeit mit dem tatsächlichen Wert ersetzt wird.

Die ClientDataSet-Komponente sucht ihr *CommandText*-Property automatisch nach Parametern ab (die sie ja an dem vorangestellten Doppelpunkt erkennen kann). Alle gefundenen Parameter werden dann der Reihe nach im *Params*-Property aufgeführt, das Sie auch schon zur Entwurfszeit einsehen können.

Folgende wichtige Vorbereitungen für die Verwendung der Parameter können Sie schon zur Entwurfszeit treffen: Öffnen Sie den Property-Editor für das *Params*-Property und stellen Sie für jeden Parameter die folgenden Properties ein: *DataType* für den Datentyp des Parameters und *ParamType* auf *ptInput*, wenn es sich wie hier um die Parameter einer *Select*-Abfrage handelt. (Je nach Datentyp können Sie außerdem noch die Properties *NumericScale*, *Precision* und *Size* einstellen.)

Der Wert des Parameters, den Sie ebenfalls schon zur Entwurfszeit setzen können, wird im Beispielprogramm erst zur Laufzeit gesetzt, und zwar wenn der Benutzer den Schalter *Update* neben dem Eingabefeld für das Datum drückt (siehe Abbildung 7.21):

```
with dmTermine do begin // Datenmodul
  cdsTagesTermine.Close;
  cdsTagesTermine.Params[0].AsString := DateEdit.Text;
  cdsTagesTermine.Open;
end;
```

Die Änderung der Parameter ist nur bei geschlossener Datenmenge möglich, daher wird vorher die *Close*-Methode aufgerufen. Das abschließende *Open* führt dazu, dass die Abfrage mit dem geänderten Parameterwert neu gestartet wird und im Falle des Beispielprogramms die Termine eines anderen Tages angezeigt werden.

Hinweis: Ein solcher Parameter wird auch benötigt, wenn Sie die Detailtabelle eines Haupt-/Detailformulars per SQL generieren. So könnte beispielsweise eine Detailtabelle, die alle Termine anzeigt, welche eine bestimmte Aktion referenzieren, wie folgt aufgebaut werden:

```
SELECT ...
   FROM Termine
   WHERE Termine.AktionNr = :AktionNr
```

Der Parameter, hier *AktionNr*, ist vom aktuellen Datensatz in der Aktionstabelle abhängig und muss auch bei jeder Änderung der Position in dieser Tabelle geändert werden. Wenn Sie *MasterSource* und *MasterFields* wie in Kapitel 7.4.5 einstellen, sorgt ein ClientDataSet automatisch für das Setzen dieses Parameters und für die Aktualisierung der Detail-Abfrage.

Aufgaben eines Datenmoduls *R183*

Außer den Komponenten kann ein Datenmodul natürlich auch noch Code enthalten und die Datenzugriffskomponenten bieten ja auch zahlreiche Ereignisse an, die im Datenmodul bearbeitet werden können. Anhand der Beispielanwendung seien hier nun ein paar dieser Ereignisse vorgestellt:

- Zunächst bietet sich das *OnCreate*-Ereignis des Datenmoduls dazu an, die Verbindung zu aktivieren (*TSQLConnection.Connect*) und die Datenmengen zu öffnen (*TDataSet.Open*), die während der gesamten Programmlaufzeit benötigt werden. Das Beispielprogramm öffnet bei diesem Ereignis alle seine Client-Datenmengen.
- Das Ereignis *OnNewRecord* einer Client-Datenmenge eignet sich dazu, bestimmte Felder mit Standardeinstellungen oder automatisch errechneten Werten vorzubelegen (z.B. mit dem aktuellen Datum). Das Beispielprogramm verknüpft dieses

Ereignis bei der Client-Datenmenge für die Termine mit der folgenden Methode, die mit Hilfe der persistenten Feldkomponenten die Werte einiger Spalten initialisiert:

```
procedure TdmTermine.cdsTermineNewRecord(DataSet: TDataSet);
begin
  cdsTermineWDHOLENT.Value := 0;
  cdsTermineWDHOLENM.Value := 0;
  cdsTermineWDHOLENJ.Value := 0;
  cdsTermineZEIT.AsDateTime := 0;
end;
```

Da in der Definition der Termindatenbank nicht vorgeschrieben wurde, dass diese Felder einen Wert haben müssen, würde das Programm auch ohne diese Methode gut funktionieren. Der Benutzer könnte dann die Felder einfach offen lassen und die datensensitiven Komponenten würden dann für diese Felder keine Null, sondern einfach ein leeres Feld anzeigen (im Programm lässt sich dieser Fall übrigens mit der Methode *IsNull* von *TField* abfragen).

- Das Ereignis *OnGetText* der Feldkomponenten kann dazu verwendet werden, den in den datensensitiven Komponenten angezeigten Text zu modifizieren. Die folgende Methode des Beispielprogramms sorgt beispielsweise dafür, dass für das Feld *Zeit* nur der Zeitanteil des Feldwertes angezeigt und der Datumsanteil ignoriert wird:

```
procedure TdmTermine.cdsTagesTermineZEITGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  if not Sender.IsNull then
    DateTimeToString(Text, 'hh:mm', Sender.AsDateTime);
end;
```

Der Inhalt des Feldes selbst wird dadurch zunächst nicht beeinflusst. Erst wenn der Benutzer den angezeigten Wert editiert und die Änderungen speichert, wirkt sich unter Umständen aus, welcher Text von *GetText* angezeigt wurde. (Entsprechend zu *OnGetText* gibt es natürlich auch ein Ereignis *OnSetText*, in dem Sie den vom Benutzer eingegebenen Wert interpretieren und eventuell vor dem Setzen des Feldes verändern können).

- Das Ereignis *AfterPost* einer Client-Datenmenge eignet sich sehr gut dazu, die ApplyUpdates-Methode aufzurufen, um die Änderungen des Benutzers auch an den Datenbank-Server zu übertragen (siehe Kapitel 7.5.4).
- Die Bearbeitung des Ereignisses *TClientDataSet.OnReconcileError* ist in Kapitel 7.5.5 beschrieben.

- Für die Bearbeitung des Events *BeforeUpdateRecord* (auch ein Ereignis der Client-Datenmengen) im Beispielprogramm siehe Kapitel 7.5.6.

Die Zusammenfassung solcher Ereignisbearbeitungen in einem Datenmodul kann dazu führen, dass der datenbankabhängige Code Ihrer Anwendung leicht zu warten ist, da alle relevanten Methoden sich an einem gemeinsamen Ort befinden. Außerdem werden auf diese Weise die Formular-Units entlastet. Sie müssen sich nicht um die Initialisierung neuer Records und auch kaum noch um die Anzeige der Daten oder die Aktualisierung kümmern. Manche Formulare des Beispielprogramms kommen daher fast ohne eigenen Code aus.

7.5.3 Die Formulare

Das Hauptfenster der Beispielanwendung (Abbildung 7.21) enthält zwei spezielle Ansichten der Termindatenbank, die über die beiden im letzten Kapitel beschriebenen SQL-Abfragen *cdsTagesTermine* und *cdsWhTermine* gewonnen werden. Ohne dass dafür im Hauptformular weitere Programmierung notwendig wäre, werden beide Datenmengen einfach durch je eine *DBGrid*-Komponente angezeigt. Die Anzeige der Daten in diesem Grid richtet sich nach den Properties der persistenten Feldkomponenten und nach dem Wert, den die im letzten Kapitel beschriebene *OnGetText*-Methode zurückliefert. Da sich sowohl die Feldkomponenten als auch die *OnGetText*-Bearbeitung im Datenmodul befinden, muss sich das Hauptformular gar nicht um diese Angelegenheit kümmern.

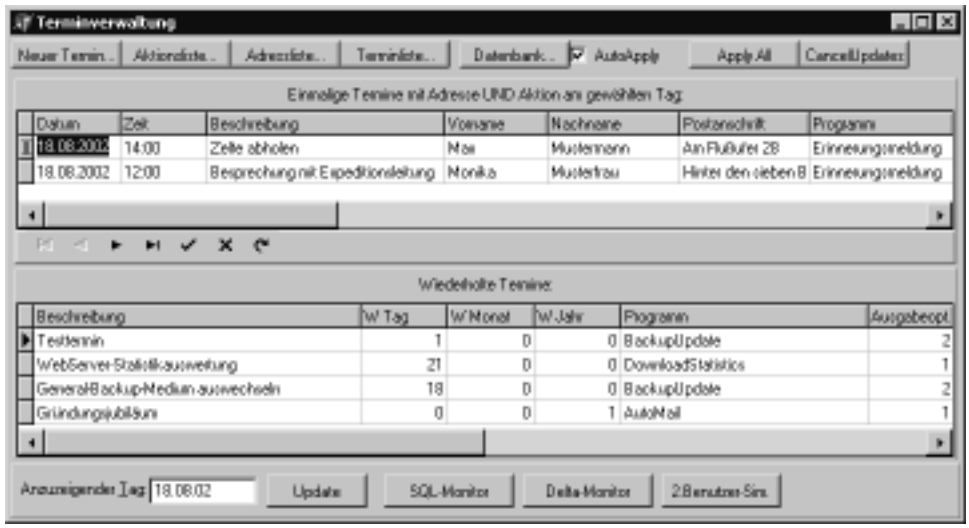


Abbildung 7.21: Das Hauptformular zeigt das Ergebnis der SQL-Abfragen und dient als Startpunkt für die Formulare zum Editieren der einzelnen Tabellen.

Außer dem Ansehen dieser Datenmengen hat der Benutzer noch die Möglichkeit, das erste DBGrid mit gewissen Einschränkungen zu editieren. Dies stellt für das Programm eine besondere Herausforderung dar, die in Kapitel 7.5.6 genauer beschrieben wird.

Ansonsten dient das Hauptformular hauptsächlich zum Öffnen der anderen Fenster. Einige Methoden des Hauptformulars werden in den folgenden Kapiteln noch besprochen (die Reaktion auf den Schalter *Update* neben dem Datumsfeld wurde bereits in Kapitel 7.5.2 erklärt).

Modale Dateneingabe-Fenster

Die Schalter AKTIONSLISTE... und TERMINLISTE... des Hauptformulars öffnen jeweils ein einfaches modales Fenster, in dem die jeweilige Tabelle editiert werden kann. Die Formulare für diese beiden Fenster bringen keine Neuerungen, was die in ihnen enthaltenen Komponenten betrifft, so dass hier nicht weiter auf diese eingegangen werden soll.

Eine grundsätzliche Überlegung wert ist jedoch, wie die modale Anzeige eines solchen Fensters mit dem Aktualisierungsprozess einer Datenmenge harmoniert. An dieser Stelle soll es nur um das Änderungsprotokoll gehen, also um die Frage, wie die Daten mit *Post* in das Änderungsprotokoll geschrieben werden. Kapitel 7.5.4 erläutert später, wie das Eintragen der Änderungen für alle Datenmengen des Programms an einer zentralen Stelle vorgenommen wird, so dass sich die einzelnen Formulare nicht mehr darum zu kümmern brauchen.

Wenn der Benutzer nun in einem modalen Fenster eine Dateneingabe-Komponente editiert, kann er das Fenster normalerweise über einen OK- oder einen ABBRUCH-Schalter schließen. Diese Schalter sind nicht »datensensitiv«, haben also keine Ahnung von der Datenbank und rufen auch nicht etwa *Post* oder *Cancel* auf, wie es zum Abschluss eines Dateneingabedialogs empfehlenswert wäre. Daher werden beide Schalter im Beispielprogramm wie folgt bearbeitet:

```
procedure TfrmAdressListe.OkButtonClick(Sender: TObject);
begin
  if dmTermine.cdsAdressen.State = dsEdit then
    dmTermine.cdsAdressen.Post;
end;

procedure TfrmAktionsListe.CancelButtonClick(Sender: TObject);
begin
  if dmTermine.cdsAdressen.State = dsEdit then
    dmTermine.cdsAdressen.Cancel;
end;
```

Post bzw. *Cancel* brauchen nur aufgerufen zu werden, wenn sich die Datenmenge im Editiermodus befindet.

Die Adressliste

Die Adresstabelle wird über ein drittes Formular zur Eingabe bereitgestellt, das Sie in Abbildung 7.22 sehen können. Links oben befindet sich zunächst ein DBGrid, das nur die Spalten *Vorname* und *Nachname* des Adressen-ClientDataSets anzeigt (die anderen Spalten wurden aus dem Property *TDBGrid.Columns* gelöscht). Es dient als Namensliste, damit der Benutzer schneller einen bestimmten Namen auswählen kann, dessen Adressdaten er dann in den rechts daneben befindlichen Einzel-Steuerelementen ansehen und editieren kann.

Das untere DBGrid gibt eine Ansicht auf die Datenmenge *cdsTermineZuAdresse*, die alle mit der aktuellen Adresse verknüpften Termine enthält. Das Adressformular wird dadurch zu einem Haupt-/Detailformular, wobei die Details nur zur Information angezeigt werden, also nicht editierbar sind.

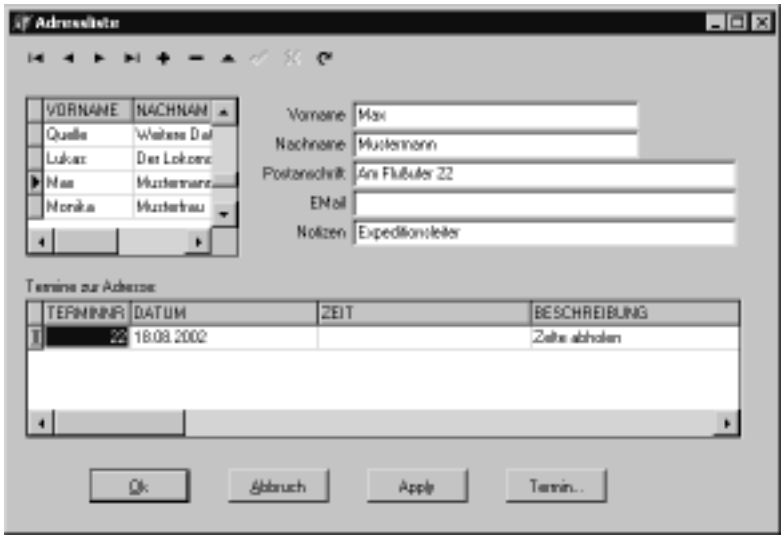


Abbildung 7.22: Das Formular zum Editieren der Adressliste

Da auch das Adressformular seine Funktion hauptsächlich auf die selbsttätige Arbeitsweise der datensensitiven Komponenten stützt, bleibt hier nur eine erwähnenswerte Aufgabe des Formulars übrig: Der Aufruf des Termin-Dialogs über den Schalter TERMIN.

Im Gegensatz zum Aufruf des Termindialogs aus dem Hauptformular soll über den TERMIN-Schalter im Adressformular ein Termin speziell für die gerade bearbeitete Adresse eingegeben werden können. Die Verknüpfung mit der Adresse soll automa-

tisch erfolgen, so dass dem Benutzer die Eingabe des Adressfeldes des neuen Termins erspart bleibt. Die folgende Methode setzt diesen Plan in die Tat um:

```
procedure TfrmAdressListe.Button3Click(Sender: TObject);
begin
  dmTermine.cdsTermine.Insert;
  dmTermine.cdsTermineADRESSNR.Value :=
    dmTermine.cdsAdressen.FieldByName('ADRESSNR').Value;
  if frmNeuerTermin.ShowModal = mrOk then
    dmTermine.cdsTermine.Post
  else dmTermine.cdsTermine.Cancel;
end;
```

Sie sorgt mit der schon aus Kapitel 7.2.3 bekannten *Insert*-Methode dafür, dass ein neuer Datensatz in das Termin-ClientDataSet eingefügt wird, und setzt dessen Feld *AdressNr* auf die Adressnummer des aktuellen Datensatzes im Adress-ClientDataSet. Nach dieser Vorbereitung wird der Termindialog ganz normal mit *ShowModal* aufgerufen, wie es auch beim Aufruf des Dialogs aus dem Hauptformular geschieht.

Der Termindialog

Beim Termindialog (Abbildung 7.1.3) lassen sich zwei Besonderheiten feststellen: Zum einen enthält er das für dieses Buch einzige Anwendungsbeispiel für *TDBLookupComboBox*-Komponenten, die die im Felder-Editor definierten Lookup-Felder der Komponente *cdsTermine* anzeigen. Sie wurden wie jede andere datensensitive Komponente einfach über ihr *DataField*-Property mit dem entsprechenden Feld verknüpft und ersparen dem Benutzer, die mit dem Termin verknüpfte Aktion und Adresse über eine Nummer eingeben zu müssen.

Zum anderen braucht für diesen Dialog kein einziges Ereignis bearbeitet zu werden, da alles automatisch abläuft oder (im Falle des Aufrufs von *Post* und *Cancel*) an den Aufrufer des Dialogs delegiert wird (siehe Listing oben).

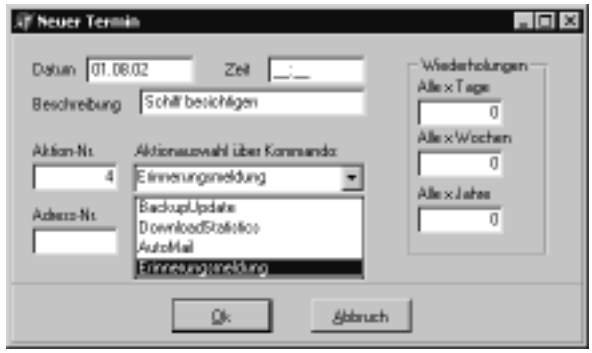


Abbildung 7.23: Ein Dialog zur Eingabe eines neuen Termins

Die weiteren Formulare der Anwendung werden in den folgenden Kapiteln noch zur Sprache kommen.

7.5.4 Updates und das Änderungsprotokoll

In Kapitel 7.2.7 wurde bereits erläutert, wie die Änderungen an dem Datensatz einer SQLClientDataSet-Komponente zuerst mit *Post* in das Änderungsprotokoll geschrieben werden und dort verweilen, bis sie entweder mit *CancelUpdates* verworfen oder mit *ApplyUpdates* in die Datenbank geschrieben werden. Während die Browser-Beispielprogramme aus Kapitel 7.2 sich noch gar nicht um die Änderungen gekümmert haben und die Dateidatenbank aus Kapitel 7.3.6 das Einfügen aller Datensätze selbst vornimmt, hat die aktuelle Beispielanwendung erstmals die Aufgabe zu erfüllen, Änderungen des Benutzers an den Datenbank-Server weiterzuleiten.

Einfache Updates *R181*

Eine besonders einfache Möglichkeit, sicherzustellen, dass alle Änderungen des Benutzers gespeichert werden, besteht darin, *ApplyUpdates* immer dann aufzurufen, wenn der Benutzer die Änderung eines Datensatzes abschließt (wenn er also eine *Post*-Operation verursacht – entweder implizit, indem er zu einem anderen Datensatz wechselt, oder explizit, indem er etwa den Post-Schalter einer *DBNavigator*-Komponente betätigt).

Eine solche Verknüpfung ist über das Ereignis *TDataSet.AfterPost* möglich. Im Minimalfall würde es bereits genügen, hier einfach nur *ApplyUpdates* aufzurufen. Im Datenmodul der Beispielanwendung sind die *AfterPost*-Ereignisse *aller* editierbaren Client-Datenmengen mit der folgenden Methode verknüpft, die quasi eine Luxus-Version des *ApplyUpdates*-Aufrufs darstellt:

```
procedure TdmTermine.cdsGeneralAfterPost(DataSet: TDataSet);
var
  ErrorCount: Integer;
begin
  with DataSet as TSQLClientDataSet do begin
    if AutoApply then begin
      ErrorCount := ApplyUpdates(-1);
      If ErrorCount <> 0 then
        ShowMessage(Format('%d Datensätze wurden nicht eingetragen.
            Refresh wird nicht durchgeführt.', [ErrorCount]))
      else
        Refresh;
    end;
  end;
end;
```

Zunächst wird *ApplyUpdates* nicht für ein bestimmtes ClientDataSet aufgerufen, sondern für die im Parameter *DataSet* übergebene Komponente (der *DataSet*-Parameter ist mit dem *Sender*-Parameter von Steuerelement-Ereignissen vergleichbar). Diese Unabhängigkeit von einer bestimmten Datenmenge ermöglicht erst die Verknüpfung dieser Methode mit den *AfterPost*-Ereignissen mehrerer Datenmengen (diese Datenmengen müssen hier aber vom Typ *TSQLClientDataSet* sein, da die obige Methode den *DataSet*-Parameter mit *as* in diese Klasse umzuwandeln versucht).

Dann berücksichtigt die Methode eine interne Variable namens *AutoApply*, über die das automatische Annehmen der Änderungen abgeschaltet werden kann. Diese Möglichkeit dient hier zum Experimentieren und zum Vergleichen und wird gleich bei der Untersuchung des Änderungsprotokolls eine wichtige Rolle spielen. Im Hauptformular des Programms finden Sie eine CheckBox, über die Sie *AutoApply* zur Laufzeit verändern können.

Die ApplyUpdates-Methode

Als Nächstes findet der entscheidende Aufruf von *ApplyUpdates* statt. Als Parameter erwartet diese Methode die maximale Zahl von Fehlern, die auftreten darf, bevor der Aktualisierungsvorgang abgebrochen wird. Sollte es zu einem solchen Abbruch kommen, werden alle vorher bereits erfolgreich im Server eingetragenen Änderungen zurückgenommen (hierfür wird intern von dem Transaktionsmechanismus der SQL-Server Gebrauch gemacht). Im Beispielprogramm ist ein solcher Abbruch des Aktualisierungsvorgangs auszuschließen, da als Parameter »-1« übergeben wird, also beliebig viele Fehler auftreten dürfen.

Obwohl Sie als Parameter statt »-1« beliebige andere Zahlen angeben können, stehen sich hierfür hauptsächlich zwei gegensätzliche Prinzipien gegenüber:

- Ein Parameter von »-1« bewirkt, dass so viele geänderte Datensätze wie möglich in die Datenbank übernommen werden. Wenn es bei einem bestimmten Datensatz zu einem Fehler kommt, wird dieser eine Datensatz nicht eingetragen, jedoch wird das Eintragen anderer Datensätze nicht verhindert.
- Ein Parameter von »0« lässt die Methode nach dem Alles-oder-nichts-Prinzip vorgehen: Entweder werden alle Datensätze in die Datenbank geschrieben oder keiner (hier kommt der erwähnte Transaktionsmechanismus zum Tragen, indem beim ersten Update-Fehler die Eintragung der bisher schon erfolgreich eingetragenen Datensätze rückgängig gemacht wird).

Die Zahl der aufgetretenen Fehler wird in jedem Fall als Ergebnis von *ApplyUpdates* zurückgeliefert und von der obigen Methode in einem Meldungsfenster angezeigt (falls von Null verschieden).

Wie es überhaupt zu Aktualisierungsfehlern kommen kann und wie sie behandelt werden können, ist Thema von Kapitel 7.5.5.

Hinweis: Da in Kapitel 7.5.3 die *Ok*-Schalter der modalen Formulare mit einem *Post*-Aufruf verknüpft wurden, wird bei jedem normalen Schließen dieser Formulare auch die obige Methode aufgerufen. Sofern Sie also zur Laufzeit die *AutoApply*-Option nicht abschalten, ist die Termindatenbank nach dem Schließen eines modalen Eingabefensters immer auf dem neuesten Stand.

Erneuerung der Daten nach ApplyUpdates

An dieser Stelle interessiert noch der abschließende Aufruf von *Refresh* im obigen Listing. Dieser sorgt dafür, dass alle Ansichten der aktualisierten Datenmenge ebenfalls aktualisiert werden. Dabei werden sowohl Änderungen am Datensatz, die auf der Seite des Servers durchgeführt werden, sichtbar (wie etwa die in Kapitel 7.5.1 beschriebene automatische Nummerierung durch Trigger) als auch Änderungen, die vom Benutzer zur Behandlung von Update-Fehlern im Reconcile-Dialog veranlasst wurden (siehe Kapitel 7.5.5).

Das Änderungsprotokoll einsehen

Eines der »Experimental-Formulare« des Beispielprogramms befasst sich mit dem Änderungsprotokoll der Client-Datenmengen. Es erklärt auch den Daseinszweck der im letzten Listing abgefragten Variable *AutoApply*. Wenn Sie *AutoApply* abschalten, werden die Änderungen nicht mehr automatisch an den Server geschickt, sondern häufen sich so lange im Änderungsprotokoll an, bis Sie einen der *Apply*-Schalter drücken, die das Beispielprogramm extra für diesen Zweck vorsieht.

Hinweis: Nach der Eingabe eines *neuen* Datensatzes muss in diesem Beispielprogramm auf jeden Fall ein *ApplyUpdates* durchgeführt werden, bevor ein weiterer neuer Datensatz eingegben wird, denn erst beim Senden des Datensatzes an den Server erhält dieser dort durch die Trigger-Funktion eine eindeutige Nummer zugeordnet. Wenn Sie zwei neue Datensätze hintereinander eingeben, ohne dass der erste eine Nummer erhalten hat, resultieren zwei Datensätze mit gleichem Inhalt in der Nummernspalte (in beiden Fällen »leer« bzw. *Null*). Da jedoch ein eindeutiger Index auf die Nummernspalte gelegt wurde, wird bei einem doppelten Wert eine Indexfehler-Exception ausgelöst. Wenn Sie also *AutoApply* abgeschaltet haben, sollten Sie nach der Eingabe eines neuen Datensatzes manuell den *Apply*-Schalter des jeweiligen Formulars drücken.

Das in Abbildung 7.24 gezeigte Formular ermöglicht zur Laufzeit einen Einblick in das Änderungsprotokoll; es ist nur dann sinnvoll zu gebrauchen, wenn das automatische

Apply abgeschaltet ist, denn sonst werden die Änderungsprotokolle ja immer automatisch geleert, wenn einer der modalen Dialoge der Anwendung geschlossen wird.

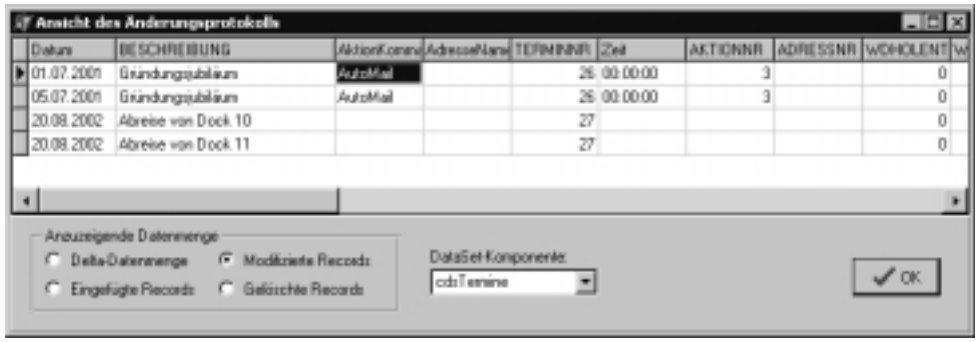


Abbildung 7.24: Ansicht der geänderten Datensätze über die ClientDataSet-Option »StatusFilter = usModified«. Jeder geänderte Datensatz wird zwei Mal angezeigt: einmal für den Zustand vor der Änderung, einmal für den Zustand danach.

Das Formular ist so allgemein gehalten, dass Sie es auch leicht in eigene Anwendungen einbauen können. Der Aufruf des Formulars im Beispielprogramm (über den Schalter DELTA-MONITOR) läuft wie folgt ab:

```
procedure TfrmTermine.Button3Click(Sender: TObject);
begin
  DeltaView.CDSContainer := dmTermine;
  DeltaView.ShowModal;
end;
```

Vor der Anzeige des Formulars wird dessen Variable *CDSContainer* auf ein Datenmodul gesetzt, das ClientDataSets enthält. Diese ClientDataSets können dann bei der folgenden Verwendung des Formulars aus einer KomboBox ausgewählt werden. Diese spezielle Funktion soll aber hier nicht untersucht werden, sondern wir nehmen nun an, dass bereits eine der Datenmengen ausgewählt und in der Variablen *CurrentDataSet* gespeichert wurde.

Über die Radioschaltergruppe des Formulars können Sie nun zwischen vier Ansichten wählen, wobei die letzten drei zusammengehören. In der ersten Ansicht soll der Inhalt des *Delta*-Properties der gewählten Datenmenge dargestellt werden. Auf direktem Weg ist das nicht möglich, da die datensensitiven Komponenten immer nur den Inhalt des *Data*-Properties anzeigen (dem vom ClientDataSet verwalteten Datenpaket). Da jedoch *Data* und *Delta* zuweisungskompatibel sind, können wir das *Delta*-Property unserer Datenmenge an das *Data*-Property einer anderen Datenmenge zuweisen. Bei dieser Zuweisung erhält diese zweite Datenmenge eine *Kopie* des *Delta*-Inhalts, aller-

dings in ihrem *Data*-Property, so dass der Inhalt des Datenpakets nun in datensensitiven Komponenten sichtbar gemacht werden kann.

Das DeltaView-Formular enthält zu diesem Zweck eine eigene Komponente *ClientDataSet1*, die mit dem Inhalt des in Frage stehenden *Delta*-Properties gefüllt werden kann. *Delta* ist allerdings nur gültig, wenn die Datenmenge aktiv ist (*Active = True*) und Änderungen vorgenommen wurden (*ChangeCount > 0*). Dies wird im Folgenden vorher abgefragt:

```
// Typ von CurrentDataSet: TSQLClientDataSet
procedure TDeltaView.UpdateView;
begin
  if Assigned(CurrentDataSet) then
    if rgViewType.ItemIndex = 0 then // Index der RadioGroup
    begin
{$B-} if (CurrentDataSet.Active = True) and
         (CurrentDataSet.ChangeCount > 0) then
      begin
        ClientDataSet1.Data := CurrentDataSet.Delta;
        // DataSource1 leitet die Daten an das DBGrid weiter:
        DataSource1.DataSet := ClientDataSet1;
      end;
    end else begin
```

Die drei anderen Ansichten von *DeltaView* kommen ohne das *Delta*-Property aus, sie beziehen sich aber dennoch auf die Änderungen, die in der Client-Datenmenge durchgeführt wurden. *TClientDataSet* erlaubt nämlich über das Property *StatusFilter*, die Anzeige auf die geänderten Datensätze zu beschränken. *StatusFilter* kann ein beliebige Kombination von *usModified* (geänderte Datensätze), *usInserted* (neu eingefügte Datensätze) und *usDeleted* (gelöschte Datensätze) enthalten. Im Programm wird jeweils nur eine dieser Konstanten gesetzt:

```
      // (Fortsetzung der Methode UpdateView)
      DataSource1.DataSet := CurrentDataSet;
      if CurrentDataSet.Active = True then
        case rgViewType.ItemIndex of
          1: CurrentDataSet.StatusFilter := [usModified];
          2: CurrentDataSet.StatusFilter := [usInserted];
          3: CurrentDataSet.StatusFilter := [usDeleted];
        end;
    end;
end;
```

In der Praxis können Sie die *StatusFilter*-Option dazu verwenden, dem Benutzer eine Rückmeldung darüber zu geben, welche Änderungen er zuletzt durchgeführt hat bzw. welche Änderungen bei der nächsten Aktualisierung an den Server geschickt werden

würden. Dies ist natürlich besonders interessant bei Anwendungen nach dem Aktenkoffermodell, bei denen sich über einen längeren Zeitraum Änderungen im Protokoll anhäufen.

7.5.5 Mehrbenutzersimulation

Das Eintragen von Änderungen auf dem Datenbank-Server ist ein kritisches Unterfangen, weil in der Praxis häufig mehrere Anwender mit derselben Datenbank arbeiten und man es daher nicht ausschließen kann, dass zwei oder mehr Anwender zur selben Zeit den gleichen Datensatz ansehen und sich entschließen, ihn zu ändern.

Programmtechnisch sieht das dann beispielsweise so aus, dass auf dem Rechner jedes Anwenders eine eigene ClientDataSet-Komponente arbeitet, die eine Kopie des Datensatzes im Speicher aufbewahrt. Wenn nun der erste Anwender den Datensatz ändert und zum Server zurückschickt, aktualisiert dieser erwartungsgemäß die Datenbank entsprechend. Wenn aber nun der zweite Anwender kurze Zeit später seine eigenen Änderungen abschickt, so könnten dadurch Änderungen des ersten Anwenders überschrieben werden, die dem zweiten Anwender noch gar nicht angezeigt wurden.

Außerdem könnte es sein, dass beide Anwender nur ein einzelnes Feld geändert haben, und zwar jeweils ein anderes. Soll der Datenbank-Server dann die beiden geänderten Felder behalten oder den Datensatz so akzeptieren, wie ihn der zweite Anwender abgeschickt hat (ohne die Änderung des ersten Anwenders)?

Ein zweites Adresslisten-Formular

Im Beispielprogramm können Sie diese Situation simulieren, indem Sie den Schalter 2-BENUTZER-SIM. drücken. Sie erhalten dann neben dem schon in Kapitel 7.5.3 beschriebenen Adresseingabe-Formular noch ein zweites Fenster, das ebenfalls eine Adressliste enthält. Wichtig ist, dass beide Fenster nicht über dieselbe ClientDataSet-Komponente auf die Tabelle zugreifen. Hierfür wird nun die Datenmenge *cdsAdressen2* benötigt, die in Kapitel 7.5.2 schon als Teil des Datenmoduls gezeigt wurde.

Diese Datenmenge lädt wie *cdsAdressen* die gesamte Adresstabelle und verfügt über ein eigenes Änderungsprotokoll. Sie können nun, wenn Sie die beiden Fenster aufgerufen haben, in beiden Änderungen am gleichen Datensatz vornehmen, ohne eines der Fenster zu schließen oder einen *Apply*-Schalter zu drücken. Erst nachdem Sie den gleichen Datensatz zweifach geändert haben, drücken Sie in beiden Fenstern den *Apply*-Schalter. Beim zweiten *Apply*-Versuch sollte es nun zu einem Aktualisierungs-Konfikt kommen.

Der Reconcile-Dialog

Wie Update-Fehler am besten behoben werden, hängt vom konkreten Anwendungsfall ab. Am flexibelsten ist es, dem Benutzer die Wahl zu lassen, was im Konfliktfall geschehen soll. In der Objektablage von Delphi finden Sie bereits einen vorgefertigten Dialog, der dafür verwendet werden kann (wenn Sie eine VCL-Anwendung geladen haben, finden Sie den Dialog auf der Seite *Dialoge* unter dem Namen *Dialogfeld zum Beheben von Fehlern*[19], bei CLX-Anwendungen unter dem Namen *CLX Dialogfeld Fehler beheben*); in Abbildung 7.25 ist dieser Dialog in Aktion zu sehen.

Dieser Dialog wurde mit der voreingestellten KOPIEREN-Option[20] der Objektablage in das Beispielprogramm eingefügt und dann im Verzeichnis des Beispielprogramms gespeichert. Dieses verfügt also über eine eigene Kopie des Dialogs, die unabhängig von der in der Objektablage befindlichen Kopie verändert werden kann (im Beispielprogramm wurde allerdings keine Veränderung vorgenommen).

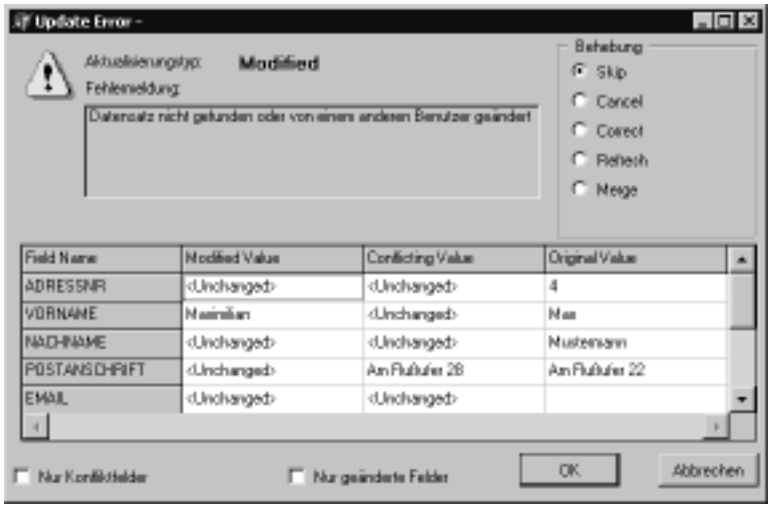


Abbildung 7.25: Der von Borland mitgelieferte Dialog zur Behandlung von Update-Konflikten

Die Verwendung des Dialogs ist sehr einfach, denn die Dialog-Unit (im Beispielprogramm unter dem Namen *RecError* gespeichert) enthält bereits eine vordefinierte Funktion zum Behandeln eines Update-Konflikts. Der Update-Konflikt wird zunächst

19 Im Test des Autors wurde der VCL-Dialog erst sichtbar, nachdem dieser Dialog in den EIGENSCHAFTEN der Objektgalerie auf eine existierende Galerie-Seite verlagert wurde.

20 Hier ist der KOPIEREN-Radioschalter im DATEI | NEU (| WEITERE)-Dialog gemeint (siehe Kapitel 1.6.4).

in Form eines *OnReconcileError*-Ereignisses der verantwortlichen Datenmenge angezeigt. Dieses Ereignis wird nun wie folgt bearbeitet:

```
// uses RecError;
procedure TdmTermine.cdsAdressenReconcileError(
  DataSet: TCustomClientDataSet; E: EReconcileError;
  UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Wie in Abbildung 7.25 gezeigt, kann der Benutzer so zur Laufzeit zwischen verschiedenen Möglichkeiten wählen, wie der Konflikt zu beheben ist. Wenn Sie *OnReconcileError* nicht bearbeiten, bekommt der Benutzer im Fehlerfall nur eine Exception-Meldung und seine Änderungen werden nicht auf dem Server eingetragen.

Im Beispielprogramm wurde die oben gezeigte Methode sowohl mit *cdsAdressen* als auch mit *cdsAdressen2* verknüpft, da der Update-Konflikt in beiden Datenmengen auftreten kann, je nachdem, in welcher Reihenfolge die *ApplyUpdates* ausgeführt werden. Die anderen Datenmengen des Programms verfügen, unter der Annahme, dass sie jeweils nur von einem Benutzer gleichzeitig gesehen werden, nicht über eine *OnReconcileError*-Behandlung.

Hinweis: Während Sie einen Update-Konflikt als Ereignis bearbeiten, wird ein solcher Konflikt innerhalb der VCL als Exception weitergeleitet. Wenn Sie die Option TOOLS | DEBUGGER-OPTIONEN... | SPRACH-EXCEPTIONS | BEI DELPHI-EXCEPTIONS STOPPEN nicht deaktivieren, hält Delphi Ihr Programm daher bei einem Update-Konflikt in jedem Fall an, auch wenn das *OnReconcileError*-Ereignis vorbildlich behandelt wird (sie könnten zwar unter TOOLS | DEBUGGER-OPTIONEN | SPRACH-EXCEPTIONS die Exception *EDatabaseError* auf die Ignorieren-Liste setzen, dies würde aber dazu führen, dass der Debugger auch alle anderen Datenbank-Exceptions ignorieren würde).

7.5.6 Updates per SQL

Dieser Abschnitt soll Ihnen einen kurzen Ausblick auf ein Thema geben, das an sich eine ausführliche Behandlung verdienen würde, die jedoch hier der Spezialliteratur überlassen werden muss. Es geht um Datenmengen, deren Änderungen nicht automatisch von *ApplyUpdates* im Server eingetragen werden können.

Ein Beispiel für eine solche Datenmenge ist die obere Terminliste im Hauptformular des Beispielprogramms (Abbildung 7.21). In ihr sind Felder aus allen drei Tabellen der Anwendung kombiniert. Wird nun ein Datensatz dieser Tabelle geändert, müssten im Extremfall Änderungen in allen drei Tabellen eingetragen werden. Das Standardverhalten der VCL besteht jedoch darin, die Änderungen aller Felder an die erste

Tabelle zu senden, die im *From*-Teil der *Select*-Anweisung aufgeführt ist. Da diese Tabelle jedoch nicht alle Spalten der Abfrage enthält, muss es hierbei zu einem Fehler kommen.

Eigene Updates durchführen

Wenn es nicht möglich ist, die intern von der VCL verwendete Update-Anweisung zu korrigieren, wie es etwa weiter unten unter *Updates für eine einzelne Tabelle* angesprochen wird, verbleibt noch die radikalste Lösung: Sie besteht darin, das Update komplett selbst durchzuführen, also auch die SQL-Update-Anweisung selbst zu generieren. *TClientDataSet* ermöglicht dies über das Ereignis *BeforeUpdateRecord*, das für jeden zu aktualisierenden Datensatz vor der Aktualisierung ausgelöst wird. In der Methode für dieses Ereignis können Sie nun die Aktualisierung selbst durchführen und den Parameter *Applied* auf *True* setzen, um dem ClientDataSet mitzuteilen, dass die Arbeit erledigt ist. Bei der Aktualisierung muss der Parameter *UpdateKind* berücksichtigt werden, der angibt, ob es sich um eine Änderungs-, eine Lösch- oder eine Einfügeoperation handelt:

```
procedure TdmTermine.cdsTagesTermineBeforeUpdateRecord(Sender: TObject;
  SourceDS: TDataSet; DeltaDS: TCustomClientDataSet;
  UpdateKind: TUpdateKind; var Applied: Boolean);
begin
  if UpdateKind = ukModify then
    ... Aktualisierung der Datensatzänderung auf dem Server
  else if UpdateKind = ukDelete then... // Löschen auf dem Server
  else if UpdateKind = ukInsert then... // Einfügen auf dem Server
  Applied := True;
end;
```

Bevor wir uns ansehen, wie dies im Beispielprogramm konkret im Fall von *ukModify* funktioniert, sei noch einmal der entscheidende Teil der *Select*-Anweisung gezeigt, durch den die Datenmenge *cdsTagesTermine* aufgebaut wird:

```
SELECT Termine.TerminNr, Termine.Datum, Termine.Zeit,
   Termine.Beschreibung, Adressen.Vorname, Adressen.Nachname,
   Adressen.Postanschrift, Aktionen.AktionNr,
   Aktionen.AusgabeOptionen, Aktionen.Kommando
FROM Adressen, Termine, Aktionen
WHERE ...
```

Um die Änderungen in der Datenbank einzutragen, müssen die neuen Werte von *TerminNr*, *Datum*, *Zeit* und *Beschreibung* in die Tabelle *Termine* eingetragen werden. *Vorname*, *Nachname* und *Postanschrift* gehören in die Tabelle Adressen; die restlichen Felder in die Tabelle Aktionen. Für jede Tabelle muss eine eigene SQL-Update-Anweisung ausgeführt werden.

Als Beispiel soll hier die Aktualisierung der Termintabelle dienen, die mit der folgenden SQL-Anweisung durchgeführt wird (im Datenmodul des Beispielprogramms ist sie im *CommandText*-Property der Komponente *qryModifyTagesTermine* zu finden):

```
update Termine
  set
    Datum = :Datum,
    Zeit = :Zeit,
    Beschreibung = :Beschreibung
  where
    TerminNr = :Old_TerminNr;
```

Die vier durch einen Doppelpunkt gekennzeichneten Parameter dieser Anweisung sind bei jeder Aktualisierung mit den aktuellen Feldwerten bzw. mit dem alten Feldwert der Terminnummer zu füllen (auch wenn die Terminnummer durch den Benutzer nicht editiert werden kann, ist es nahe liegend, die alte Nummer zu verwenden, denn es soll ja der alte Datensatz in der Datenbank gefunden werden).

Vor der Verwendung der Parameter sind außerdem für jeden Parameter die Parameterart (Eingabeparameter) und der Datentypen einzustellen. Dies geht am bequemsten zur Entwurfszeit über das Property *Params* der DataSet-Komponente (*Params[i].ParamType* wird auf *ptInput* gesetzt, *Params[i].DataType* auf den Datentyp des jeweiligen Feldes).

Hinweis: In der gezeigten *Update*-Anweisung wird der zu aktualisierende Datensatz über die Terminnummer identifiziert. Hierfür ist es wichtig, dass das Feld *TerminNr* überhaupt in der Client-Datenmenge enthalten ist (dafür muss es in der *Select*-Anweisung genannt werden). Im Beispielprogramm wird die Terminnummer nicht im DBGrid angezeigt, da sie für den Benutzer vermutlich keine Rolle spielt, jedoch wird sie in der *Select*-Anweisung genannt, ebenso wie natürlich auch die Aktionsnummer, die für das Update der Aktionstabelle benötigt wird.

SQL-Anweisungen ausführen

Um eine solche eigene SQL-Anweisung durchzuführen, bieten sich mit den dbExpress-Komponenten zwei Wege an:

- Mit der Methode *TSQLConnection.Execute* können Sie Ihrer Verbindungskomponente eine SQL-Anweisung als String übergeben, die dann direkt ausgeführt wird.
- Wenn Sie die SQL-Anweisung lieber als Komponente zur Entwurfszeit definieren wollen, können Sie eine *TSQLDataSet*-Komponente in das Datenmodul einfügen und die SQL-Anweisung in deren *CommandText*-Property eingeben. *CommandType* kann dann auf *ctQuery* gesetzt werden, jedoch wird die DataSet-Komponente nicht aktiv geschaltet oder mit *Open* geöffnet, sondern die SQL-Anweisung wird mit der Methode *ExecSQL* ausgeführt. Diese Möglichkeit wurde im Beispielprogramm gewählt.

Alte und neue Feldwerte

Nun ist die Frage, wie das Programm an die Feldwerte des zu aktualisierenden Datensatzes gelangen soll, denn diese müssen ja noch als Parameter in die SQL-Anweisung eingesetzt werden. Hier kommt nun ein weiterer Parameter des *BeforeUpdateRecord*-Ereignisses ins Spiel: die Datenmenge *DeltaDS*. Diese enthält alle Felder der geänderten Datenmenge, und Sie können mit den gewohnten Methoden *FieldByName* auf diese zugreifen. Die bisher noch nicht besprochenen *TField*-Properties *NewValue* und *OldValue* geben Ihnen nun Zugriff auf den alten bzw. neuen Wert des Feldes. *NewValue* ist nur bei den Feldern gesetzt, die gegenüber dem alten Wert geändert wurden.

Um beispielsweise den Parameter *:Beschreibung* aus der obigen SQL-Anweisung ermitteln zu können, kann der folgende Code ausgeführt werden:

```
Beschreibung := DeltaDS.FieldByName('Beschreibung').NewValue;
if Beschreibung = '' then
  Beschreibung := DeltaDS.FieldByName('Beschreibung').OldValue;
```

Der Wert *Beschreibung* kann dann in den gleichnamigen Parameter einer *TSQLDataSet*-Komponente eingesetzt werden – wie das in Kapitel 7.5.2 verwendete *TClientDataSet* verfügt auch diese über ein *Params*-Property, in dem die Parameter der Abfrage aufgezählt werden und über das die Werte zugewiesen werden können:

```
qryModifyTagesTermine.ParamByName('Beschreibung').Value := Beschreibung;
```

Dieser Vorgang wird in der Beispielanwendung auch für die anderen drei Parameter der oben gezeigten SQL-Anweisung wiederholt. Die Ermittlung der anderen Feldwerte mit *NewValue* und *OldValue* werden wir uns hier ersparen, der folgende Code-Auszug zeigt, wie die SQL-Anweisung letztlich zusammengesetzt und ausgeführt wird:

```
with qryModifyTagesTermineTermine do begin
  ParamByName('Beschreibung').Value := Beschreibung;
  ParamByName('OLDTerminNr').Value := ...
  ParamByName('Datum').Value := ...
  ParamByName('Zeit').Value := ...
end;
qryModifyTagesTermineTermine.ExecSQL(False);
```

Handarbeit für die Erzeugung der SQL-Anweisung

Leider funktionierte die oben gezeigte Verwendung des *Params*-Properties im Test nur mit BDE-Komponenten (wenn etwa statt *TSQLConnection* und *TSQLClientDataSet* eine Kombination aus *TDatabase* und *TBDEClientDataSet* verwendet wurde, siehe Projekt *BDETerminViewer* auf der CD-ROM). Bei Verwendung von dbExpress im Projekt *TerminVerwaltung* wurden die Parameter derart falsch in die SQL-Anweisung eingesetzt, dass die Syntax der Anweisung zerstört wurde und Interbase eine Fehlermeldung zurücklieferte. Das Beispielprogramm geht den Weg daher ganz zu Fuß und erzeugt

die SQL-Anweisung mitsamt Parametern selbst. Als Schablone dient nun eine Stringkonstante, in der die Parameter jeweils durch ein »%s« stellvertreten werden:

```
const
  sqlstrModifyTagesTermineTermine =
    'update Termine'+
    ' set'+
    '   Datum = "%s",'+
    '   Zeit = "%s",'+
    '   Beschreibung = "%s"'+
    ' where'+
    '   TerminNr = "%s"';
```

Zum Einsetzen von Werten in einen solchen String sieht die Laufzeitbibliothek von Delphi die *Format*-Funktion vor. Mit ihrer Hilfe erhält die *TSQLDataSet*-Komponente des Beispielprogramms eine SQL-Anweisung ohne syntaktische Mängel:

```
qryModifyTagesTermine.CommandText :=
    Format(sqlstrModifyTagesTermine, [t1, t2, t3, t4]);
```

Dabei sind die Variablen *t?* die Werte der Felder, die wie oben gezeigt vorher mit *NewValue* und *OldValue* ermittelt wurden. Die fertige SQL-Anweisung wird schließlich mit

```
qryModifyTagesTermine.ExecSQL(False);
```

ausgeführt, wobei der Parameter *False* besagt, dass die Komponente nicht selbst versuchen soll, Parameter in den *CommandText* einzuarbeiten.

Grenzen und Erweiterungsmöglichkeiten

Um das Beispielprogramm übersichtlich zu halten, werden übrigens in der Terminliste des Hauptformulars nur die Änderungen berücksichtigt, die sich auf die Tabellen *Aktionen* und *Termine* beziehen. Die aus der Adress-Tabelle stammenden Felder wurden auf *ReadOnly* geschaltet, damit hier Änderungen erst gar nicht möglich sind (dieses *ReadOnly*-Property ist über die persistenten Feldkomponenten der Datenmenge *cdsTagesTermine* im Datenmodul zu finden; rufen Sie für diese Komponente den Felder-Editor auf und wählen Sie eines der Felder).

Außerdem wurden Löschoperationen in der kombinierten Terminliste unterbunden, indem der entsprechende Schalter der *DBNavigator*-Komponente entfernt wurde (Property *VisibleButtons*), ebenso wie der Schalter zum Einfügen eines neuen Records. Für beide Operationen wären nämlich weitere SQL-Anweisungen für alle betroffenen Tabellen notwendig. Schließlich wurde in der zweiten Terminliste des Hauptformulars jegliches Editieren unmöglich gemacht, in dem die zugrunde liegende ClientDataSet-Komponente *cdsWhTermine* auf *ReadOnly* gesetzt wurde.

Updates für eine einzelne Tabelle

Mit dem in Kapitel 7.5.7 vorgestellten SQL-Monitor können Sie übrigens erforschen, welche Update-Anweisung die Komponenten von Delphi verwenden, wenn Sie *ApplyUpdates* aufrufen, ohne das Ereignis *BeforeUpdateRecord* zu bearbeiten. Wenn Sie den Code der oben gezeigten *BeforeUpdateRecord*-Methode ausklammern, das Programm starten und eine Änderung in die Hauptformular-Tabelle einzutragen versuchen, erhalten Sie erst einmal eine Exception, können dann aber im weiteren Programmverlauf im SQL-Protokoll beispielsweise den folgenden Text lesen:

```
Update Adressen
  set
    Beschreibung = ?
  where
    TerminNr = ? and
    Datum = ? and
    Zeit = ? and
    Vorname = ? and
    ...
```

Wenn wir annehmen, dass bei diesem Update immer nur eine der Tabellen aktualisiert werden soll (und nicht alle drei Tabellen wie im Beispielprogramm), so lassen sich Möglichkeiten finden, die automatisch erzeugte SQL-Anweisung so zu korrigieren, dass sie funktioniert. Dazu könnten wir hinter *Update* die Tabelle angeben, die aktualisiert werden soll. Nehmen wir einmal an, die Nennung der Tabelle *Adressen* im obigen Listing wäre schon korrekt, dann müsste als Weiteres beachtet werden, dass im *where*-Teil nur Felder angegeben werden, die auch in dieser Tabelle enthalten sind. Wir müssten also einfach alle »falschen« Felder aus der Anweisung entfernen.

Tatsächlich enthalten die Komponenten von Delphi einen Mechanismus, der genau dieser Vorgehensweise entspricht, er soll hier als Ausblick kurz skizziert werden (Details dazu gibt die Online-Hilfe):

- Über die Bearbeitung des Ereignisses *OnGetTableName* von *TClientDataSet* können Sie der Komponente mitteilen, welche Tabelle aktualisiert werden soll.
- Über das Property *ProviderFlags* der persistenten Feldkomponenten für die Felder der Tabelle können Sie einstellen, ob und wann dieses Feld in der Update-Anweisung berücksichtigt werden soll. Im obigen Beispiel würden wir beispielsweise die Flags *ProviderFlags.pfInUpdate* und *ProviderFlags.pfInWhere* auf *False* setzen, um ein Feld komplett aus der automatisch erzeugten Update-Anweisung herauszuhalten.

Updates mit der BDE

Wenn Sie statt dbExpress die BDE, also etwa *TQuery* statt *TSQLClientDataSet*-Komponenten, verwenden, können Sie die für die Updates notwendigen SQL-Anweisungen dialoggesteuert von *TUpdateSQL*-Komponenten erzeugen lassen. Für jede zu aktuali-

sierende Tabelle benötigen Sie eine eigene *TUpdateSQL*-Komponente. Mit jeder dieser Komponenten können Sie drei SQL-Anweisungen erzeugen – jeweils eine für die Modifikation, für das Einfügen und für das Löschen eines Datensatzes. Nachdem Sie eine *UpdateSQL*-Komponente über Property *TQuery.UpdateObject* mit der Datenmenge verbunden haben, rufen Sie den SQL-Anweisungseditor über einen Doppelklick auf die *UpdateSQL*-Komponente auf. Allerdings kann immer nur eine *UpdateSQL*-Komponente mit dem *UpdateObject*-Property verknüpft sein, daher müssen Sie, wenn Sie mehrere dieser Komponenten benötigen, das Property *UpdateObject* zur Entwurfszeit wieder löschen und die einzelnen *UpdateSQL*-Komponenten zur Laufzeit manuell aufrufen. Eine genauere Untersuchung dieser Update-Prozesse würde den Rahmen dieses Kapitels sprengen, ausführliche Erläuterungen dazu finden Sie in Delphis Online-Hilfe. Ein Beispiel für die Verwendung von *UpdateSQL*-Objekten zum Update der beiden Aktions- und Termintabelle (wie im oben ausführlicher besprochenen Beispiel) finden Sie auf der CD im Projekt *BDETerminViewer*.

7.5.7 Ein SQL-Monitor

Zum Abschluss soll noch ein Fenster in die Anwendung eingebaut werden, durch das man einen Blick hinter die Kulissen der Datenzugriffskomponenten werfen und die SQL-Anweisungen, mit denen die dbExpress-Komponenten im Hintergrund mit dem Datenbank-Server kommunizieren, beobachten kann.

In der dbExpress-Komponentenpalette finden Sie zu diesem Zweck eine Komponente namens *TSQLMonitor*, die gewissermaßen als Ersatz für den eigenständigen SQL-Monitor dient, den Sie beim Ausführen von BDE-Anwendungen mit DATENBANK | SQL-MONITOR aus der IDE heraus aufrufen können. *TSQLMonitor* ist eine nicht-visuelle Komponente, die über das Property *SQLConnection* mit der Verbindungskomponente verknüpft, deren Datenverkehr überwacht werden soll. Am sichersten ist es, die Monitor-Komponente in dasselbe Datenmodul zu setzen wie die Verbindungskomponente. Zwar können Sie die Komponente auch in einem Formular unterbringen, jedoch funktioniert die Verbindung zwischen *SQLMonitor* und *SQLConnection* nur dann, wenn das Formular der Monitor-Komponente *vor* dem Datenmodul erzeugt wird (dies bezieht sich auf die Reihenfolge der Formularerzeugung in den Projektoptionen).

In der Beispielanwendung wird das Datenmodul jedoch als Erstes initialisiert, da es die Grundlage für die gesamte Anwendung liefert. Die Monitor-Komponente wird daher im Datenmodul platziert, und ein eigenes Formular dient dazu, die Meldungen dieser Komponente anzuzeigen (siehe Abbildung 7.26).

Abbildung 7.26: Dieses Formular zeigt die TraceList der nicht-visuellen TSQLMonitor-Komponente an.

Ein SQL-Monitorfenster mit Zeitstempeln *R188*

Die Monitorkomponente muss zuerst aktiviert werden, indem das *Active*-Property gesetzt wird. Dies kann schon zur Entwurfszeit geschehen. Die Meldungen stehen im Folgenden über das Property *TSQLMonitor.TraceList* zum Abruf bereit. Über zwei Ereignisse – *OnTrace* und *OnLogTrace* – können Sie sich von neuen Meldungen informieren lassen und diese im Falle von *OnTrace* sogar bearbeiten, bevor sie Eingang in das Protokoll finden.

Das Monitor-Formular des Beispielprogramms enthält den folgenden Code:

```
procedure TFrmSQLMonitor.FormCreate(Sender: TObject);
begin
  Memo1.Lines := dmTermine.SQLMonitor1.TraceList;
  dmTermine.SQLMonitor1.OnTrace := SQLMonitor1Trace;
end;

procedure TFrmSQLMonitor.FormDestroy(Sender: TObject);
begin
  dmTermine.SQLMonitor1.OnTrace := nil;
end;

procedure TFrmSQLMonitor.SQLMonitor1Trace(Sender: TObject;
  CBInfo: pSQLTRACEDesc; var LogTrace: Boolean);
var
  TimeStamp: String;
begin
  TimeStamp := DateTimeToStr(Now)+'  ';
  if Strlen(CBInfo.pszTrace) + length(TimeStamp) + 1
       < sizeof(CBinfo.pszTrace) then
  begin
    Move(CBInfo.pszTrace[0], CBInfo.pszTrace[length(TimeStamp)],
                             strlen(CBInfo.pszTrace));
    Move(TimeStamp[1], CBInfo.pszTrace[0], length(TimeStamp));
  end;
  Memo1.Lines.Add(CBInfo.pszTrace);
end;
```

Zunächst wird beim *OnCreate*-Ereignis das *OnTrace*-Event des SQL-Monitors mit einer Formularmethode verknüpft, da dies im Objektinspektor nicht möglich ist, weil die

Komponente nicht in diesem Formular enthalten ist. Passend dazu wird die Ereignisverknüpfung in *OnDestroy* wieder gelöst, denn es könnte ja sein, dass das Formular einmal vor dem Datenmodul freigegeben wird. Ein Versuch des SQL-Monitors, die Formularmethode aufzurufen, würde dann im Daten-Nirvana enden. In der *OnCreate*-Methode werden außerdem die bisherigen Meldungen des SQL-Monitors im Memo-Feld angezeigt (das Datenmodul des SQL-Monitors wird vor dem Anzeige-Formular erzeugt, daher können schon vorher erste Meldungen protokolliert worden sein).

Die Bearbeitung der *OnTrace*-Methode fügt dem Meldungstext *CBInfo.pszText* noch einen Zeitstempel mit der aktuellen Zeit hinzu. Da *pszText* nicht als normaler String, sondern als festes Zeichen-Array mit 1024 Zeichen Länge definiert ist, findet dieses Hinzufügen oben statt, indem Speicherbereiche verschoben und kopiert werden (siehe auch Online-Hilfe zu *Move*). Die Länge eines nullterminierten Strings wie des erwähnten Zeichenarrays muss außerdem mit *strlen* (statt mit *length*) ermittelt werden.

Der erweiterte *pszText* wird dann vom Beispielformular an den Text des Memos angehängt und von *SQLMonitor1* automatisch in eine Datei geschrieben (sein *AutoSave*-Property wurde auf *True* gesetzt).

Hinweis: Sie können das SQL-Protokoll auch einfach in eine Datei schreiben lassen, indem Sie die Properties *FileName* und *AutoSave* von *TSQLMonitor* verwenden.

8 Die kooperative Delphi-Anwendung

Das Thema, das sich wie ein roter Faden durch dieses letzte große Kapitel zieht, ist der Austausch von Daten und Funktionalität zwischen einer Delphi-Anwendung und anderen Anwendungen:

- Als einfachste Datenaustausch-Plattform kommt in Abschnitt 8.1 zunächst die Zwischenablage zur Sprache, über die eine Delphi-Anwendung sehr leicht Text- und Bilddaten, mit etwas mehr Programmierung aber auch selbst definierte Datenformate bewegen kann.
- Abschnitt 8.2 befasst sich mit zwei weiteren Formen des Datenaustauschs, die Windows seinen Anwendungen anbietet: Die Protokolle DDE und OLE, für die Delphi in der SYSTEM-Komponentenpalette einige Komponenten bereitstellt.
- Abschnitt 8.3 widmet sich anschließend der grundlegendsten Form des Code-Austauschs unter Windows – den DLLs. Er zeigt die Erzeugung und Verwendung von DLLs und beschreibt, wie Sie Delphi-Formulare auf eine besonders zeitsparende Weise aus anderen Programmiersprachen heraus ansprechen und wie Sie Klassen zwischen Delphi und C++ austauschen können.
- Die Abschnitte 8.4 bis 8.7 stehen dann ganz im Zeichen des Component Object Models: In Abschnitt 8.4 geht es um die Verwendung, in Abschnitt 8.5 um die Entwicklung von COM-Objekten für Windows. Nachdem Abschnitt 8.6 sich der COM-Automation aus Sicht eines Clients gewidmet und dabei auch die Programmierung des Internet Explorers behandelt hat, zeigt Abschnitt 8.7 die Programmierung von COM-Automations-Servern und berücksichtigt dabei auch das Thema DCOM, bei dem zwei Anwendungen selbst dann zusammenarbeiten können, wenn sie auf verschiedenen Rechnern laufen.
- Abschnitt 8.8 schließlich befasst sich mit einer anderen, für den Internet-Benutzer alltäglichen Form von Zusammenarbeit zwischen zwei Anwendungen auf verschiedenen Rechnern: der Interaktion eines Web-Browsers mit einer Web-Server-Anwendung. Das Themenspektrum des Abschnitts reicht von der einfachen CGI-Anwendung, die schon mit der Personal-Ausgabe von Delphi erstellt werden kann, über WebBroker-Anwendungen bis zu den von Delphi 6 neu eingeführten WebSnap-Komponenten.

8.1 Die Zwischenablage

Die VCL definiert eine Klasse *TClipboard*, die den programmtechnischen Umgang mit der Zwischenablage stark vereinfacht. In vielen Fällen kommen Sie jedoch auch ohne diese Klasse aus, denn einige Delphi-Komponenten arbeiten schon von selbst mit der Zwischenablage zusammen. So verarbeiten Editierfelder die Tastaturkürzel für die Befehle *Einfügen*, *Ausschneiden* und *Kopieren* selbst dann, wenn Ihre Anwendung diese nicht in einem Menü zur Verfügung stellt.

Viele Komponenten stellen außerdem Methoden mit den Namen *PasteFromClipboard*, *CopyToClipboard* und *CutToClipboard* zur Verfügung, die Sie beispielsweise in den *OnClick*-Methoden von Menüpunkten aufrufen können. Es sind dies:

- ein- und mehrzeilige Editierfelder aller Art: *TEdit*, *TMemo*, *TRichEdit*, *TMaskEdit*, *TDBEdit* und *TDBMemo*. Die genannten Methoden haben dieselbe Funktion wie die eben genannten Tastenkürzel. Während *TRichEdit* natürlich auch Daten im RTF-Format versteht, akzeptieren die anderen Editierfeldtypen nur das einfache Textformat.
- *TDBImage*-Komponenten zur Anzeige von Bildern aus Datenbanken können über gleichlautende Methoden Bilder in der Zwischenablage ablegen und von dort lesen. Die von Datenbanken unabhängige *TImage*-Komponente besitzt solche Methoden nicht, jedoch geht es genauso einfach, wenn Sie die in Kapitel 8.1.2 erwähnten *Assign*-Methoden aufrufen (*Clipboard.Assign(Image.Picture)* zum Kopieren in die Zwischenablage und *TImage.Picture.Assign(Clipboard)* für die umgekehrte Richtung).

Zwei weitere Komponenten haben ebenfalls Kontakt zur Zwischenablage, jedoch nicht mit den üblichen *Copy/Cut/Paste*-Methoden:

- die Komponente *TDdeServerItem*, die die Informationen für eine DDE-Verbindung (die »Absenderadresse« des Servers) in die Zwischenablage kopiert (siehe Kapitel 8.2.1);
- die Komponente *TOleContainer*, die auf OLE-Objekte spezialisiert ist (siehe Kapitel 8.2.4).

8.1.1 Der Aufbau der Zwischenablage

Aus der Sicht des Programmierers ist die Zwischenablage ein Speicherplatz, der von Windows verwaltet wird und in dem jede Anwendung Daten in verschiedenen Formaten ablegen kann. Diese Daten sind öffentlich zugänglich und können von jeder anderen Anwendung gelesen (und auch überschrieben) werden. Meistens ist es auch sinnvoll, dass eine Anwendung ihre eigenen Daten wieder aus der Zwischenablage zurückliest, um so die üblichen Kopieroperationen zu erlauben.

Gesteuert werden diese Vorgänge vom Anwender, für den die Zwischenablage noch ein zweites Gesicht hat: Für ihn wird sie durch das Windows-Utility *Zwischenablage* repräsentiert, das jedoch tatsächlich nur ein Programm ist, das versucht, den Inhalt der ansonsten unsichtbaren Zwischenablage darzustellen.

Formate

Eine wichtige Eigenschaft der Zwischenablage ist, dass sie zur selben Zeit nur ein Datenobjekt aufnehmen kann, welches jedoch in mehreren Formaten vorliegen kann. Wenn Sie beispielsweise aus einer OLE-Server-Anwendung ein OLE-Objekt in die Zwischenablage kopieren, schreibt die Anwendung das Objekt außer als OLE-Objekt womöglich in verschiedenen anderen Formaten in die Zwischenablage (z.B. als einfache Bitmap). Andere Anwendungen, die vielleicht nur ein Format kennen, haben dann eine höhere Chance, dass sie die Daten in einem ihnen genehmen Format laden können.

Die von *TClipboard* unterstützten Zwischenablage-Formate sind:

Format	**Bedeutung**
CF_TEXT	nullterminierter String im normalen Windows-Zeichensatz (ANSI)
CF_BITMAP	normale, geräteabhängige Bitmap
CF_METAFILEPICT	Metadatei in speziellem Format
CF_PALETTE	Farbpalette, von der VCL nur in Verbindung mit einer Grafik (CF_BITMAP, CF_METAFILEPICT oder CF_PICTURE) beachtet
CF_COMPONENT	erstes von der VCL registriertes Format, für Komponenten
CF_PICTURE	zweites von der VCL registriertes Format, für den Inhalt eines *TPicture*-Objekts

Die nicht von der VCL registrierten Formate sind unter Windows standardmäßig vorhanden, wobei es von dieser Sorte noch weitaus mehr Formate gibt, beispielsweise *CF_WAVE* für Klangdaten, *CF_TIFF* für Bilddaten im TIFF-Format und *CF_HDROP* für »Dateisammlungen« , die aus dem Explorer kopiert wurden (siehe Beispielprogramm). Auch einfacher Text muss nicht immer (nur) im *CF_TEXT*-Format vorliegen, sondern kann auch das Format *CF_OEMTEXT* (verwendet statt dem ANSI- den OEM-Zeichensatz) oder *CF_UNICODETEXT* (nur unter Windows NT/2000, jedes Zeichen ein Wide-Char) haben.

8.1.2 TClipboard

Obwohl sie nur über wenige Properties und Methoden verfügt, erlaubt die Klasse *TClipboard* einen sehr effektiven Zugriff auf die Zwischenablage. *TClipboard* ist in der Unit *Clipbrd* enthalten und ist keine Komponente, sondern direkt von *TPersistent* abge-

leitet. Da Sie demnach keine Clipboard-Objekte zur Entwurfszeit in das Formular einfügen können, müssen Sie die Unit *Clipbrd* in jedem Fall selbst zur *uses*-Klausel Ihrer Unit hinzufügen.

Die wichtigsten Elemente von *TClipboard* sind:

- Zum Setzen des Zwischenablageinhalts das Property *AsText* und die Methoden *Assign* und *SetAsHandle* (und *SetAsComponent*, falls Sie eine Notwendigkeit sehen, Komponenten in die Zwischenablage zu kopieren).
- Zum Auslesen des Inhalts dienen die Methoden *GetAsHandle*, *GetAsComponent* und wieder das Property *AsText*; das Gegenstück zu *TClipboard.Assign* befindet sich außerhalb von *TClipboard* und ist die *Assign*-Methode von *TPicture*.
- Zur Bestimmung der verfügbaren Formate die Funktion *HasFormat* und das Array-Property *Formats* mit zugehörigem Zähler *FormatCount*.
- Zum manuellen Öffnen und Schließen der Zwischenablage die Methoden *Open* und *Close*.

Einfache Operationen

Clipbrd enthält nicht nur die Klasse, sondern auch ein globales Objekt davon, das wie zu erwarten *Clipboard* heißt und auf das Sie direkt zugreifen können. Um beispielsweise Text in die Zwischenablage zu kopieren, der sofort allen Anwendungen zugänglich ist, genügt eine einfache Zuweisung:

```
Clipboard.AsText := 'Hallo Zwischenablage!';
```

Ebenso können Sie *AsText* zum Auslesen des Textes verwenden.

Für die von *TClipboard* unterstützten Grafikformate gibt es keine Properties von der Art *AsPicture*, sondern die Methode *Assign*. Übergeben Sie dieser Methode ein *TBitmap*-, *TMetafile*-, *TGraphic*- oder *TPicture*-Objekt, um dessen Inhalt in die Zwischenablage zu kopieren, z.B.:

```
Clipboard.Assign(Graphic); { Kopieren der Grafik in die Zwischenablage }
```

Öffnen und Schließen

Bei den oben gezeigten Operationen öffnet und schließt das *Clipboard*-Objekt die Windows-Zwischenablage automatisch, bevor und nachdem es den eigentlichen Datentransfer durchführt. Wenn Sie mehrere verschiedene Objekte (in verschiedenen Formaten) in die Zwischenablage kopieren wollen, müssen Sie die Zwischenablage jedoch so lange geöffnet halten, bis alle gewünschten Objekte kopiert sind. Um das zu erreichen, schließen Sie die Kopieraktionen zwischen expliziten Aufrufen von *Open* und *Close* ein:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  with Clipboard do begin
    Open;
    Assign(Image1.Picture); { Grafikformat }
    AsText := 'Hallo Zwischenablage'; { Textformat }
    Close;
  end;
end;
```

Close schließt die Zwischenablage erst dann, wenn es genauso oft aufgerufen wurde wie *Open*. Das bedeutet, dass die Aufrufe von *Open* und *Close*, die innerhalb der einzelnen Kopieraktionen automatisch stattfinden, keine Wirkung haben.

Formatabfrage für Menüs

Wenn Sie die Zwischenablageoperationen in einem Menü angeben, kann es interessant sein, die derzeit in der Ablage befindlichen Formate zu überprüfen. So können Sie beim Öffnen des Menüs die Menüpunkte deaktivieren, die sich auf ein nicht vorhandenes Zwischenablageformat beziehen.

Hierzu genügt es nicht, die üblichen *OnClick*-Ereignisse zu bearbeiten, Sie benötigen ein Ereignis, das schon beim Öffnen des Menüs erzeugt wird. Beim Öffnen des Menüs kommt es entweder

- zu einem *OnClick*-Ereignis des (Haupt-)Menüpunkts, dessen Untermenü gerade geöffnet wird,
- oder zu einem *OnPopup*-Ereignis, falls es sich beim geöffneten Menü um eine einzelne *TPopupMenu*-Komponente handelt.

Bei diesem Ereignis können Sie das *Checked*-Property aller Menüpunkte neu setzen. Um das Format abzufragen, verwenden Sie im einfachsten Fall die *TClipboard*-Funktion *HasFormat*, der Sie das gefragte Format übergeben. Alternativ stellt *TClipboard* alle gerade verfügbaren Formate im Property *Formats* zur Verfügung.

```
{ Methode aus dem Mini-Demo ClipFrmt.dpr, dessen Menü die Punkte
  Text, Bitmap, Picture, Metadatei und Komponente enthält: }
procedure TForm1.ClipboardMenuClick(Sender: TObject);
begin
  with Clipboard do begin
    Text.Checked := HasFormat(CF_TEXT);
    Bitmap.Checked := HasFormat(CF_BITMAP);
    Picture.Checked := HasFormat(CF_PICTURE);
    Metadatei.Checked := HasFormat(CF_METAFILEPICT);
    Komponente.Checked := HasFormat(CF_COMPONENT);
 (* Dateien, deren Namen aus einem Explorer-Fenster
    in die Zwischenablage kopiert wurden (zur Feststellung
    dieser Namen siehe den Quelltext auf der CD-ROM): *)
```

```
      DroppedFiles.Checked := HasFormat(CF_HDROP);
    end;
  end;
```

Es ist sehr vorteilhaft, wenn Sie die Zwischenablageoperationen in einem Menü anbieten, denn dann können Sie mit einer Methode wie der obigen beim Aufklappen des Menüs immer die Menüpunkte deaktivieren, die für ein nicht zur Verfügung stehendes Zwischenablageformat stehen. Wenn die Zwischenablageoperationen auch über ständig sichtbare Schalter aufgerufen werden können und diese etwa deaktiviert werden sollen, falls ein Zwischenablageformat nicht zur Verfügung steht, muss der Zustand der Zwischenablage ständig überwacht werden. Hierzu bieten sich etwa die *OnUpdate*-Ereignisse von Aktionen an oder das *OnIdle*-Ereignis von *TApplication*, falls Sie die Schalter nicht mit Aktionslisten verknüpfen wollen.

Kopieren anwendungsspezifischer Daten *R68*

Besonders nützlich kann sich die Zwischenablage auch erweisen, wenn Sie Daten in einem anwendungsspezifischen Format zwischen verschiedenen Fenstern der Anwendung kopieren wollen (diese Fenster können etwa von zwei verschiedenen, gleichzeitig laufenden Instanzen der Anwendung stammen oder zwei Dokumentfenster innerhalb einer MDI-Anwendung sein).

Im TreeDesigner können Sie beispielsweise mit BEARBEITEN | KOPIEREN den markierten Teil des aktuellen Dokuments in die Zwischenablage kopieren (oder mit AUSSCHNEIDEN dahin verschieben) und mit BEARBEITEN | EINFÜGEN wieder einfügen – entweder in dasselbe Dokument, um ein Duplikat zu erhalten, oder in das Dokument eines anderen Dokumentfensters.

Für diese Funktionalität bietet es sich oft an, den Code zum Erzeugen einer Datei wiederzuverwenden, und zwar zum Erzeugen eines Memory-Streams. Dieser kann dann leicht über die Zwischenablage kopiert werden. Hierzu ist es als Erstes erforderlich, ein eigenes Zwischenablageformat zu registrieren, ein Beispiel aus der Unit DOCFORM des TreeDesigners:

```
var
  TDClipboardFormat: Integer;
initialization
  TDClipboardFormat := RegisterClipboardFormat('TreeDesigner_Data');
```

Für das Kopieren wird nun die bereits existierende Methode zum Schreiben der markierten Objekte in einen Stream (hier *TGraphicDocument.WriteToStream*) verwendet. Dann wird mit den beiden altertümlichen, aber von der Microsoft-Dokumentation empfohlenen API-Funktionen *GlobalAlloc* und *GlobalLock* ein Speicherbereich erzeugt, der anschließend mit dem Inhalt des Memory-Streams gefüllt wird. Das Handle dieses Speicherbereichs darf dann an *Clipboard.SetAsHandle* übergeben werden:

```
procedure TDocumentForm.Kopieren2Click(Sender: TObject);
var
  s: TMemoryStream;
  MemHandle: THandle;
  MemBlock: pointer;
begin
  s := TMemoryStream.Create;
  // Wenn MarkedElementCount = 0 werden alle Elemente kopiert.
  // TreeDesigner-Methode zum Schreiben der Dokumentdaten aufrufen:
  Document.WriteToStream(s, Document.MarkedElementCount>0);
  MemHandle := GlobalAlloc(GMEM_DDESHARE, s.Size); // Windows-API-Funk.
  MemBlock := GlobalLock(MemHandle); // Windows-API-Funktion
  s.Seek(0, soFromBeginning); // Jetzt den Stream wieder von vorne...
  s.Read(MemBlock^, s.Size); // ...lesen und in MemBlock schreiben.
  GlobalUnlock(MemHandle);  // Windows-API-Funktion
  Clipboard.SetAsHandle(TDClipboardFormat, MemHandle);
  s.Free;
end;
```

Wichtig für das Zurücklesen der Daten bei BEARBEITEN | EINFÜGEN ist vor allem, dass Sie die Zwischenablage so lange offen halten, bis die Daten aus dem von der Zwischenablage gehaltenen Speicherblock in das *TMemoryStream*-Objekt kopiert wurden. Hierfür sind zusätzliche Aufrufe von *Clipboard.Open* und *Clipboard.Close* erforderlich:

```
Clipboard.Open;
try
  MemHandle := Clipboard.GetAsHandle(TDClipboardFormat);
  if MemHandle <> 0 then begin
    Size := GlobalSize(MemHandle); // Windows-API
    MemBlock := Windows.GlobalLock(MemHandle); // Windows-API
    s := TMemoryStream.Create;
    // Den Speicher der Zwischenablage in den Memory-Stream einlesen:
    s.Write(MemBlock^, Size);
    Windows.GlobalUnlock(MemHandle); // Windows-API
    s.Seek(0, soFromBeginning); // Stream wird nun von vorne ausgelesen.
    Document.ReadFromStream(s);
    s.Free;
    Invalidate; // Fensterinhalt neu ausgeben.
  end;
finally
  Clipboard.Close;
end;
```

8.2 DDE und OLE

Während es Zwischenablagen auch unter anderen Betriebssystemen gibt und sie etwa mit Kylix unter Linux eine *TClipboard*-Klasse benutzen können, die der in Kapitel 8.1 beschriebenen sehr ähnlich ist, gibt es unter Windows noch zwei spezielle Datenaustausch-Protokolle:

- DDE ist zwar noch nicht so alt wie die Zwischenablage, trotzdem hat es seit Einführung des OLE und später weiterer ActiveX-Technologien an Bedeutung verloren, während die Zwischenablage immer noch so gut wie unersetzlich ist. Es gibt jedoch immer noch einen Bereich, in dem sich DDE besonders vorteilhaft einsetzen lässt: beim Austausch von einfachen Daten wie beispielsweise Textdaten über einen *heißen Draht*. Hierbei werden die Daten zunächst wie bei einer normalen Zwischenablageoperation von einer Anwendung in eine andere kopiert, bleiben danach jedoch miteinander verbunden, so dass sich die Kopie der Daten automatisch aktualisiert, wenn auch das Original verändert wird. Delphi-Beispielanwendungen hierfür finden Sie in den Kapiteln 8.2.1 bis 8.2.3.
- Die ursprüngliche und wörtliche Bedeutung des Begriffes »OLE« , mit der sich die Kapitel 8.2.4 bis 8.2.6 befassen, ist das »Verbinden und Einbetten von Objekten« (Object Linking and Embedding)[21]. Auch hier werden Daten aus einer Server- in eine Client-Anwendung eingefügt, jedoch handelt es sich bei diesen Daten nun gewissermaßen um intelligente Objekte bzw. Dokumente, die auch nach dem Einfügen in die Client-Anwendung noch von ihrer Ursprungsanwendung editiert werden können. Dieses Editieren kann sogar »in place« stattfinden – so können Sie etwa in das Beispielprogramm des Kapitels 8.2.6 Dokumente aus Office- und Grafikanwendungen verschiedener Hersteller einbinden, die dann innerhalb des Fensters der Delphi-Anwendung editiert werden können. Beim Editieren werden diverse Leisten der Server-Applikationen in das Delphi-Anwendungs-Fenster eingeblendet.

Die Komponenten der VCL erleichtern den Umgang mit diesen auf API-Ebene teilweise extrem komplizierten Protokollen um einige Größenordnungen. Dafür sind allerdings auch Einschränkungen in Kauf zu nehmen wie etwa, dass DDE-Daten von der Server-Anwendung nicht sehr flexibel bereitgestellt werden können (wie diese Einschränkung zu umgehen ist, wird in Kapitel 8.2.3 beschrieben), oder dass es keine VCL-Komponenten gibt, mit denen sich ein OLE-*Server* implementieren ließe (um diese Einschränkung zu umgehen, bräuchte es allerdings ein eigenes Buch ...).

8.2.1 DDE-Grundlagen und -Komponenten

Die DDE-Kommunikation läuft immer in einer Verbindung (Conversation / Unterhaltung) zwischen zwei Anwendungen ab, von denen die eine als Server und die andere als Client arbeitet. DDE ist hier sehr flexibel: Anwendungen können gleichzeitig mehrere »Gespräche« führen und dabei in einem Teil davon als Server und im anderen Teil als Client auftreten.

21 In einem Zwischenteil seiner Laufbahn wurde der Begriff »OLE« auch auf das Component Object Model (COM) ausgeweitet – so wurde etwa die heutige COM-Automation einmal als OLE-Automation bezeichnet, jedoch hat Microsoft mittlerweile mit »ActiveX« ein neues Schlagwort gefunden, das als Oberbegriff für OLE, COM und andere Technologien herhalten kann.

Die Verbindungsaufnahme beginnt grundsätzlich durch eine Anfrage des Clients. Dazu muss dieser den Namen kennen, unter dem der Server seine Dienste anbietet. Die DDE-Bezeichnung für »Dienst« ist *Service*. Der *Service* ist die erste von drei Angaben, die zur genauen Angabe einer Verbindung notwendig sind, die also quasi die Adresse der Information angeben, die der Client haben möchte. Als weitere Angaben sind erforderlich: ein allgemeines Thema (*topic*) sowie ein spezieller Gegenstand (*item*) der Unterhaltung.

Diese abstrakte Beschreibung macht deutlich, dass es ganz dem Server überlassen bleibt, in welche Themen er seine Dienste gliedert. Die Organisation der Daten auf einem PC legt jedoch ganz bestimmte Bedeutungen von Thema und Gegenstand nahe:

- Das Thema ist meistens der Name einer Datei, die im Server bearbeitet werden kann. Sie enthält die Daten, für die sich der Client interessiert, die er aber nicht direkt lesen kann, weil er das Dateiformat nicht kennt.
- Die eigentliche Information befindet sich im *Item*, der aus einem Teil der Datei besteht. Wie Sie den gewünschten Dateibereich angeben, hängt vom Server ab. Tabellenkalkulationsprogramme verstehen beispielsweise eine Angabe von *A1..B2* als rechteckigen Bereich, der die Zellen *A1* bis *B2* umfasst.

Heiße Drähte

Heiße Drähte zwischen zwei Dokumenten sind immer dann nützlich, wenn ein Dokument Teile eines anderen Dokuments enthält und es nicht mehr genügt, den Teil des Quelldokuments über die Zwischenablage zu kopieren, weil sich das Quelldokument ständig ändern kann und der Kopiervorgang über die Zwischenablage jedes Mal wiederholt werden müsste.

Wenn Sie einen heißen Draht zwischen den beiden Dokumenten errichten, wird die Client-Anwendung automatisch benachrichtigt, sobald sich der gemeinsame Datenbestandteil in der Server-Anwendung geändert hat.

DDE und OLE realisieren verschiedene Möglichkeiten für einen heißen Draht, erlauben aber auch eine »kalte« Verbindung, bei der der Client die Daten für jede Aktualisierung neu anfordern muss.

Herausfinden der DDE-Parameter

Wenn der Anwender eine DDE-Verbindung zwischen zwei Applikationen aufbauen will, sollte er nicht die komplizierte Adresse, bestehend aus Service, Topic und Item, angeben müssen, sondern eine einfache Möglichkeit haben, die Daten vom Server in den Client zu senden. Hier kommt wieder die Zwischenablage ins Spiel. DDE-Server

bieten oft die Möglichkeit, neben den Daten in den üblichen Text- und Grafikformaten auch eine »DDE-Absenderadresse« in einem speziellen Format in der Zwischenablage abzulegen.

Statt der Text- oder Grafikdaten kann der DDE-Client diese Adresse aus der Zwischenablage auslesen und über diese eine DDE-Verbindung zum Server aufbauen.

DDE mit Komponenten

Es folgt ein Überblick über Delphis DDE-Komponenten, die anschließend praktisch eingesetzt werden. Je zwei der vier DDE-Komponenten auf der Seite *System* der Komponentenpalette beschäftigen sich mit den Aufgaben eines DDE-Servers bzw. mit denen eines Clients. DDE bietet neben dem Aufbau eines heißen Drahts noch weitere Möglichkeiten, die ebenfalls in diesen Komponenten berücksichtigt werden: Der Client kann umgekehrt auch an den Server Daten senden, und er kann diesen über Makros steuern. Alle Komponenten sind nicht-visuell, sind also nur zur Entwurfszeit im Formular als Icon sichtbar. Die Komponenten sind im Einzelnen:

- *TDdeServerItem* enthält den Gegenstand der DDE-Verbindung aus der Sicht des Servers. Die Daten, die ausgetauscht werden sollen, liegen in Form eines *String*-Properties bzw. als Liste von Strings vor. Der Name der *TDdeServerItem*-Komponente wird auch gleichzeitig zum Namen des DDE-Items, den der Client in der Adressangabe verwenden muss, um genau diese Daten zu erhalten. Eine *TDdeServerItem*-Komponente kann (muss aber nicht) mit einer *TDdeServerConv*-Komponente verbunden sein.
- *TDdeServerConv* verwaltet die Verbindung zum Client und stellt Events zur Verfügung, in denen Sie Makros bearbeiten können, die der Client gesendet hat. Der Name der *TDdeServerConv*-Komponente wird automatisch zum Namen des DDE-Themas. Der Client muss dieses in der DDE-Adresse angeben, wenn er die DDE-Items (*TDdeServerItem*-Komponenten) ansprechen will, mit denen die *TDdeServerConv*-Komponente verbunden ist.
- Auf der Client-Seite ist *TDdeClientConv* für das Management der Kommunikation zuständig. Sie enthält Methoden, mit denen Sie Makros und andere Daten (über *PokeData*) an den Server senden können.
- Schließlich repräsentiert *TDdeClientItem* die (im Server befindlichen) DDE-Daten aus der Sicht des Clients. Auch hier werden die Daten als String bzw. als String-Liste dargestellt. *TDdeClientItem* muss mit einer *TDdeClientConv*-Komponente verbunden werden und wird automatisch aktualisiert (wenn der Server die automatische Aktualisierung, also den heißen Draht, unterstützt).

Sie können in der Client-Anwendung auch auf die *TDdeClientItem*-Komponente und in der Server-Anwendung auf die *TDdeServerConv*-Komponente verzichten:

- Im Falle des Clients müssen Sie dann die Daten mit *RequestData* jedes Mal explizit anfordern, wenn Sie eine Aktualisierung wünschen.
- Bei einem Server ohne *TDdeServerConv*-Komponente verwendet Delphi automatisch die Titelzeile des Formulars als DDE-Themenbezeichnung, was eine sicher nicht empfehlenswerte Vermischung von Benutzerschnittstelle und programminternen Datenbezeichnungen darstellt.

8.2.2 Ein heißer Draht zwischen Delphi-Anwendungen

Dieses Kapitel demonstriert anhand einer einfachen DDE-Verbindung zwischen zwei Delphi-Anwendungen, wie Sie Ihre Anwendungen mit den VCL-Komponenten zu DDE-Servern und -Clients machen können.

Der Beispiel-Server (Abbildung 8.1) enthält ein Memofeld, das der Benutzer editieren kann. Der Beispiel-Client enthält ebenfalls ein Memo, das den Inhalt des Server-Memos wiedergibt und immer sofort aktualisiert werden soll, wenn sich dessen Inhalt ändert (der zweite in der Abbildung gezeigte »Hot Link« wird in Kapitel 8.2.3 beschrieben).

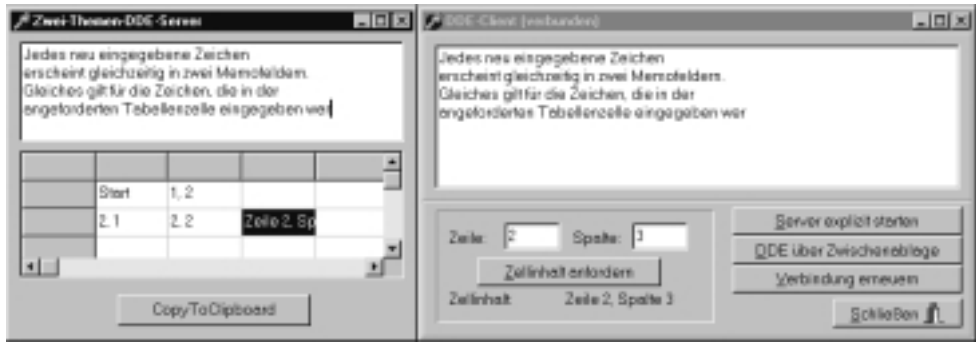


Abbildung 8.1: Die beiden »Hot Links« zwischen Server und Client beziehen sich auf den Text im Memo des Servers und auf den Inhalt der vom Client angeforderten Tabellenzelle.

Der Memo-Server *R66*

Ein funktionierender DDE-Server, der einen Memotext als DDE-Item zur Verfügung stellt (Projekt *SrvrApp*, Formulardatei *SrvrFrm*) ist in nur vier kurzen Schritten erstellt, wobei der erste Schritt, wie schon erwähnt, sogar noch optional ist:

- Geben Sie dem Formular eine *TDdeServerConv*-Komponente, die beiden Properties brauchen nicht verändert zu werden (der voreingestellte [Themen-]Name ist ausnahmsweise nicht *DdeServerConv1*, sondern *DDEServer1*, ein für Clients gerade noch zumutbarer Themen-Name).
- Fügen Sie eine *TDdeServerItem*-Komponente hinzu, die Sie mit der ersten Komponente verbinden, indem Sie das Property *ServerConv* auf diese setzen. Im Beispielprogramm wurde der Name des *DdeServerItems* in *StaticServerItem* geändert, damit dieser vom zweiten, im nächsten Kapitel hinzukommenden Item erkennbar verschieden ist.
- Nehmen Sie eine *Memo*-Komponente in das Formular auf. Im Beispielformular befindet sich diese auf einer Seite einer *Notebook*-Komponente.
- Schließlich erstellen Sie eine Ereignisbearbeitungsmethode für das *Memo*-Event *OnChange*, in der Sie das *Lines*-Property des Memos an das *Lines*-Property des DDE-Items weitergeben:

```
procedure TForm1.MemoChange(Sender: TObject);
begin
  ServerItem.Lines := Memo.Lines;
end;
```

Jedes Mal, wenn der Anwender den Text des Memo-Felds verändert, kopiert diese Methode die gesamten Textdaten in den *DdeServerItem*. Falls ein DDE-Client die Verbindung zu diesem Server, dem Topic *DDEServer1* und dem Item *StaticServerItem* aufgenommen hat, sorgt die VCL dafür, dass der Client sofort von dieser Veränderung benachrichtigt wird. Falls der Client eine andere Delphi-Anwendung ist, befinden sich die Daten dann im Property *TDdeClientItem.Lines*.

Für den völlig reibungslosen Ablauf müssen Sie die Zeile aus der obigen *OnChange*-Methode auch in die *OnCreate*-Methode des Server-Fensters kopieren, damit der Inhalt des Memos nach dem Programmstart auch dann sofort im *ServerItem* zur Verfügung steht, wenn Sie ihn schon zur Entwurfszeit initialisiert haben. Ansonsten müssten Sie die vorgegebenen Daten erst verändern, damit sie erstmals an den Client gesendet werden.

Die Zwischenablage

Damit der Beispiel-Server die DDE-Adresse (Service / Topic / Item) auch in der Zwischenablage ablegen kann, erhält er zusätzlich einen Schalter, der dem üblichen *Kopieren*-Menüpunkt entspricht. Allerdings werden hier keinerlei Text- oder Grafikdaten in die Zwischenablage kopiert, sondern nur die DDE-Verbindungsdaten:

```
procedure TForm1.BtnCopyToClipboardClick(Sender: TObject);
begin
  ServerItem.CopyToClipboard;
end;
```

In vielen DDE-Clients können Sie diese Verbindungs-Information auslesen und die Verbindung aufbauen, indem Sie den Menüpunkt BEARBEITEN | EINFÜGEN auswählen, und auch in Delphis DDE-Clients können Sie diese Funktion leicht bereitstellen.

Der Client

Der Beispiel-Client (Projekt *CliApp*) benötigt als Erstes eine *TDdeClientConv*-Komponente, deren Parameter wie folgt geändert werden:

- *DdeService* und *DdeTopic* geben die ersten beiden der drei Verbindungsparameter an. Wenn die Server-Anwendung bereits aktiv ist, können Sie die DDE-Verbindungsdaten von dort in die Zwischenablage kopieren. Drücken Sie dazu in *SrvrApp* den Schalter *CopyToClipboard*. Daraufhin können Sie den Property-Editor zu einem der beiden Properties aufrufen, um den *DDE Info*-Dialog zu erhalten. Drücken Sie dort den Schalter *Verbindung einfügen*, und Delphi füllt beide Properties mit der DDE-Information aus der Zwischenablage aus.
- *ServiceApp* enthält den Namen der ausführbaren EXE-Datei des Servers (ohne die Dateiendung). Wenn Sie diesen hier angeben, versucht die VCL automatisch, den Server zu starten, wenn die Verbindungsaufnahme einmal erfolglos verläuft.
- *FormatChars* wird auf *True* gesetzt, damit die VCL die Zeilenende-Markierungen nicht aus den DDE-Daten herauslöscht und die Zeilentrennung auch im Client noch sichtbar bleibt. Sollte Ihr Server einmal unbrauchbare Steuerzeichen in den DDE-Daten mitsenden, lassen Sie dieses Property bei seinem voreingestellten Wert *False*.

Als zweite Komponente benötigt der Client eine *TDdeClientItem*-Komponente, zu der wir nach einigen Bemerkungen zum Verbindungsmodus kommen.

Der Verbindungsmodus

TDdeClientConv lässt Ihnen durch sein Property *ConnectMode* die Wahl zwischen einem automatischen und einem manuellen Verbindungsmodus. Im Modus *ddeAutomatic* wird die Verbindung zum Server nach dem Öffnen des Formulars sofort erzeugt, im Modus *ddeManual* müssen Sie zuerst die Methode *OpenLink* aufrufen. Das Beispielprogramm verwendet den manuellen Verbindungsmodus, dem die Methode für das *OnCreate*-Ereignis wie folgt Rechnung trägt:

```
procedure TClientForm.FormCreate(Sender: TObject);
begin
  DdeClientConv1.OpenLink;
end;
```

Der voreingestellte automatische Modus bewirkt, dass die DDE-Verbindung schon zur Entwurfszeit beim Laden des Formulars aufgebaut wird. Falls die Verbindungsauf-

nahme zu diesem Zeitpunkt scheitert, löscht die VCL die DDE-Verbindungsparameter, die über die Dialogbox eingefügt wurden, wieder, wodurch die Verbindung auch zur Laufzeit nicht mehr aufgebaut wird – es sei denn, Sie fügen die DDE-Parameter wieder neu ein (hierfür müssen Sie dann zur Entwurfszeit extra den Server starten). Diese Umstände sind beim manuellen Verbindungsmodus nicht erforderlich.

Die ClientItem-Komponente des Clients

Wir kommen nun zur *TDdeClientItem*-Komponente des Beispiel-Clients, in der zwei Properties eingestellt werden:

- Das Property *DdeConv* muss auf die *TDdeClientConv*-Komponente weisen, die den Client zum gewünschten Service und Topic verbindet (wenn Sie mehrere solcher Komponenten haben, können Sie so zwischen mehreren Themen und / oder Servern wählen).
- *DdeItem* nimmt nun den noch fehlenden Verbindungsparameter auf. Falls sich die eben aus der Zwischenablage kopierten DDE-Daten immer noch in der Zwischenablage aufhalten, können Sie den zugehörigen Item-Namen aus der aufklappbaren Liste auswählen. Falls nicht, erlaubt Ihnen Delphi (bzw. die VCL) zur Entwurfszeit leider nicht, einen eigenen Namen einzugeben, sondern Sie müssen den Server erneut beauftragen, seine DDE-Daten in die Zwischenablage zu kopieren.

Die Methoden des Clients

Ein DDE-Client erfordert einen höheren Programmieraufwand als der Server. Neben der *FormCreate*-Methode, die bereits gezeigt wurde, enthält das Beispielprogramm drei Methoden, die die drei in Abbildung 8.1 gezeigten Schalter bearbeiten, und eine, die den Anwender über den Abbruch der Verbindung informiert.

Am wichtigsten ist jedoch die Methode für das Event *OnChange* des *DdeClientItems*. Dieses tritt immer dann auf, wenn der Server die Daten aktualisiert (wenn sich in der anderen Beispiel-Anwendung also *TDdeServerItem.Lines* verändert). Die Methode des Beispielprogramms verhält sich gerade umgekehrt zu der korrespondierenden *OnChange*-Methode des Server-Memos:

```
procedure TClientForm.StaticClientItemChange(Sender: TObject);
begin
  Memo.Lines := StaticClientItem.Lines;
end;
```

Mit dem Schalter DDE ÜBER ZWISCHENABLAGE rufen Sie die folgende Methode auf, die die DDE-Informationen aus der Zwischenablage holt und eine neue DDE-Verbindung aufbaut:

```
procedure TClientForm.PasteFromCbButtonClick(Sender: TObject);
var
  Service, Topic, Item: string;
begin
  if GetPasteLinkInfo(Service, Topic, Item) then begin
    { unbedingt in dieser Reihenfolge auszuführen: }
    DdeClientConv1.SetLink(Service, Topic);
    StaticClientItem.DdeItem := Item;
    { nur im manuellen Modus notwendig: }
    DdeClientConv1.OpenLink;
  end;
end;
```

GetPasteLinkInfo ist eine Hilfsroutine der Unit *DdeMan*, die auch die DDE-Komponenten enthält. Die Routine speichert die drei Teile der DDE-Adresse in den Variablenparametern, so dass diese in einem zweiten Schritt zur Aufnahme einer Verbindung verwendet werden können. Dabei entspricht *SetLink* dem gleichzeitigen Setzen der Properties *DdeService* und *DdeTopic* in der *DdeClientConv*-Komponente, das Property *TDdeClientItem.DdeItem* muss wie im Objektinspektor einzeln gesetzt werden.

Die Methode für den Schalter SERVER EXPLIZIT STARTEN besteht lediglich aus einem Aufruf der Windows-API-Funktion *WinExec*:

```
procedure TClientForm.StartServerButtonClick(Sender: TObject);
begin
  WinExec('SrverApp.exe', SW_SHOWNORMAL);
end;
```

Schließlich soll die DDE-Verbindung auch dann wiederhergestellt werden können, wenn der Server mitten in der Verbindung geschlossen und erneut geöffnet wird. Daher gibt es im Beispielformular den Schalter VERBINDUNG ERNEUERN, der wie beim Programmstart in *FormCreate* die Methode *OpenLink* aufruft:

```
procedure TClientForm.VerbindungErneuernClick(Sender: TObject);
begin
  DdeClientConv1.OpenLink;
end;
```

Falls der Anwender gewarnt werden soll, wenn der Server die Verbindung abbricht, können Sie das *OnClose*-Event der *TDdeClientConv*-Komponente bearbeiten. Das Beispielprogramm begnügt sich damit, bei jedem Verbindungsabbruch eine Meldung auszugeben:

```
procedure TClientForm.DdeClientConv1Close(Sender: TObject);
begin
  MessageDlg('Verbindung wurde beendet.', mtInformation, [mbOK], 0);
end;
```

TDdeClientConv bietet übrigens auch ein entsprechendes Ereignis *OnOpen* an, das beim erfolgreichen Aufbau der Verbindung eintritt.

8.2.3 Dynamische Items und Makros

Die große Aufgabe besteht nun darin, dass der Server dem Client erlaubt, einzelne Zellen des *StringGrids* abzufragen, wie in Abbildung 8.1 angedeutet. Für den Client ist das unter Delphi kein Problem: Sie müssen nur das Property *DDEItem* der *TDdeClientItem*-Komponente ändern, um einen anderen Item und damit eine andere Zelle abzufragen. Allerdings gibt es beim Server VCL-bedingte Schwierigkeiten.

Neue Items auf Anfrage *R160*

Ein großer Nachteil von Delphis DDE-Server-Komponenten ist, dass sie nicht dynamisch auf Adressänderungen im Item-Teil der Adresse reagieren können, wie es beispielsweise Tabellenkalkulationen ständig tun müssen. Einer solchen können Sie üblicherweise die verschiedensten Items abverlangen: *$A1$A1*, *$A2$A2*, *$A1$A2* usw. Wenn der Client einen solchen Bereich verlangt, konstruiert die Server-Anwendung erst nach dieser Anfrage ein neues DDE-Objekt. In einer Delphi-Anwendung könnte die Komponente *TDdeServerConv* z.B. ein Ereignis *OnRequestData* einführen, in dem Sie dann auf die neue Anfrage reagieren könnten.

Falls Sie über den Quelltext der VCL verfügen, können Sie in der Unit *DdeMan* auch eine ähnliche Methode *RequestData* finden, allerdings nur in der privaten Klasse *TDdeSrvrConv* im Implementationsteil der Unit. Die derzeitige Version der VCL kapselt damit so viele DDE-Kommunikationsabläufe vollkommen vom Komponenten-Anwender ab, dass der DDE-Item immer schon vorhanden sein muss, wenn er vom Client abgefragt wird.

Anfrage durch Makros

Die beiden Beispielprogramme müssen ihren Wunsch, beliebige Zellinhalte austauschen zu können, also auf eine andere Weise durchsetzen, und zwar tun sie das auf die folgende Art.

Bevor die Client-Anwendung ihren *DdeClientItem* auf eine der Zellen programmiert (wie oben erwähnt durch Ändern des *DdeItem*-Properties), muss sie den Server durch einen Makro-Aufruf vorwarnen. Darin teilt sie dem Server mit, dass sie gleich eine DDE-Anfrage nach einer bestimmten Zelle starten wird. Der Server kann nun als Reaktion auf dieses Makro schnell eine *TDdeServerItem*-Komponente mit dem passenden Namen erstellen, so dass die folgende DDE-Anfrage nach der Zelle erfolgreich verläuft.

Makros aufrufen R161

Die Komponente *TDdeClientConv* gibt Ihnen zwei Möglichkeiten, Makros aufzurufen: Mit *ExecuteMacro* können Sie einen einfachen String als Makro ausführen, mit *ExecuteMacroLines* eine ganze String-Liste.

Das Makro für den Beispielserver soll aus Zeilen bestehen, da die drei Bestandteile des Makros so leichter zu trennen sind (über das *Lines*-Property). Um die Zelle (5,6) anzufordern, sendet der Client die folgenden Zeilen:

```
CreateDDEItem
5
6
```

Der folgende Ausschnitt zeigt den Teil der Methode für den Schalter ZELLINHALT ANFORDERN, der für die Absendung des Makros zuständig ist. Sie baut mit Hilfe der Eingaben in die Eingabefelder *Zeile* und *Spalte* eine String-Liste nach dem beschriebenen Muster auf und sendet diese mit *ExecuteMacroLines* an den Server:

```
procedure TClientForm.GetCellButtonClick(Sender: TObject);
var
  Macro: TStrings;
begin
  Macro := TStringList.Create;
  try
    Macro.Add('CreateDDEItem');
    Macro.Add(NumEdit1.Text);
    Macro.Add(NumEdit2.Text);
    DDEClientConv1.ExecuteMacroLines(Macro, False);
    ... Verbindungsaufnahme (siehe nächster Abschnitt) ...
  finally
    Macro.Free; { Stringliste auch bei einer Exception freigeben }
  end;
end;
```

Dynamische Items abfragen

Um die Verbindung aufzubauen, drücken Sie zur Laufzeit in der Client-Anwendung den Schalter ZELLINHALT ANFORDERN, dessen oben gezeigte *OnClick*-Methode nur noch um zwei Zeilen ergänzt werden muss. Zunächst erhält der Server während des Aufrufs von *Application.ProcessMessages* Gelegenheit, das Makro zu verarbeiten, dann wird der Name des DDE-Items nach dem Muster »DX_Y« erzeugt, wobei *X* und *Y* die Zeilen- und Spaltenangaben sind:

```
{ Einzufügen in die obige Methode: }
    Application.ProcessMessages;
    DynamicClientItem.DDEItem := 'D'+NumEdit1.Text+'_'+NumEdit2.Text;
```

Die *DdeClientItem*-Komponente heißt zwar *DynamicClientItem*, wurde aber schon zur Entwurfszeit in das Formular eingesetzt. Sie ist nur insoweit dynamischer als *StaticClientItem* aus dem letzten Kapitel, als sie ihren Namen zur Laufzeit gezielt wechselt.

Wenn der Item-Name in der obigen Methode nach *ProcessMessages* gesetzt wird, sendet der Server die Daten der Zelle sofort an den Client zurück, was zu einem *OnChange*-Ereignis bei *DynamicClientItem* führt. Dieses ähnelt den bisherigen *OnChange*-Ereignissen und schreibt die Zelldaten in das dafür vorgesehene *TLabel*-Feld:

```
procedure TClientForm.DynamicClientItemChange(Sender: TObject);
begin
  CellData.Caption := DynamicClientItem.Text;
end;
```

Makros bearbeiten und dynamische Items erzeugen

Für jeden Makro-Aufruf erzeugt die *TDdeServerConv*-Komponente ein *OnExecuteMakro*-Ereignis und gibt diesem den Makrotext in Form eines Stringlisten-Parameters mit auf den Weg. Im Beispiel enthält diese genau die drei Strings, die die Client-Anwendung in ihre eigene String-Liste geschrieben hat, bevor sie diese an *ExecuteMacroLines* übergeben hat.

Nach einem kurzen Test, ob die Positionsangabe aus Zahlen im richtigen Bereich besteht, schreitet die *OnExecuteMacro*-Methode des Beispielprogramms zur Tat und erzeugt dynamisch eine *TDdeServerItem*-Komponente. Sie setzt dabei die Properties *ServerConv* und *Name*, die sonst auch zur Entwurfszeit gesetzt werden.

Am Schluss setzt die Methode den *Text* des DDE-Items auf den Zellinhalt der *TGrid*-Komponente. Falls das Makro nach einer ungültigen Zelle verlangt hat, setzt sie den DDE-Text auf »Fehler«, Windows und die VCL leiten ihn schließlich zur *OnChange*-Methode des Clients weiter.

```
procedure TForm1.DDEServer1ExecuteMacro(Sender: TObject; Msg: TStrings);
var
  Error: Integer;
begin
 (* Umwandeln des Makros in Zahlendaten *)
  Val(Msg[1], Row, Error);  if Error <> 0 then Row := -1;
  Val(Msg[2], Col, Error);  if Error <> 0 then Row := -1;
 (* Überprüfen des Zahlenbereichs *)
  if (Col < 1) or (Col >= StringGrid.ColCount)
     or (Row < 1) or (Row >= StringGrid.RowCount)
     then Row := -1;
 (* Überprüfen der ersten Makrozeile *)
  if (CompareText(Msg[0], 'CreateDDEItem') = 0) then begin
    if Assigned(DynamicItem) then
      DynamicItem.Free;
    DynamicItem := TDdeServerItem.Create(self);
```

```
      DynamicItem.Name := 'D'+Msg[1]+'_'+Msg[2];
      DynamicItem.ServerConv := DDEServer1;
      if Row = -1 then
        DynamicItem.Text := 'Fehler'
      else
        DynamicItem.Text := StringGrid.Cells[Col, Row];
    end;
  end;
```

Letzte Aufgabe des Servers ist nun, die mit viel Mühe aufgebaute dynamische Verbindung auch jedes Mal zu aktualisieren, wenn die entsprechende Zelle in der Stringtabelle verändert wird. Dazu lauert die folgende Methode auf Änderungen in der gewählten Zelle und sendet diese umgehend an den Client, der sie in der schon gezeigten *OnChange*-Methode anzeigt.

```
{ Bearbeitung für das Ereignis OnSetEditText der TGrid-Komponente }
procedure TForm1.StringGridSetEditText(Sender: TObject; ACol,
  ARow: LongInt; const Value: string);
begin
  if Assigned(DynamicItem) then
    if (ACol = Col) and (ARow = Row) then
      DynamicItem.Text := Value;
end;
```

Hinweis: DDE-Makros können auch dazu verwendet werden, Programmgruppen im Programm-Manager bzw. im START-Menü von Windows anzulegen, z.B. durch den Aufruf von *DdeClientConv1.ExecuteMacro('[CreateGroup(»Neue Gruppe«)]', False);* Die zahlreichen Möglichkeiten dieser DDE-Schnittstelle finden Sie im Inhaltsverzeichnis der Win32-Hilfe unter *Shell Dynamic Data Exchange Interface*. Eine modernere Variante zum Anlegen von Programmgruppen ist in Kapitel 8.4.2 erläutert.

8.2.4 OLE-Grundlagen

Die Bezeichnung *Object Linking and Embedding* weist bereits auf die beiden verschiedenen Typen eines OLE-Objekts hin, die verknüpften (*Linking*) und die eingebundenen Objekte (*Embedding*). Am wichtigsten ist jedoch zuerst einmal das *Objekt* selbst – es ist mit dem *Item* einer DDE-Verbindung vergleichbar und kann sowohl ein ganzes Dokument als auch ein Ausschnitt daraus sein. Ein Rückblick auf die Zeit vor OLE zeigt, dass Einbetten und Verknüpfen von Objekten in gewisser Weise schon immer möglich war.

Einbinden und Verknüpfen ohne OLE

Vor der Einführung von OLE boten die meisten Anwendungen als einzige Möglichkeit des Datenaustauschs mit anderen Anwendungen den Import oder den Export von Dateien oder das Kopieren über die Zwischenablage an. Um eine Grafik in einen Text einzufügen, konnte ein Textverarbeitungsprogramm entweder die gesamte Grafik in das eigene Dokument hineinkopieren oder im Dokument eine Referenz auf eine externe Datei speichern. In beiden Fällen musste das Textverarbeitungsprogramm das Format der Grafikdatei kennen, um die Grafik lesen und inmitten des Textes darstellen zu können. Wenn die Textverarbeitung die zweite Möglichkeit verwendet und die Grafik als Referenz auf eine externe Datei eingebunden hat, konnte der Anwender diese nach Belieben ändern und speichern, und spätestens beim nächsten Öffnen des Textdokuments würde das Textprogramm die neue Version der Grafik anzeigen.

Das Wesen eines OLE-Objekts

Auch mit OLE sind diese beiden verschiedenen Operationen möglich – Sie können ein OLE-Objekt in die Datei einer Anwendung aufnehmen oder statt dessen eine Referenz auf eine externe Datei verwenden, wobei Sie diese externe Datei unabhängig vom Client verändern können (auch bei OLE wird die Anwendung, aus der das Objekt stammt, als *Server*, die Anwendung, in der es verwendet wird, als *Client* bezeichnet).

Während die Daten, die in ein anderes Dokument eingebunden werden, bei der herkömmlichen Methode vollkommen passiv sind und mal von der einen, mal von der anderen Anwendung interpretiert werden, können Sie sich ein OLE-Objekt als funktionierendes Objekt vorstellen, dass neben den passiven Daten noch zwei sehr wichtige Fähigkeiten mitbringt:

- Es kann sich selbst in verschiedenen Größen in ein beliebiges Fenster zeichnen oder auf Druckern ausgeben.
- Und es ist uneingeschränkt editierbar, so wie in der Anwendung, unter der es erstellt wurde.

Beide Fähigkeiten erhält das OLE-Objekt von der Server-Anwendung. Für den Client gibt es keinen Grund mehr, das Datenformat des eingebundenen Objekts zu kennen, für OLE-Objekte sind also keine Import- und Export-Filter notwendig. Daraus folgt wiederum, dass ein OLE-Client eine unbegrenzte Anzahl verschiedener Formate lesen kann, solange diese von einem OLE-Server bereitgestellt werden.

Ein OLE-Objekt übt seine »Fähigkeit, editierbar zu sein« aus, indem es die Server-Anwendung aufruft. Sind Client und Server Anwendungen, die die Version 2.0 der OLE-Spezifikation unterstützen, so kann der Server das Fenster des Clients zum Editieren von eingebetteten Objekten benutzen und sogar seine eigenen Menüs und Mauspaletten darin einbinden.

Einbinden und Verknüpfen

Wenn eine der beiden Anwendungen nur OLE 1.0 kennt, so findet das Editieren getrennt vom Client im Original-Fenster des Servers statt. Objekte, die nur verknüpft sind, werden logischerweise auch bei OLE 2.0 getrennt von der Client-Anwendung editiert: Es kann ja mehrere Clients geben, die dieses Objekt enthalten, so dass alle das gleiche Anrecht auf Änderungen des OLE-Objekts haben. Das getrennte Editieren unterstreicht in diesem Fall also den unabhängigen Status des verknüpften OLE-Objekts.

Ansonsten gelten für beide Methoden die gleichen Vor- und Nachteile wie vor der Zeit des OLE. Für eingebettete Objekte sprechen die folgenden Punkte:

- Eingebetteten Objekten macht es keine Probleme, wenn die ursprüngliche Datei verschoben oder gelöscht wird, da die Client-Anwendung mit einer Kopie arbeitet. Bei einem verknüpften Objekt geht das OLE-Protokoll davon aus, dass die Datei nicht verschwindet oder sich in einem anderen Verzeichnis versteckt.
- Änderungen in eingebetteten Objekten schlagen sich weder in der Originaldatei noch in anderen Clients, die dasselbe Objekt benutzen, nieder. Dies ist natürlich nur dann ein Vorteil, wenn Sie genau das bezwecken.

Für verknüpfte Objekte spricht,

- dass sie die Größe des Client-Dokuments nicht so stark ansteigen lassen wie eingebettete Objekte, die ihre gesamten Daten dort ablagern.
- dass Sie dadurch auch insgesamt Speicherplatz sparen, wenn ein verknüpftes Objekt in mehreren Clients verwendet wird.
- dass sich alle Änderungen am Objekt gleichzeitig in allen Clients widerspiegeln.

OLE in der VCL

Die VCL enthält keine Komponente, die genau ein OLE-Objekt kapselt. Statt dessen ist die *TOleContainer*-Komponente soviel wie *ein Fenster, das ein OLE-Objekt anzeigt* (zur Frage, wo sich das eigentliche OLE-Objekt befindet, sei auf das Ende von Kapitel 8.2.6 verwiesen).

Die Komponenten-Unterstützung der VCL ist auf OLE-Client-Anwendungen beschränkt. OLE-Server sind zu komplex, als dass sie sich mit ein paar visuellen Komponenten zusammenbauen ließen. Um eine Server-Anwendung zu schreiben, müssten Sie die Schnittstellen des Component Object Models verwenden und eigene COM-Klassen schreiben.

OLE-Objekte zur Entwurfszeit

TOleContainer verfügt über verschiedene Möglichkeiten, ein OLE-Objekt schon zur Entwurfszeit in das Fenster einzubinden. Ein zur Entwurfszeit eingefügtes OLE-Objekt wird in der Formulardatei (beim Kompilieren der Anwendung auch in der EXE-Datei) gespeichert und beim Start der Anwendung automatisch geladen. Zur Laufzeit können Sie es dann wie gewohnt editieren.

Allerdings ist das Speichern des veränderten Objekts etwas problematisch: Da *TOleContainer* nicht die gesamte EXE-Datei umorganisiert, nur um das veränderte OLE-Objekt wieder dort zu speichern, müssen Sie solche OLE-Objekte in einer eigenen Datei speichern. Da das ursprüngliche Objekt weiter in der EXE-Datei bliebe, müsste es beim nächsten Programmstart extra durch das neue Objekt ersetzt werden, abgesehen davon, dass die veraltete Version des Objekts in der EXE-Datei womöglich völlig unnötig Speicher verbraucht.

Es gibt also viele Nachteile für das Einbinden von OLE-Objekten zur Entwurfszeit – abgesehen davon, dass dies dem eigentlichen Sinn von OLE widerspricht: OLE sollte zu einem flexibleren Umgang mit Daten unterschiedlicher Herkunft führen und nicht zu einer statischen Verbindung, wie sie bei einem fest vorgegebenen OLE-Objekt zustande käme.

Die grundlegenden Dialoge

Die Entwurfszeit-Fähigkeiten von *TOleContainer* eignen sich jedoch gut dazu, die beiden wichtigsten Dialoge für OLE-Clients auszuprobieren, denn Sie können beide Dialoge über das Popup-Menü der *TOleContainer*-Komponente aufrufen.

Mit dem Dialog *Objekt einfügen* (Abbildung 8.2) können Sie

- ein neues OLE-Objekt erstellen, das automatisch eingebettet wird. Sie wählen dazu aus der Liste des Dialogs einen OLE-Objekttyp aus, von dem ein neues Objekt erstellt werden soll. Dieser Vorgang ist vergleichbar mit dem Erstellen einer Objektinstanz von einer Object-Pascal-Klasse.
- eine vorhandene Datei als OLE-Objekt einbinden, indem Sie den Schalter *Von Datei erstellen* wählen (der Dialog wechselt dann seinen Inhalt, so dass Sie eine Datei auswählen können). Nur in diesem Fall müssen Sie das Objekt nicht einbetten, sondern können auch eine Verknüpfung zur Datei herstellen.

Über den Dialog *Inhalte einfügen* können Sie ein OLE-Objekt aus der Zwischenablage in den OLE-Container einfügen; dieser Dialog ist nur dann im Menü aufgeführt, wenn die Zwischenablage ein OLE-Objekt enthält. Er unterstützt sowohl Verknüpfungen als auch Einbettungen.

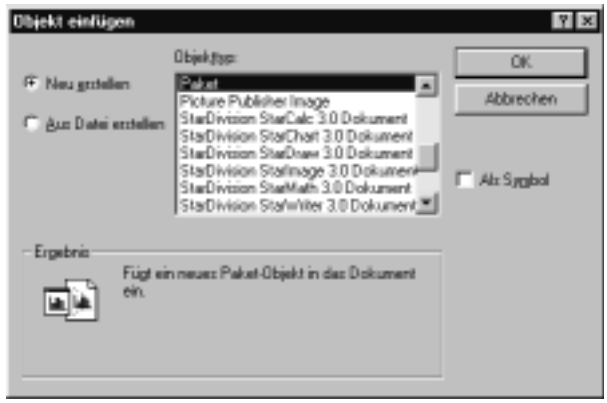


Abbildung 8.2: Der Dialog zum Einfügen von OLE-Objekten

8.2.5 Implementation eines OLE-Clients

Die Komponente *TOleContainer* besitzt einige Methoden, mit denen Sie die übliche Benutzerschnittstelle eines OLE-Clients zum Einfügen und Verwalten von OLE-Objekten besonders einfach aufbauen können.

Die Benutzerführung

Solange der OLE-Container noch leer ist (falls Sie ihm also nicht schon zur Entwurfszeit ein Objekt zugewiesen haben), bestehen grundsätzlich die gleichen Möglichkeiten wie zur Entwurfszeit: Sie können ein neues Objekt erstellen, aus einer Datei erzeugen, von der Zwischenablage kopieren oder per Drag&Drop einfügen (Letzteres nur mit OLE 2.0). Für die ersten beiden Aufgaben stehen Ihnen die beiden schon von der Entwurfszeit bekannten Dialoge zur Verfügung.

Die folgenden Ausschnitte setzen voraus, dass Sie eine *TOleContainer*-Komponente fest im Formular installiert haben, und zeigen, wie Sie zur Laufzeit OLE-Objekte in diese laden können. Sie können in einen Container so oft ein neues Objekt laden, wie Sie wollen: Wenn der OLE-Container schon ein Objekt enthält, gibt die VCL dieses automatisch frei.

Dialoge zum Einfügen

Zur Laufzeit können Sie die beiden schon zur Entwurfszeit aufrufbaren Dialoge auch dem Benutzer Ihrer Anwendung zugänglich machen, indem Sie jeweils eine einfache Methode aufrufen:

```
OleContainer.InsertObjectDialog; // Dialog Objekt einfügen
OleContainer.PasteSpecialDialog; // Dialog Inhalte einfügen
```

Beide Methoden rufen jeweils den Dialog auf und tauschen ein eventuell noch im Container enthaltenes OLE-Objekt durch ein neues aus, sofern der Benutzer im Dialog ein neues Objekt auswählt und den Dialog mit *OK* schließt.

Editieren und Bearbeiten

Das Editieren der OLE-Objekte geschieht normalerweise automatisch: Das Property *AutoActivate* der *TOleContainer*-Komponenten ist bereits mit *aaDoubleClick* vorbelegt, so dass der Benutzer das Objekt mit einem Doppelklick aktivieren kann. Wenn Sie *aaManual* einstellen, müssen Sie selbst für das Editieren sorgen, indem Sie *DoVerb(ovShow)* aufrufen. *aaGetFocus*, der dritte mögliche Wert von *AutoActivate*, ist besonders beim Vor-Ort-Editieren in OLE 2.0 interessant: In diesem Modus schaltet das Objekt sofort in den Editiermodus, wenn die Container-Komponente den Fokus erhält.

Oft bietet ein Menü, das die *Verben* des OLE-Objekts aufzählt, einen alternativen Weg, ein Objekt in den Editiermodus zu schalten. Zu den Verben gehören neben dem *Bearbeiten* des Objekts weitere Verben, die den Fähigkeiten des jeweiligen Objekts entsprechen (in Analogie zum OOP entsprechen die Verben damit den Methoden, die von Windows definierten Klangdateien kennen z.B. das Verb *Abspielen*).

Da die Komponente *TOleContainer* zur Laufzeit freiwillig ein Popup-Menü erzeugt, das alle Verben des Objekts auflistet und ausführen kann, brauchen Sie sich um diese Funktion nicht zu kümmern (es sei denn, diese Funktion ist unerwünscht: dann schalten Sie das Property *AutoVerbMenu* ab).

Dialoge zur Verwaltung der Verbindungen

Schließlich bieten OLE-Clients oft eine Möglichkeit an, die Verbindungs-Attribute eines OLE-Objekts zu verändern. So können Sie verknüpfte Objekte beispielsweise in eingebettete Objekte umwandeln und die automatische Aktualisierung (den »heißen Draht«) an- und abschalten.

In Delphi rufen Sie dazu mit der Methode *TOleContainer.ObjectPropertiesDialog* einen mehrseitigen Eigenschaftsdialog auf, der immer nur für das jeweilige OLE-Objekt zuständig ist und der auch die oben genannten Verknüpfungs-Funktionen bietet, sofern es sich um ein verknüpftes Objekt handelt.

Menüpunkt-Deaktivierung

Zur Implementierung der genannten Menüpunkte gehört normalerweise auch, dass Sie die Menüpunkte deaktivieren, wenn sie gerade nicht sinnvoll anwendbar sind. Der Dialog *Inhalte Einfügen* ist beispielsweise nur dann sinnvoll, wenn die Zwischenablage auch ein OLE-Objekt enthält.

Wie schon im Abschnitt *Formatabfrage für Menüs* in Kapitel 8.1.2 können Sie auch hier eine *OnClick*-Methode für den Hauptmenüpunkt schreiben, der die zu aktivierenden bzw. deaktivierenden Menüpunkte enthält. Diese wird dann immer aufgerufen, wenn der Benutzer das Menü aufklappt.

Ob die einzelnen Menübefehle gerade anwendbar sind, können Sie wie folgt feststellen:

- Ob der Dialog *Inhalte einfügen* verfügbar ist, erfahren Sie von der Methode *TOleContainer.CanPaste*.
- Der Eigenschaftsdialog ist nur anwendbar, wenn der Container ein OLE-Objekt enthält (wenn sein Property *State* nicht den Wert *osEmpty* aufweist).

Die Bearbeitung des *OnClick*-Ereignisses kann beispielsweise wie folgt aussehen (siehe auch *Bearbeiten1Click* im *MultiOle*-Beispiel):

```
InhalteEinfuegen.Enabled := OleContainer.CanPaste;
Eigenschaften.Enabled := OleContainer.State <> osEmpty;
```

8.2.6 Zusammengesetzte Dokumente mit TOleContainer

Das Beispielprogramm *MultiOle*, dessen Formular *OleNotebook* in Abbildung 8.3 gezeigt wird, demonstriert die Speicherung mehrerer OLE-Objekte in einem »Dokument«. Normalerweise steuert die OLE-Client-Anwendung zu einem solchen zusammengesetzten Dokument (*Compound Document*) auch eigene Daten bei, prinzipiell spielt es jedoch keine Rolle, dass die OLE-Objekte im Beispielprogramm die einzigen Daten in diesem Dokument sind.

Dynamische OLE-Objekt-Verwaltung *R159*

OleNotebook kann mehrere OLE-Objekte aufnehmen, die es, wie der Name schon sagt, nach der Art eines Notebooks anordnet. Allerdings verwendet es keine *TNotebook*-, sondern lediglich eine *TTabSet*-Komponente, die die Beschriftung für die einzelnen OLE-Container liefert. Es muss selbst dafür sorgen, dass von allen Containern immer nur einer sichtbar ist, und zwar der über das *TabSet* angewählte (siehe die Methode *ShowContainer* unten).

Zur Verwaltung der OLE-Objekte sieht das Programm die folgenden Methoden vor:

```
function CreateOleContainer: TOleContainer;
procedure AddOleContainer(C: TOleContainer);
procedure ShowContainer(Index: Integer);
```

Abbildung 8.3: Das Beispielprogramm »bündelt« zahlreiche OLE-Objekte in einer Datei – hier findet gerade eine Vor-Ort-Editierung mit eingeblendeten Menü- und Werkzeugleisten statt.

CreateOleContainer erzeugt eine neue *TOleContainer*-Komponente und initialisiert ihre Properties auf passende Werte (*Parent*, *Visible*, *Align* und *AutoSize*); *AddOleContainer* fügt einen mit *CreateOleContainer* erstellten Container der Container-Liste hinzu (die Speicherung der *TOleContainer*-Komponenten findet in einem *TList*-Objekt namens *OleCollection* statt) und die Methode *ShowContainer* zeigt schließlich einen der Container an (sie schlägt quasi die Seite *Index* des nicht vorhandenen *TNotebook*-Elements auf).

Der Menüpunkt BEARBEITEN | NEUES OBJEKT wird beispielsweise wie folgt bearbeitet:

```
procedure TOleNotebook.NeuesObjektClick(Sender: TObject);
var
  NewContainer: TOleContainer;
begin
  NewContainer:=CreateOleContainer;
  if NewContainer.InsertObjectDialog then begin
    AddOleContainer(NewContainer);
    ShowContainer(TabSet.Tabs.Count-1);
  end else NewContainer.Free;
end;
```

Die Verwaltung der OLE-Container ist sehr einfach und soll daher nur an einer der drei genannten Verwaltungsmethoden exemplarisch gezeigt werden: *ShowContainer* macht einen Container aus der Liste *OleCollection* sichtbar, indem sie sein *Visible*-Pro-

perty auf *True* setzt. Sie speichert den Container in der Formularvariablen *VisibleContainer*, damit er das nächste Mal wieder versteckt werden kann:

```
procedure TOleNotebook.ShowContainer(Index: Integer);
begin (* leicht vereinfachte Version *)
  if Assigned(VisibleContainer) then
    VisibleContainer.Visible := False;
  if (Index >= 0) and (OleCollection.Count > Index) then begin
    VisibleContainer := OleCollection[Index];
    VisibleContainer.Visible := True;  { sichtbar machen }
    ActiveControl := VisibleContainer;   { Fokus setzen }
  end else
    VisibleContainer := nil;
end;
```

Das Programm enthält im Bearbeiten-Menü auch einen Punkt *Löschen*, bei dem der aktuelle Container mit *Remove* aus der *OleCollection* entfernt und anschließend mit *Free* mitsamt dem OLE-Objekt freigegeben wird. Da auch das Erstellen der neuen Container in der Hilfsmethode *CreateOleContainer* durch einen einfachen Aufruf von *TOleContainer.Create(self)* sehr einfach ist, wenden wir uns gleich den Dateioperationen zu.

Speichern von OLE-Objekten

TOleContainer vereinfacht die ansonsten komplexen Lade- und Speichervorgänge von OLE-Objekten auf eine einzige Anweisung, die Sie schon von anderen VCL-Klassen her kennen: *SaveToFile* bzw. *LoadFromFile*. Allerdings schreibt *SaveToFile* nur ein einzelnes OLE-Objekt in die Datei, weshalb diese Methoden für ein zusammengesetztes Dokument nicht geeignet sind.

Aus diesem Grund bietet *TOleContainer* auch noch die Methoden *SaveToStream* und *LoadToStream* an, denen Sie einen offenen Stream als Parameter übergeben. Das Beispielprogramm *MultiOle* kann mit diesen Methoden die gesamte OLE-Objektsammlung in einer Datei speichern und an einem Stück wieder laden. Es bearbeitet den Menüpunkt DATEI | SPEICHERN UNTER wie folgt:

```
procedure TOleNotebook.Speichernunter1Click(Sender: TObject);
var
  S: TFileStream;
  I, Count: Integer;
begin
  if SaveDialog.Execute then begin
    S := TFileStream.Create(SaveDialog.FileName, fmCreate);
    try
      Count := OleCollection.Count;
      { Anzahl der Objekte speichern: }
      S.Write(Count, sizeof(Count));
      for i := 0 to Count-1 do
        TOleContainer(OleCollection[i]).SaveToStream(S);
```

```
      finally
        S.Free;
      end;
    end;
  end;
```

Da die *TOleContainer*-Komponente selbst weiß, wo die einzelnen Objekte aufhören, kann das Programm die Objekte direkt aneinander reihen. Die Methode zum Lesen der Objekte funktioniert entsprechend und braucht daher nicht abgedruckt zu werden.

Seit Delphi 2 verwendet *TOleContainer* per Voreinstellung ein Dateiformat, das nicht mit der Komponente von Delphi 1 kompatibel ist, dies können Sie jedoch über das Property *OldStreamFormat* ändern. Das Format der von *MultiOle* geschriebenen Dateien ist sowieso zwischen den Delphi-Versionen unterschiedlich, da es von der Größe des Typs *Integer* abhängig ist (siehe Verwendung von *sizeof* in der obigen Methode).

Hinweis: Selbst wenn Sie nur ein einzelnes eingebettetes OLE-Objekt in eine Datei schreiben, erhalten Sie dann keine Datei, die dem normalen Dateiformat des Servers entspricht. Zum einen kann der Server zum Schreiben der OLE-Objekte ein anderes als sein übliches Dateiformat wählen, zum anderen enthalten die OLE-Dateien zusätzliche OLE-Daten. Dies führt schließlich dazu, dass OLE-Dateien paradoxerweise nur vom OLE-Client gelesen werden können, obwohl nur der Server die darin enthaltenen Objekt-Daten interpretieren kann.

OLE ohne Fenster-Overhead *R158*

Der wohl größte Nachteil von *TOleContainer* ist, dass es sich eben nur um einen Container und nicht um das OLE-Objekt selbst handelt (ähnlich wie die *TImage*-Komponente, die auch *TPictureContainer* heißen könnte, ein *TPicture*-Objekt enthält). Sie können ein OLE-Objekt jedoch auch einzeln ansprechen, wenn Sie sich auf die Ebene des Windows-Component-Object-Models begeben.

TOleContainer besitzt zwei Properties, die auf derartige COM-Schnittstellen weisen: *OleObjectInterface* auf eine *IOleObject*- und *StorageInterface* auf eine *IStorage*-Schnittstelle. *OleObjectInterface* können Sie beispielsweise an die API-Funktion *OleDraw* weitergeben, um das OLE-Objekt auf einem beliebigen *Canvas* auszugeben:

```
OleDraw(ContainerToDraw.OleObjectInterface, DVASPECT_CONTENT,
        Form2.Canvas.Handle, Form2.ClientRect);
```

Der obige Aufruf entstammt dem Beispielprogramm *MultiOle* und zeichnet das OLE-Objekt des aktuellen OLE-Containers auf ein Zweit-Formular (Menüpunkt BEARBEITEN | EXTERNES ZEICHNEN).

8.3 Effektiver Austausch von DLLs

Eines der wichtigsten Grundelemente der Windows-Betriebssystem-Familie sind die dynamischen Link-Bibliotheken. Eine DLL (*Dynamic Link Library*) ist ähnlich wie eine EXE-Datei ein fertig übersetztes Programm-Modul, das jedoch nicht eigenständig ausgeführt wird, sondern anderen Modulen (EXE-Dateien und DLLs) seine Dienste in Form einer Funktions*bibliothek* anbietet.

Anders als beispielsweise Units, die bereits vom Compiler in die ausführbare EXE-Datei eingebunden werden, werden DLLs erst beim Laden der EXE-Datei mit dieser verbunden (dies wird als *dynamisches Linken* bezeichnet). Wie in anderen Sprachen auch können Sie DLL-Funktionen in Object Pascal so aufrufen wie Funktionen, die sich innerhalb der EXE-Datei befinden.

Nachdem schon das ActiveX-Control in Kapitel 6.8 eine DLL war (die Shell-Erweiterung in Kapitel 8.5 wird ein weiteres Beispiel sein), befasst sich dieses Kapitel noch mit »normalen« DLLs, die ihre Dienste anstatt in Form von COM-Klassen durch einfache Funktionen anbieten.

DLLs sind eine grundlegende Möglichkeit, in verschiedenen Programmiersprachen zu entwickeln und so die sich möglicherweise ergänzenden Vorteile von mehreren Entwicklungssystemen ausnutzen zu können. Da Delphi bereits so viele Vorteile verschiedener Systeme vereint, gibt es natürlich nicht viele Entwicklungssysteme, die bei der Entwicklung einer Delphi-Anwendung weitere Vorteile durch DLLs beisteuern könnten. Vorstellbar sind jedoch beispielsweise die folgenden Fälle: Eine Delphi-Anwendung könnte durch nicht-visuelle Funktionen einer C++-DLL unterstützt werden, und umgekehrt könnten visuell entwickelte Delphi-Formulare in andere Anwendungen exportiert werden, wenn es aus irgendwelchen Gründen (noch) nicht möglich ist, ganz auf Delphi umzusteigen.

8.3.1 DLLs in Delphi und C++

Beim Einbinden einer DLL in Delphi oder C++ müssen Sie Dinge beachten, um die Sie sich sonst normalerweise nicht zu kümmern brauchen. Diese Dinge sind zwar lästig, wenn man sie einmal vergisst und so eine Fehlermeldung hervorruft, sie sind jedoch an sich sehr einfach in den Griff zu bekommen.

Für die Verbindung von C++- und Delphi-DLLs gilt wie allgemein auch bei der Verwendung anderer Sprachen, dass DLL und Anwendung in den folgenden Punkten in Übereinstimmung gebracht werden müssen:

- Die aufrufende Anwendung muss den Namen der DLL-Funktionen verstehen. Dies klingt einfach, jedoch hängt ein C++-Compiler per Voreinstellung Typinformationen (die Typen der Parameter und des Rückgabewertes) an den Funktions-

namen an, so dass die Funktion in der DLL nicht mehr über den ursprünglichen Namen angesprochen werden kann.

- Die Typen der Funktionsparameter der DLL müssen in den Sprachen von DLL und Anwendung übereinstimmend deklariert werden. Für einfache Zahlentypen ist das kein Problem; so verwenden Sie z.B. statt des Object-Pascal-Typs *integer* in C++ einfach den Typ *int*. Bei der Übergabe von (Zeigern auf) Records müssen Sie sicherstellen, dass jedes Record-Feld in beiden Sprachen gleich groß ist und an derselben Adresse angeordnet wird (Stichwort *Alignment* / Ausrichtung der Variablen).

  Wenn hinter einer Variablen besondere Funktionalität von Object Pascal steckt, können Sie sie natürlich kaum sinnvoll an eine anderssprachige DLL weitergeben, wenn diese Funktionalität dort unbekannt oder anders implementiert ist. So gibt es z.B. zu den langen Standard-Strings von Object Pascal in C++ keine Entsprechung. Allerdings können Sie diese Strings an eine Delphi-DLL weitergeben, wenn Sie gewisse Bedingungen beachten (siehe hierzu den Kommentar, den Delphi in jede über die Objektgalerie neu erzeugte DLL einfügt).

- Das aufrufende Modul muss sich schließlich an die in der DLL vorausgesetzte Aufrufkonvention halten (Reihenfolge der Parameter und Behandlung des Stacks). Auch in den meisten C++-Systemen stehen Ihnen die in Kapitel 2.5.2 behandelten Aufrufkonventionen von Object Pascal zur Verfügung. So können Sie beispielsweise eine DLL-Funktion sowohl unter Object Pascal als auch unter C++ als *stdcall* deklarieren.

Was dies für die Praxis bedeutet, sehen wir uns nun anhand zweier in Delphi geschriebener Beispiel-DLLs an, von denen die folgende in eine Delphi-Anwendung, die zweite (Kapitel 8.3.2) in eine Borland C++-Anwendung eingebunden wird.

Ein für den Export bestimmtes Formular

FormDLL, die erste Beispiel-DLL, enthält ein besonders einfaches Formular zur Auswahl eines Verzeichnisses (es enthält dazu eine *TDirectoryListBox*- und eine *TDriveComboBox*-Komponente). Dieses Formular benötigt keine einzige selbst geschriebene Methode, da es mit *ShowModal* als modaler Dialog ausgeführt und das ausgewählte Verzeichnis von außen durch die Funktion *SelectDir* aus dem Property *DirectoryListBox1.Directory* ausgelesen wird:

```
function SelectDir(Dest: PChar): Boolean;
var
  DirSelect: TDirSelect;
begin
  Result := False;
  DirSelect := TDirSelect.Create(Application);
  try
    if DirSelect.ShowModal = mrOK then begin
```

```
        StrPCopy(Dest, DirSelect.DirectoryListBox1.Directory);
        Result := True;
      end;
    finally
      DirSelect.Free;
    end;
  end;
```

SelectDir speichert den Inhalt des Properties *DirectoryListBox1.Directory* in dem als Parameter erhaltenen nullterminierten String (dieser ist erforderlich, damit die Funktion auch von C/C++-Code aus bequem aufgerufen werden kann).

Hinweis: *SelectDir* setzt voraus, dass der übergebene String ausreichend groß ist, um jede beliebige Pfadangabe zu übernehmen. Beim Aufruf der Funktion wird dieser String daher später die Länge *MAX_PATH*, die maximal zugelassene Pfadlänge, erhalten.

DLL-Module in Delphi *R154*

Die Funktion *SelectDir* ist die Funktion, die wir nun über eine DLL anderen Anwendungen zur Verfügung stellen werden. Eine DLL wird unter Delphi als eigenständiges Projekt betrachtet. Um eine funktionierende DLL zu ergeben, muss eine Projektdatei die folgenden Voraussetzungen erfüllen:

- Sie muss mit dem Schlüsselwort *library* beginnen,
- sie darf im Hauptprogramm nicht die üblichen Anweisungen wie etwa *Application.Run* oder *Application.CreateForm* enthalten (statt sofort Formulare zu erzeugen, wartet eine DLL vielmehr darauf, von außen aufgerufen zu werden),
- die zu exportierenden Funktionen (die, die von außerhalb der DLL aufgerufen werden sollen) müssen mit der *export*-Direktive deklariert
- und in einem *exports*-Abschnitt der Projektdatei genannt werden.

Unter 32-Bit-Delphi ist die Deklaration der Funktion mit *export* optional, erforderlich ist hier nur noch die Nennung der Funktion im *exports*-Abschnitt.

Das Formular zur Auswahl des Verzeichnisses, in dem auch die Funktion *SelectDir* enthalten ist, wird von Delphi wie immer in einer eigenen Unit angelegt. Diese können wir per *uses*-Klausel in die Projektdatei der DLL einbinden:

```
library FormDll;

uses
  Forms,
  DllForm in 'DLLFORM.PAS' {DirSelect};
```

```
{$R *.RES}

exports
  SelectDir name 'SelectDir';

begin
end.
```

Diese Projektdatei erfüllt bereits die meisten der genannten Bedingungen (*library*, *exports*, Hauptprogramm ohne die üblichen *CreateForm*-Aufrufe).

Die letzte Bedingung, die zu exportierende Funktion mit *export* zu deklarieren, wird von der Unit des Formulars erfüllt. *SelectDir* muss im Interface-Teil deklariert werden, damit sie im *exports*-Abschnitt der Projektdatei verwendet werden kann:

```
unit Dllform;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, FileCtrl;

type
  TDirSelect = class(TForm)
    ...
  end;

var
  DirSelect: TDirSelect;

function SelectDir(Dest: PChar): Boolean; export;

implementation
{$R *.DFM}
... hier folgt SelectDir ...
```

Sie können eine so erzeugte DLL-Projektdatei ganz normal kompilieren, jedoch können Sie sie nicht mit START | START ausführen; zum Testen der DLL benötigen Sie eine Anwendung, die diese DLL oder das von dieser zur Verfügung gestellte Formular einbindet.

Einbinden der DLL *R155*

Wir sehen uns zum Abschluss an, wie die DLL *FormDLL* in eine Delphi-Anwendung importiert werden kann. Auf der CD trägt diese Anwendung den Namen *DLLUser*; sie besteht aus einem Formular mit einem Schalter, über den der modale Dialog der DLL gestartet wird. Das *OnClick*-Ergebnis wird wie folgt bearbeitet:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Ergebnis: array[0..MAX_PATH] of Char;
begin
  SelectDir(Ergebnis);
{  MacheEtwasMitDem(Ergebnis); }
end;
```

Die einzige weitere Aufgabe besteht darin, die Funktion *SelectDir* so zu deklarieren, dass sie aus der DLL eingebunden wird. Da die Funktion in der Pascal-Unit *DllForm* beheimatet ist, könnten wir sie natürlich per *uses DllForm* einbinden; um die Funktion jedoch aus der DLL zu nutzen, muss die Funktion erneut deklariert werden, und zwar wie folgt:

```
function SelectDir(Dest: PChar): Boolean; external 'FORMDLL.DLL';
```

In der *external*-Direktive teilen Sie dem Compiler mit, dass er die Funktion nicht in dieser Unit zu erwarten braucht, sondern dass sie in einer externen DLL namens `formdll.dll` zu finden ist.

Import-Units

Wenn Sie eine DLL mit vielen Funktionen verwenden, ist es empfehlenswert, alle diese Funktionen in einer eigenen Unit zu deklarieren. Die oben gezeigte Deklarationsweise mit dem *external*-Zusatz wird dann im Implementationsteil der Unit verwendet. Kopieren Sie diese Deklaration dann von dort noch einmal in den Interface-Teil und entfernen Sie dort die *external*-Direktive. Die »Implementation« der Funktion besteht dann nicht aus einem Funktionsrumpf, sondern lediglich aus der *external*-Angabe, die dem Compiler mitteilt, wo sich die Funktion befindet.

Einbinden der DLL in C++

Während die Zusammenarbeit zwischen DLL und Anwendung quasi automatisch gelingt, wenn beide in Delphi geschrieben sind, müssen bei der Verbindung von C++ und Delphi weitere Maßnahmen getroffen werden, die im folgenden Kapitel untersucht werden.

8.3.2 Ein fester Formular-Anschluss für C++

Ein großer Nachteil der im letzten Kapitel beschriebenen Methode, Formulare in eine andere Anwendung einzubinden, besteht darin, dass Sie für jedes Formular eine neue funktionale Schnittstelle schreiben müssen, die Eingabeparameter von außen annimmt, um sie als Vorbelegung in die Felder des Formulars zu schreiben, und die die Eingaben des Benutzers als Rückgabewert liefert (so etwa im letzten Kapitel die Funktion *SelectDir*, die allerdings noch *sehr* einfach war und keine Eingabeparameter hatte).

Wenn Sie auf diese Weise viele Formulare entwickeln, wird dieses Schreiben einer DLL-Funktion zu einer ständigen Zusatzarbeit, die den unter Delphi so einfachen Prozess der Formularentwicklung erheblich verkompliziert.

Wünschenswert wäre es, wenn Sie ein Formular nach der visuellen Entwicklung in Delphi *sofort* von C++ oder einer anderen Sprache aus ansprechen könnten – ohne eine neue »Verpackungs-Funktion« wie *SelectDir*.

Dieses Kapitel stellt eine feste Schnittstelle vor, mit der Sie diesen Idealfall *fast* erreichen (nur fast, weil Sie für jedes Formular eine Extra-Zeile schreiben müssen). Sobald Sie das Formular entworfen, die Extra-Zeile geschrieben und die DLL neu übersetzt haben, können Sie aus einer Anwendung z.B. den Inhalt von Eingabefeldern und anderen Steuerelementen setzen, auslesen und die Steuerelemente aktivieren und deaktivieren. Der Schlüssel zu dieser Schnittstelle sind unter anderem ein weiteres Mal die Laufzeit-Typinformationen (RTTI, siehe Kapitel 2.3.6), virtuelle Konstruktoren und Klassenreferenzen (Kapitel 2.3.4).

Verwendung der Schnittstelle in C++

Der folgende C++-Code (auf der CD zu finden im Borland C++-Projekt *FormCall.ide*) ruft beispielsweise das in Abbildung 8.4 gezeigte Formular auf. Es verändert vorher Einstellungen der Komponenten *Label1*, *Edit1* und *TopProgressBar* und liest, falls das Formular mit dem *OK*-Schalter beendet wurde, den Wert des Editierfelds *Edit1* aus, um ihn in einem Meldungsfenster anzuzeigen:

```
// Dialog vorbereiten
DelphiDlg Dlg = PrepareDialog("TDllTestForm");
SetControlString(Dlg, "Label1", "Geben Sie einen Text ein:");
SetControlString(Dlg, "Edit1", "Vorgabe-Text");
SetControlValue(Dlg, "TopProgressBar", 50);
// Dialog ausführen
char Input[300], MsgBuf[300];
if (ExecuteDialog(Dlg) == 1) {
  // Dialog auslesen
  GetControlString(Dlg, "Edit1", Input, sizeof(Input));
  sprintf(MsgBuf, "Die Eingabe von \"%s\" wurde aus dem "
                  "Delphi-Dialog ausgelesen.", Input);
} else strcpy(MsgBuf, "Dialog wurde abgebrochen.");
MessageBox(0, MsgBuf, "Delphi-DLL-Test", 0);
FreeDlg(Dlg);
```

Alle in diesem Beispiel verwendeten Funktionen sind in der Unit *DelphInt* enthalten, die auf der CD enthalten ist; ihre Funktionen sind am Schluss dieses Kapitels zusammengefasst.

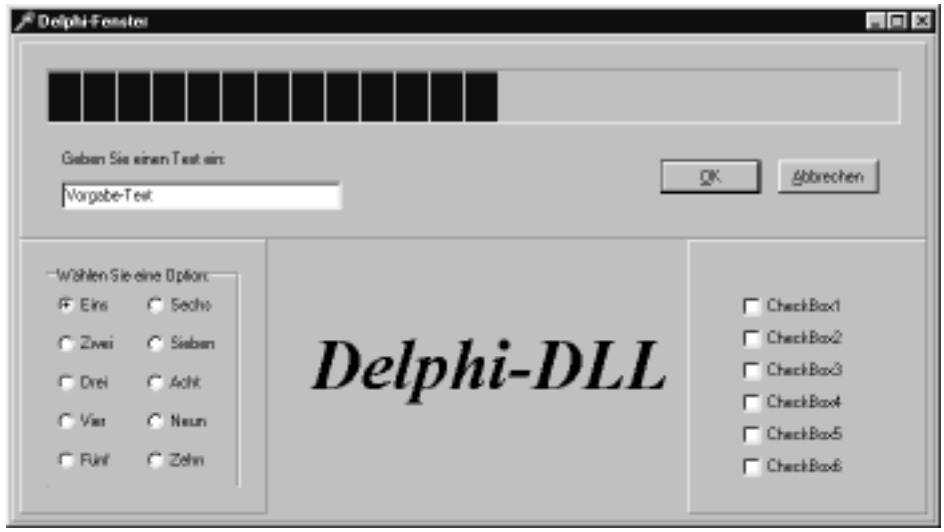


Abbildung 8.4: Ein Beispielformular, dessen Komponenten durch C++-Code angesprochen werden

Funktion der Unit DelphInt

Die Besonderheit des oben gezeigten C++-Codes ist die Verwendung des Klassennamens *TDllTestForm* und der Namen der Komponenten wie *Label1*. Die Aufgabe der DLL ist es dann, zur Laufzeit die entsprechende Klasse bzw. Komponente zu finden.

So ist es Aufgabe der Funktion *PrepareDialog*, eine Klassenreferenz auf die als Parameter übergebene Formularklasse zu finden und über den virtuellen Konstruktor ein neues Objekt dieser Klasse zu erzeugen. *PrepareDialog* gibt einen Zeiger auf den Dialog zurück, den die C++-Anwendung wieder den anderen Delphi-Funktionen als Parameter übergeben kann (dieser Zeiger ist also für den C++-Code eine Art Handle):

```
function PrepareDialog(Name: PChar): TForm;
var
  FormClass: TFormClass;
  Dialog: TForm;
begin
  try
    FormClass := TFormClass(FindClass(StrPas(Name)));
    Dialog := FormClass.Create(Application);
    Result := Dialog;
  except
    Result := nil;
  end;
end;
```

Die VCL-Routine *FindClass* wurde bereits in Kapitel 4.3.3 im Zusammenhang mit der Speicherung polymorpher Objekte beschrieben. Sie setzt voraus, dass die gesuchte

Klasse vorher mit einem Aufruf von *RegisterClass* registriert wurde. Dies ist auch der Grund, warum Sie für jedes Formular, das Sie von einer anderen Anwendung aufrufen lassen wollen, eine »Extra-Zeile« schreiben müssen – einen Aufruf von *RegisterClass*, der am besten im Initialisierungsabschnitt der Unit aufgehoben ist.

Zum Finden der Komponenten anhand ihres Namens ist keine explizite Registrierung erforderlich: Hierzu ist die Methode *FindComponent* der Klasse *TComponent* (siehe auch Kapitel 3.1.2) auch so in der Lage. Die Prozedur *SetControlValue* verwendet sie, um die im String-Parameter angegebene Komponente in dem Formular ausfindig zu machen, das ihr als erster Parameter übergeben wird (und das vorher in *PrepareDialog* erzeugt wurde). Mit Hilfe der Laufzeittypinformationen kann sie schließlich herausfinden, welche Klasse hinter dieser Komponente steckt. Je nach Klasse ist der zentrale Wert, den die Komponente darstellt, in einem anderen Property zu finden. In *TEdit* ist es beispielsweise der *Text*, in *TCheckBox* das Property *Checked* und in *TProgressBar* das Property *Value*.

Das folgende Listing zeigt, wie Sie die genannten Komponentenklassen auf diese Art identifizieren und entsprechend behandeln können. Natürlich ist es Ihnen überlassen, was Sie als »Wert« einer Komponente ansehen (so könnte *SetControlValue* – oder eine andere Funktion – auch das Property *Tag* setzen, das in jeder Komponente vorhanden ist).

```
procedure SetControlValue(Dialog: TForm; Control: PChar; Val: LongInt);
var
  C: TComponent;
begin
  if Dialog = NIL then exit;
  C := Dialog.FindComponent(StrPas(Control));
  if c is TCheckBox then with c as TCheckBox do begin
    Checked := Val<>0;
  end else if c is TEdit then with c as TEdit do begin
    Text := IntToStr(Val);
  end else if c is TProgressBar then with c as TProgressBar do begin
    Position := Val;
  ... hier folgt die Abfrage weiterer Komponenten ...
end;
```

Die anderen Funktionen der Unit gehen auf ähnliche Weise vor und brauchen daher nicht abgedruckt zu werden. Wichtig ist natürlich insbesondere, dass *DelphInt* die Funktionen *ExecuteDialog* und *FreeDlg* bereitstellt, die die *TForm*-Methoden *ShowModal* bzw. *Free* aufrufen (siehe CD).

Anwenden der Unit DelphInt *R156*

Die Verwendung der Unit *DelphInt* zur Erstellung von DLLs wird auf der CD von der Delphi-DLL *InDelphi* demonstriert: Erstellen Sie einfach ein DLL-Projekt, in das Sie

zunächst die Unit *DelphInt* einbinden. Dann können Sie dem Projekt beliebige Formulare hinzufügen und visuell entwerfen (um die Formulare auch in Aktion zu testen und eventuell zu debuggen, müssen Sie sie natürlich in den Rahmen eines normalen Projekts aufnehmen, da DLLs alleine nicht ausführbar sind).

Für jedes Formular, das Sie von außen ansprechen wollen, schreiben Sie eine Zeile der Art

```
RegisterClass(TDllTestForm);
```

in den Initialisierungsteil der Unit oder in das Hauptprogramm der DLL. Übersetzen Sie die DLL dann mit PROJEKT | *[Projektname] ERZEUGEN*.

Verbindung der DLL mit der Anwendung

Um die Funktionen der DLL nun aus einer anderen Anwendung ansprechen zu können, müssen Sie die oben verwendeten Funktionen wie z. B *PrepareDialog* einmalig in diese Anwendung importieren. In Kapitel 8.3.1 wurde bereits die Einbindung von Delphi-DLLs in andere Delphi-Projekte demonstriert; hier sehen wir uns an, wie dasselbe für C++-Anwendungen geht.

Die C++-Header-Datei *indelphi.h* deklariert zunächst alle von der DLL bereitgestellten Funktionen. Dabei werden die drei am Anfang von Kapitel 8.3.1 genannten Voraussetzungen für eine funktionierende Verbindung zwischen DLL und Anwendung wie folgt erfüllt:

- Alle Deklarationen werden in einen *external »C« { ... }*-Block eingeschlossen. Dies bewirkt, dass der C++-Compiler die Funktionsnamen nicht um Typinformationen erweitert.
- Die Parametertypen von Object Pascal werden nach C++ übersetzt, indem per *typedef*-Anweisung neue Typen deklariert werden, die den Namen der Pascal-Typen tragen. So können Sie in der Deklaration der Funktionen dieselben Typenbezeichner verwenden wie in Object Pascal, lediglich die Deklarationssyntax muss noch an C++ angepasst werden.
- Schließlich wird die Aufrufkonvention in Borland C++ durch das Schlüsselwort *__pascal* eingestellt. Entsprechend sind die Funktionen natürlich auch unter Delphi in dieser Art deklariert worden.

Das folgende Listing zeigt exemplarisch einige der Deklarationen aus *indelphi.h* (unter 16-Bit-Windows müssen Sie eventuell noch ein *FAR* hinzufügen; die genauen Unterschiede zwischen 16- und 32-Bit finden Sie im Quelltext auf der CD, der dies per bedingter Kompilierung unterscheidet):

```
typedef char Boolean;
typedef void *DelphiDlg;
```

```
typedef char *PChar;
typedef long LongInt;

extern "C" {
  DelphiDlg __pascal PrepareDialog(PChar Name);
  void __pascal SetControlString(DelphiDlg Dialog,
                PChar Control, PChar Val);
  void __pascal SetControlValue(DelphiDlg Dialog,
                 PChar Control, LongInt Val);
  int __pascal ExecuteDialog(DelphiDlg Dialog);
  ...
}
```

Als Letztes benötigen Sie noch etwas, das der *external*-Deklaration in Delphi entspricht, damit der Linker von C++ die deklarierten DLL-Funktionen nicht vergeblich in den C++-Bibliotheken sucht. Hierzu gibt es in C++ je nach Entwicklungsumgebung verschiedene Möglichkeiten (unter Borland C++ 5.0 sind die Möglichkeiten der Definitionsdatei, der *_import*-Direktive und der Importbibliothek dokumentiert).

Das Projekt *FormCall* auf der CD verwendet eine Importbibliothek. Diese erzeugen Sie am einfachsten von der DOS-Befehlszeile mit dem Befehl *ImpLib indelphi.lib indelphi.dll*. Sie erhalten dadurch eine Bibliothek namens *indelphi.lib*, die dem C++-Linker alle in *indelphi.dll* exportieren Funktionsnamen nennt. Diese müssen Sie noch in Ihr C++-Projekt einbinden, bevor Sie es erfolgreich linken können.

Hinweis: Wenn Sie umgekehrt eine C++-DLL aus Delphi aufrufen wollen, deklarieren Sie die Funktionen in C++ so wie oben die importierten Funktionen als *extern »C«* ; in Delphi können Sie diese dann so importieren, wie in Kapitel 8.3.1 am Beispiel der Unit *DLLUser* gezeigt. Sie müssen allerdings noch sicherstellen, dass Delphi die richtige Aufrufkonvention verwendet. Sind die Funktionen in C++ wie oben als *__pascal* deklariert, müssen Sie in 32-Bit-Delphi die *pascal*-Direktive hinter die Funktionsdeklaration schreiben (*pascal* war lediglich in Delphi 1 die voreingestellte Aufrufkonvention).

Funktionsübersicht

Die folgende Tabelle fasst die wichtigsten Funktionen der Unit *DelphInt* zusammen:

Funktion	Beschreibung
PrepareDialog	sucht die als Parameter angegebene, registrierte Formularklasse, erzeugt ein Formular davon und gibt einen Zeiger (Handle) auf dieses zurück.
ExecuteDialog	führt das Dialogformular, dessen Handle Sie als Parameter angeben, aus und gibt den Rückgabewert der Methode *ShowModal* zurück.

Funktion	Beschreibung
SetControlValue	setzt einen bestimmten Wert der als String-Parameter angegebenen Komponente. Was dieser Wert ist, hängt vom Typ der Komponente ab (siehe obiges Listing).
SetControlString	wie *SetControlValue*, setzt den Wert der Komponente jedoch auf einen String, sofern der Typ der Komponente dies erlaubt.
GetControlValue	liefert den durch die Komponente dargestellten Wert zurück, der durch *SetControlValue* gesetzt werden kann.
GetControlString	entspricht *GetControlValue* mit dem Unterschied, dass ein String geliefert wird, falls es der Typ der Komponente zulässt.
FreeDlg	gibt den Dialog frei.

Hinzu kommen zwei Funktionen, mit denen Sie einzelne Komponenten des Dialogs von der Anwendung aus verstecken und sichtbar machen können (*ShowControl*) und mit der Sie sie aktivieren und deaktivieren können (*EnableControl*). Je nach Bedarf werden Sie wahrscheinlich weitere Funktionen hinzufügen wollen, wenn Sie diese Schnittstelle in einer C++- oder einer anderen Anwendung verwenden sollten.

8.3.3 Objektaustausch zwischen C++ und Delphi

Zum Schluss soll hier noch eine Technik vorgestellt werden, mit der sich auch Objekte zwischen Borland C++ und Object Pascal austauschen lassen, und zwar ohne die Zuhilfenahme des Component Object Models. Als Beispiel dafür dient die im Anhang näher erläuterte und praktisch verwendete C++-DLL ITEADLL, die dem aufrufenden Delphi-Programm Objekte zur Verfügung stellt, welche in Object Pascal auf »natürliche« Weise über Methoden aufgerufen werden können. Das »Delphi-Programm« kann dabei übrigens selbst wieder eine DLL sein, wie etwa das in Anhang A vorgestellte Experten-Package, das unter anderem den nachgebauten CodeExplorer enthält.

Virtuelle Methodentabellen

Der an dieser Stelle entscheidende Unterschied zwischen Funktionen und Methoden liegt darin, dass sich Methoden nicht wie die Funktionen in Kapitel 8.3.1 exportieren lassen. Ebenfalls nicht möglich ist es, einfach die gesamte C++-Klasse mit einer *external*-Direktive in eine Object-Pascal-Unit zu importieren.

Die Grundlage für den Austausch von Objekten sind die virtuellen Methoden. Da virtuelle Methoden bereits per Definition über eine späte Bindung aufgerufen werden, ist es nicht notwendig, dass ihre Aufrufe schon beim Laden der DLL an konkrete Adressen gebunden werden. Dies wiederum erspart uns die Notwendigkeit, sie mit *export* und *external* zu ex- und importieren.

Der Aufruf einer virtuellen Methode sieht so aus, dass das Programm zur Laufzeit in einem intern verwalteten Teil des Objekts, der VMT (Virtual Method Table), nachsieht,

an welcher Adresse sich die benötigte virtuelle Methode befindet, und sie dann erst aufruft. Solange also eine solche VMT zur Verfügung steht, können virtuelle Methoden auch dann problemlos aufgerufen werden, wenn sie sich in einer anderen DLL befinden. Und da Borland C++ und Object Pascal für ihre Objekte VMTs des gleichen Formats verwenden, steht dem Austausch von Objekten nichts mehr im Wege.

Klassendeklaration in C++ und Object Pacal *R134*

Nehmen wir beispielsweise die folgende Klassendeklaration in C++ an (sie stammt aus der erwähnten `iteadll.dll`):

```
class ScopeHandler {
  ... private Variablen ...
public:
  ScopeHandler(CIdentifier *_i);
  ~ScopeHandler();
  virtual void GetName(char *Dest, int DestLen);
  virtual ScopeHandler *GetParent();
  virtual void GetDeclPos(int *Line, int *Col);
  ...
```

Da diese Klasse nicht von einer anderen Klasse abgeleitet ist, erhält ihre erste virtuelle Methode den ersten Eintrag der VMT, die anderen Methoden folgen entsprechend. Um dieselbe VMT-Reihenfolge in Object Pascal zu erhalten, wird dort ebenfalls eine Klasse *ScopeHandler* deklariert:

```
ScopeHandler = class
  procedure CopyName(Dest: PChar; Len: Integer); virtual; cdecl; abstract;
  function GetParent: ScopeHandler; virtual; cdecl; abstract;
  procedure GetDeclPos(var Line, Col: Integer); virtual; cdecl; abstract;
  ...
```

Alle virtuellen Methoden der C++-Klasse werden in derselben Reihenfolge als *virtual* deklariert. Die *cdecl*-Direktive sorgt dafür, dass die Methoden mit dem Aufrufformat von C angesprochen werden, und die *abstract*-Direktive erspart Ihnen die Notwendigkeit, die Methodenrümpfe im Implementationsteil der Unit definieren zu müssen. Bei genauem Vergleich der beiden Deklarationen werden Sie feststellen, dass die Deklarationen verschiedene Namen für die erste Methode angeben. Dies spielt jedoch keine Rolle, da – anders als beim Linken von exportierten Funktionen – beim Linken von Objekten über VMTs niemand eine übereinstimmende Namensgebung überwacht.

Nicht-virtuelle Methoden

Wie verhält es sich nun mit nicht-virtuellen Methoden der C++-Klasse? Diese dürfen in der Object-Pascal-Klassendeklaration logischerweise nicht als virtuell deklariert werden, weil dann die nachfolgenden Methoden einen falschen VMT-Index erhalten würden. Sie können allerdings als normale Methoden in Delphi deklariert werden – nur

werden, wenn Sie diese Methoden aufrufen, keine Methoden der DLL angesprochen, was auch ganz logisch ist, denn statische (und folglich nicht-abstrakte) Methoden *müssen* im Implementationsteil der Unit definiert werden.

Dies führt uns zu der interessanten Möglichkeit, dass wir die aus C++ übernommene Klasse in Object Pascal durch *statische* Methoden erweitern können. Rufen wir für ein Objekt dieser Klasse eine statische Methode auf, wird eine Object-Pascal-Methode aufgerufen, und zwar direkt über eine vom Compiler eingetragene Adresse. Bei Aufruf einer virtuellen Methode verzweigt das Programm dank der VMT in die DLL. Die Klasse *ScopeHandler* wurde in Object Pascal beispielsweise um eine *Free*-Methode erweitert, deren Existenzberechtigung sich aus dem nächsten Abschnitt ergibt.

Lebenslauf eines C++-Objekts

Die Besonderheit gegenüber der Verwendung eines echten Object-Pascal-Objekts liegt darin, dass Sie ein C++-Objekt in Object Pascal nicht mit einem Konstruktoraufruf konstruieren können, denn solange es kein Objekt gibt, gibt es auch keine VMT und somit keine Möglichkeit, über die VMT in die DLL zu springen. Was bleibt, ist die herkömmliche Methode, in die DLL zu verzweigen. Die DLL muss also zumindest eine normale Funktion exportieren, die ein erstes C++-Objekt erzeugt. In der Itéa-DLL ist dies die Funktion *CreateParser:*

```
CBPParser* _export CreateParser()
{
  return new CBPParser; // ruft den C++-Konstruktor auf.
}
```

Durch den Konstruktoraufruf in der DLL entsteht ein neues Objekt mit einer VMT. Die Funktion liefert einen Zeiger auf dieses Objekt zurück. Dieser Zeiger ist mit einer Objektvariablen von Object Pascal voll kompatibel, auch wenn ein Objekt in Delphi nicht wie ein Zeiger behandelt wird.

CreateParser wird von Object Pascal aus wie folgt aufgerufen:

```
// Aus Itea.pas:
initialization
  GlobalParser := _CreateParser;
```

Daraufhin lassen sich die virtuellen Methoden von *GlobalParser* wie gewohnt aufrufen. Zur Freigabe des Objekts lässt sich nun ohne weiteres auch eine Methode verwenden, sofern sie virtuell ist. Diese Vorgehensweise hat jedoch einen kleinen Haken: Von Object-Pascal-Objekten sind Sie vielleicht gewohnt, dass Sie *Free* auch dann aufrufen können, wenn das Objekt *nil* ist. Dies liegt daran, dass *Free* in Object Pascal *keine* virtuelle Methode ist und aus einer Abfrage ähnlich der folgenden besteht:

```
if Assigned(self) then Destroy; // Destroy ist der (virtuelle!) Destruktor
```

Diese Abfrage ist erforderlich, da der Aufruf von virtuellen Methoden über ein *nil*-Objekt (mangels VMT) in einer Schutzverletzung endet. Daher fügt die Unit *Itea* der Klasse *ScopeHandler* eine neue, nicht-virtuelle *_Free*-Methode hinzu (nach dem im Abschnitt *Nicht-virtuelle Methoden* beschriebenen Prinzip):

```
procedure ScopeHandler.Free;
begin
  if Assigned(self) then _Free; // _Free ist die virtuelle C++-Methode.
end;
```

Object-Pascal-Objekte an C++ übergeben

Es ist natürlich auch umgekehrt möglich, Objekte in Object Pascal zu konstruieren und in C+ aufzurufen. Das Delphi-Package mit den Beispielexperten auf der CD demonstriert dies in der Unit *IteaCodeExplorer*, die eine Klasse *CStatusReporter* implementiert. Diese ist von der in der DLL definierten Klasse *CAbstractStatusReporter* abgeleitet. Die DLL-Methode *CParser.ParseMainModuleX* erwartet als Parameter ein Objekt dieser abstrakten Klasse, dessen Methoden während der Ausführung von *ParseMainModuleX* aufgerufen werden, um dem Delphi-Programm den Fortschritt des Parsens anzuzeigen. Die Methode *TIteaCodeExplorerForm.ExploreUnit* kann ganz normal ein Objekt der Klasse *CStatusReporter* konstruieren und an die C++-DLL weitergeben. Das Ergebnis davon sehen Sie bei Verwendung des Itéa-CodeExplorers in der Statuszeile des Explorer-Fensters: Dauert ein Parse-Vorgang einige Momente, finden Sie in dieser Zeile Angaben über den Fortschritt des Parsens. Diese Angaben werden durch eine von C++ aufgerufene *CStatusReporter*-Methode ausgegeben.

8.4 Verwendung externer COM-Klassen

In diesem Kapitel geht es einmal nicht um die Komponenten der VCL, sondern um die »Komponenten« von Windows bzw. um dessen COM-Schnittstellen, die den Interfaces von Delphi entsprechen.

Wegweiser

Nachdem Kapitel 2.7 bereits eine Einführung in das Component Object Model gegeben hat und in anderen Kapiteln gezeigt wird, wie Sie eigene COM-Objekte *erzeugen* (ActiveX-Steuerelemente in Kapitel 6.8, Shell-Erweiterungen in Kapitel 8.5 und COM-Automationsobjekte in Kapitel 8.7), geht es in diesem Kapitel darum, COM-Schnittstellen, die von anderen Anwendungen bzw. DLLs zur Verfügung gestellt werden, zu benutzen.

Speziell wird es um die Interfaces der Windows-Shell gehen, und zwar zunächst eher theoretisch am Beispiel der neuen Shell-Komponenten von Delphi 6, dann an einem

kleinen praktischen Beispiel zum Einrichten von Shell-Links (Kapitel 8.4.2) und zum Schluss an einem größeren Beispiel zur Verwendung der *IShellFolder*-Schnittstelle in einem Programm zur Nachbildung des Windows-Explorers (8.4.3).

8.4.1 Von Schnittstellen und Objekten

Windows-Objekte (genauer COM-Objekte) haben von außen gesehen keine besondere Ähnlichkeit mit den Objekten einer objektorientierten Programmiersprache wie Object Pascal, denn sie sind nur über eine Vielzahl von COM-Schnittstellen ansprechbar, wobei man von außen nicht feststellen kann, welche dieser Schnittstellen intern zu einem bestimmten Objekt bzw. einer bestimmten Klasse gehören.

Schnittstellen versus API-Funktionen

Über die Methoden der Schnittstellen können Sie ein Windows-Objekt beispielsweise manipulieren oder seine Daten abrufen. Diese Methoden ersetzen immer mehr die bisherigen API-Funktionen. Anders ausgedrückt fasst eine Schnittstelle mehrere sinnverwandte Funktionen, die bisher als einzelne API-Funktionen vorgelegen haben, zu einer Einheit zusammen.

Wenn Sie diese Funktionen von einer objektorientierten Sprache wie Object Pascal aus aufrufen, ist es auch nicht mehr notwendig, ein Handle als ersten Parameter anzugeben, sondern Sie können die Schnittstellenfunktionen wie die Methoden eines Objekts aufrufen, wobei das »Objekt« hier *eine* von mehreren Schnittstellen eines Windows-Objekts ist. Als Ganzes lässt sich das Windows-Objekt nicht ansprechen. Sie können es nur über einzelne Schnittstellen ansprechen, diese Schnittstellen selbst jedoch wieder als Objekte ansehen, wenn Sie in Ihrer Anwendung darauf zugreifen.

Windows-Objekte und Delphi-Objekte

Die Zusammenhänge zwischen Delphi-Objekten, Delphi-Klassen, Windows-Objekten und Windows-Schnittstellen können am besten anhand eines Beispiels deutlich gemacht werden. Als Beispiel sollen hier die Ordner der Windows-Shell dienen, deren Details an dieser Stelle nicht von Bedeutung sind.

Da die Ordner der Windows-Shell einander ähnliche Objekte sind, könnte man sie zunächst einmal in einer Klasse »Ordner der Windows-Shell« zusammenfassen. Nach außen hin ist eine solche Klasse jedoch nicht definiert; das Ordner-Objekt lässt sich nur über verschiedene Interfaces ansprechen, die das Objekt quasi aus verschiedenen Perspektiven betrachten.

Nehmen wir an, wir wollten mit zwei dieser Shell-Ordner arbeiten, und zwar mit den Ordnern *Arbeitsplatz* und *C:* (dass der Ordner *C:* im Ordner *Arbeitsplatz* enthalten ist, spielt hier keine Rolle). Wir haben es bei diesen Ordnern mit zwei »Windows-Objek-

ten« zu tun. Von diesen Windows-Objekten könnten wir nun z.B. die folgenden Delphi-Variablen erzeugen, die Sie vereinfachend jeweils als Objekt ansehen können:

- *Objekt1* = *IUnknown*-Schnittstelle von »Ordner C:\«
- *Objekt2* = *IShellFolder*-Schnittstelle von »Ordner C:\«
- *Objekt3* = *IShellFolder*-Schnittstelle von »Arbeitsplatz-Ordner«
- *Objekt4* = *IEnumObjects*-Schnittstelle von »Arbeitsplatz-Ordner«
- *Objekt5* = *IShellFolder2*-Schnittstelle von »Arbeitsplatz-Ordner«

Da jede Schnittstelle durch die Vererbung bereits eine *IUnknown*-Schnittstelle enthält, ist *Objekt1* in diesem Beispiel unnötig, da *Objekt2* alle Möglichkeiten von *Objekt1* mit einschließt.

Wichtig ist, wie schon erwähnt, dass es kein Delphi-Objekt gibt, das für einen ganzen Ordner steht bzw. alle Methoden eines Ordners auf einmal anbietet. Weitere Schnittstellenobjekte zum selben Objekt können Sie über die Methode *IUnknown.QueryInterface* oder mit dem *as*-Operator von Object Pascal erfragen. Im Beispielprogramm dieses Kapitels werden weder *QueryInterface* noch *as* benötigt, da es noch weitere Möglichkeiten gibt, an andere Schnittstellen heranzukommen.

Erzeugen und Verwenden eines COM-Objekts

Bevor wir in Kapitel 8.4.2 auf die konkrete Anwendung des Component Object Models zu sprechen kommen, soll hier kurz ein Beispiel zeigen, wie einfach die Verwendung einer COM-Schnittstelle der Windows-Shell ist. Um Zugriff auf eine solche Schnittstelle zu erhalten, benötigen Sie den Dienst einer API-Funktion. Um etwa die *IShellFolder*-Schnittstelle des Desktop-Ordners der Windows-Shell (der im Explorer als Wurzel aller anderen Objekte dargestellt wird) zu erhalten, rufen Sie die Shell-API-Funktion *SHGetDesktopFolder* auf:

```
var
  Desktop: IShellFolder;
begin
  SHGetDesktopFolder(Desktop);
```

Nach diesem Aufruf können Sie diese Schnittstelle über das Objekt *Desktop* verwenden, indem Sie wie gewohnt Methoden aufrufen. Trotzdem unterscheidet sich das *Desktop*-Objekt stark von normalen Delphi-Objekten:

- Das Objekt wird nicht innerhalb der Delphi-Anwendung erzeugt, sondern in einer Shell-DLL, die im Adressraum der Delphi-Anwendung eingeblendet wird. Sie können daher weder einen Konstruktor aufrufen, um ein *IUnknown*-Objekt zu erzeugen, noch einen Destruktor, um es freizugeben. Tatsächlich brauchen Sie sich um

die Freigabe überhaupt nicht zu kümmern, denn seit Delphi 3 verwaltet die VCL COM-Objekte automatisch (siehe hierzu auch Kapitel 8.5.5).

- Auch das Objekt selbst und all seine Methoden befinden sich außerhalb der Delphi-Anwendung. Alle darauf folgenden Methodenaufrufe für das oben gezeigte *Desktop*-Objekt führen daher aus der Delphi-Anwendung hinaus (wie der Aufruf einer normalen API-Funktion).

Je nach benötigter Schnittstelle müssen Sie natürlich eine andere API-Funktion verwenden, um ein erstes Exemplar dieser Schnittstelle zu erhalten. Eine allgemeine Funktion, die für viele Schnittstellentypen verwendet wird, ist *CoCreateInstance*. In Kapitel 8.4.2 wird sie beispielsweise zur Initialisierung der Variablen *ShellLink* (Interface-Typ *IShellLinkA*) verwendet:

```
CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
                 IID_IShellLinkA, ShellLink);
```

Shell-Interfaces in TShellTreeView und TShellListView

Die bereits in Kapitel 1.9.3 erwähnten neuen Shell-Ansichts-Beispielkomponenten von Delphi 6 stellen einen einfachen Weg zur Verfügung, einige dieser Schnittstellen wie in einem Zoo bewundern zu können: Sowohl *TShellTreeView* als auch *TShellListView* halten für jedes angezeigte Shell-Objekt jeweils drei verschiedene Shell-Interfaces über Properties abrufbereit, die jedoch aufgrund der Funktionalität der beiden Komponenten kaum direkt angesprochen werden müssen. Es handelt sich um die Schnittstellen *IShellFolder*, *IShellFolder2* und *IShellDetails*, mit denen sich unter vielen anderen die folgenden Informationen ermitteln lassen:

- die Bezeichnung eines Shell-Objekts, wie sie im Windows-Explorer angezeigt wird, z.B. »Desktop«, »Lokaler Datenträger (C:)«
- der Pfad des Objekts im Dateisystem, z.B. »D:\Dokumente und Einstellungen\Administrator\Desktop« oder »C:\« (Bezeichnung und Pfad werden von *IShellFolder.GetDisplayNameOf* geliefert)
- die Fähigkeiten des Objekts, z.B. ob man es kopieren, löschen, verschieben, umbenennen kann oder ob es eine Eigenschaftsseite besitzt
- weitere Attribute wie z.B. ob es sich um einen Link oder um ein Dateisystem handelt (Fähigkeiten und Attribute werden von *IShellFolder.GetAttributesOf* zurückgeliefert).

Es ist ein wesentlicher Vorteil der Komponenten *TShellTreeView* und *TShellListView*, dass sie die genannten Informationen über Properties wie *DisplayName*, *PathName*, *Capabilities* und *Properties* zur Verfügung stellen. Um diese Informationen tatsächlich

mit den oben genannten *IShellFolder*-Schnittstellen zu ermitteln, wäre die komplizierte Arbeit mit *Item Identifier Lists* notwendig, die in Kapitel 8.4.3 am Beispielprogramm *ShellExplorer* genauer erläutert wird.

Der Zugriff auf die genannten Properties geschieht über Datenstrukturen des Typs *TShellFolder*. Für jedes angezeigte Objekt speichern die beiden Komponenten jeweils eine solche Datenstruktur. Diese wird jedoch – abweichend von der üblichen Verhaltensweise von TreeViews und ListViews – nicht etwa in den *Items* der Componenten (etwa in *Items.Data*) gespeichert, sondern in den Properties *Folders* und *SelectedFolders*. Folgende Zeile findet beispielsweise den Pfad des ausgewälten TreeView-Ordners heraus:

```
Label.Caption := ShellTreeView1.SelectedFolder.DisplayName;
```

Das Beispielprogramm *NewControlsTest* (Abbildung 8.5) füllt eine *ComboBox1* mit den *DisplayNames* der markierten Objekte im ShellListView:

```
procedure TForm1.ComboBox1DropDown(Sender: TObject);
// verknüpft mit ComboBox1.OnDropDown
// -> die Liste wird vor jedem Aufklappen erneuert.
var
  i: Integer;
begin
  ComboBox1.Items.Clear;
  for i := 0 to ShellListView1.Items.Count - 1 do begin
    if ShellListView1.Items[i].Selected then
      ComboBox1.Items.Add(ShellListView1.Folders[i].DisplayName);
  end;
end;
```

Weitere Informationen über die Einträge und Shell-Objekte zeigt das Beispielprogramm in einem *ValueListEditor*. Dies ist allerdings eher ein Beispiel für die Verwendung der von Delphi 6 eingeführten *TValueListEditor*-Komponente und der Klasse *Strings*, daher soll hier ein kleiner Auszug aus der Methode genügen, die bei jeder Änderung der Auswahl im TreeView ausgeführt wird (zum ValueListEditor siehe auch Abbildung 8.5):

```
procedure TForm1.ShellTreeView1Change(Sender: TObject; Node: TTreeNode);
var
  i: integer;
begin
  // Aktualisierung des ValueListEditors unterbinden, bis alle
  // Einträge hinzugefügt wurden:
  ValueListEditor1.Strings.BeginUpdate;
  try
    ValueListEditor1.Strings.Clear;
    with ValueListEditor1.Strings, ShellTreeView1.SelectedFolder do begin
      // Schlüssel-Wertpaare der Form "S=W" werden von ValueListEditor
      // in zwei getrennten Spalten dargestellt:
```

```
        Add('DisplayName='+DisplayName);
        Add('PathName='+PathName);
        Add('Parent='+Parent.DisplayName);
        Add('Level='+IntToStr(Level));
        if fcCanCopy in Capabilities then Add('CanCopy=True');
        if fcCanDelete in Capabilities then Add('CanDelete=True');
        ...
      end;
    finally
      ValueListEditor1.Strings.EndUpdate;
    end;
  end;
```

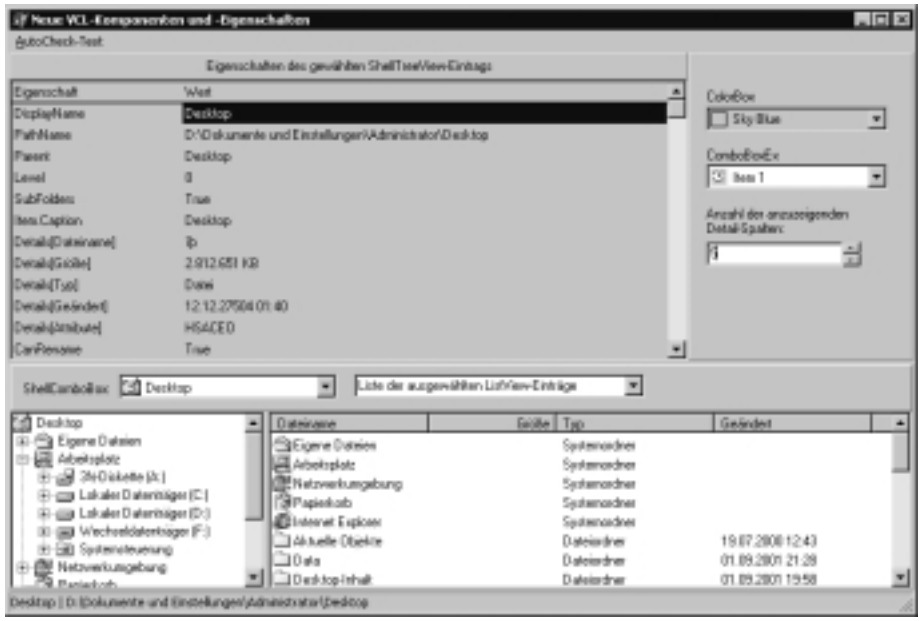


Abbildung 8.5: Eine Demonstration der neuen Komponenten TValueListEditor, TShellTreeView und TShellListView

Hinweis: Ein ValueList*Editor* kann auch zur reinen *Anzeige* von Werten verwendet werden, wenn sein Property *Options.goEditing* auf *False* gesetzt wurde. Dies ist im Beispielprogramm der Fall.

Ermitteln der Detail-Informationen

Der Windows-Explorer verfügt über eine Detail-Ansicht, in der für jedes Objekt verschiedene Spalten von Informationen angezeigt werden. Per Voreinstellung werden allerdings wesentlich weniger Spalten angezeigt, als insgesamt verfügbar sind (unter Windows 2000 finden sich etwa unter ANSICHT | SPALTEN AUSWÄHLEN... 37 verfügbare

Spalten). Zwar enthält die *TShellFolder*-Struktur der beiden Delphi-Komponenten auch ein *Details*-Property, das dem Namen nach Zugriff auf diese Detail-Informationen bieten soll, jedoch resultierte die Abfrage dieses Properties im Test des Autors immer in einer Schutzverletzung.

Dies gibt uns die Gelegenheit, die von *TShellList/TreeView* gespeicherten COM-Schnittstellen einmal direkt aufzurufen. Die Schnittstelle *IShellFolder2* steht für jeden Eintrag der beiden Komponenten im Property *ShellFolder2* zur Verfügung. Sie enthält eine Funktion namens *GetDetailsOf*, die Detail-Informationen zu den in einem Ordner enthaltenen Objekten vermittelt. Das heißt: Um die Details von *File1* zu erhalten, dürfen Sie nicht *File1.GetDetailsOf* aufrufen, sondern müssen sich an den Elternordner wenden (*TShellList/TreeView* speichern diesen Elternordner im *Parent*-Property der *TShellFolder*-Daten). *GetDetailsOf* erwartet drei Parameter:

- Die Nummer der Spalte wird im *zweiten* Parameter angegeben (erste Spalte = 0).
- Im ersten Parameter können Sie *nil* angeben, um den Titel dieser Spalte zu erhalten; wenn Sie den Inhalt der Spalte wissen möchten, geben Sie hier einen Bezeichner des Eintrages an, für den der Spalteninhalt ermittelt werden soll. Dieser »Bezeichner« muss ein Zeiger auf einen *Shell Item ID* sein, die in Kapitel 8.4.3 noch genauer erläutert werden. An dieser Stelle sind sie nicht weiter interessant, da *TShellList/TreeView* für jedes Objekt auch eine solche ID bereit hält, die wir direkt an *GetDetailsOf* weitergeben können.
- Im dritten Parameter wird eine *TShellDetails*-Struktur angegeben, in der *GetDetailsOf* das Ergebnis ablegt. Zur Interpretation dieser Struktur mit *StrRetToString* siehe auch den Hinweis unten.

Die folgende Funktion des Beispielprogramms *NewControlsTest* ermittelt je nach *Title*-Parameter Titel oder Spaltendaten eines angegebenen *ShellList/TreeView*-Eintrags. Das Beispielprogramm verwendet diese Funktion, um einige Spalteninformationen im ValueListEditor anzuzeigen (siehe Abbildung 8.5):

```
function GetDetails(Folder: TShellFolder; Index: Integer;
                    Title: Boolean): String;
var
  X: TShellDetails; // hierfür: uses ShlObj
begin
  if Assigned(Folder.Parent) and
     Assigned(Folder.Parent.ShellFolder2) then
  begin
    with Folder.Parent.ShellFolder2 do
      if Title then GetDetailsOf(nil, Index, X)
        else GetDetailsOf(Folder.RelativeID, Index, X);
    Result := StrRetToString(Folder.AbsoluteID, X.str);
  end;
end;
```

Diese Funktion wird in der oben bereits gedruckten Methode *ShellTreeView1Change* wie folgt aufgerufen:

```
Add(Format('Details[%s]=%s',
           [GetDetails(ShellTreeView1.SelectedFolder, i, True),
            GetDetails(ShellTreeView1.SelectedFolder, i, False)]));
```

Dies soll als erstes Beispiel genügen. Wir halten fest, dass bei aller Normalität, die der Aufruf

```
ShellFolder2.GetDetailsOf(nil, Index, X)
```

ausstrahlt, hier doch ein wesentlicher Unterschied zum Aufruf diverser anderer Windows-API-Funktionen wie etwa *PostMessage*, *SetMapMode* und *Shell_NotifyIcon* besteht: Die Funktion *GetDetailsOf* ist eine *Methode* und wird für ein bestimmtes Objekt aufgerufen (gerade deshalb sieht der Aufruf in Object Pascal ja so »normal« aus), wenn sich dieses Objekt auch weit außerhalb der VCL befindet.

Hinweis: Die in *GetDetails* aufgerufene Funktion *StrRetToString* ist in der Unit von *TShellTreeView* und *TShellListView* enthalten. Da sie dort jedoch als privat deklariert ist, wurde sie kurzerhand in die Formular-Unit des Beispielprogramms kopiert. Die 23 Zeilen dieser Funktion sollen Ihnen hier erspart werden, jedoch ist es bezeichnend für die Kompliziertheit der Microsoft-Schnittstelle, zu welcher Arbeit der normalerweise so simple Aufruf einer String-Funktion wird: Die *GetDetailsOf* liefert das String-Ergebnis in einer mehrfach geschachtelten Struktur. Der eigentliche String befindet sich in der *STRRET*-Struktur *TShellDetails.str*, in der der String widerum in drei verschiedenen Formaten abgelegt werden kann (diese drei Fälle zu unterscheiden und zu interpretieren ist Aufgabe der Funktion *StrRetToString*).

Eine Referenz der älteren dieser Shell-Schnittstellen finden Sie übrigens in der Windows-API-Referenz (unter Delphi 6 unter dem Menüpunkt HILFE | WINDOWS-SDK, falls installiert), für eine Dokumentation der neueren Interfaces benötigen Sie eine aktuellere Informationsquelle wie etwa die Online-Version des MSDN Library unter *msdn.microsoft.com/library*, in der Sie per Suchfunktion schnell zu einzelnen Schnittstellen wie *IShellDetails* gelangen. Ein Beispiel für die Verwendung von *IShellFolder* finden Sie im *ShellExplorer*-Beispielprogramm in Kapitel 8.4.3.

8.4.2 Programmgruppen und Datei-Verknüpfungen

Als erstes Beispiel für die Verwendung von Shell-Funktionen bzw. Klassen soll die Erzeugung von Programmgruppen und Verknüpfungen (Links) zu Programmdateien behandelt werden. Zwar können Sie Programmgruppen auch in aktuellen Windows-

Versionen noch mit der alten von Windows 3.1 eingeführten DDE-Schnittstelle erzeugen (siehe Win32-Hilfe unter *Shell Dynamic Data Exchange Interface*), jedoch ist die neue Shell-Schnittstelle erheblich flexibler.

Dies beginnt schon damit, dass Sie die Programmgruppen mit der Shell nicht nur im Start-Menü, sondern in einer Vielzahl wichtiger Systemordner erzeugen können. Abbildung 8.6 zeigt die Oberfläche des Beispielprogramms *ShellLinks*, in dessen Feld *Ziel der Gruppe/des Links* Sie eines aus fünf Zielen für einen neuen Programmordner auswählen können. Die Checkbox *für alle Benutzer* verdoppelt die Zahl der Möglichkeiten noch einmal. Ist sie gewählt, wird der Ordner beispielsweise nicht im persönlichen Programm-Menü des aktuellen Benutzers, sondern im allgemeinen Abschnitt des Programm-Menüs angelegt, das für alle Benutzer verfügbar ist.

Erzeugen von Programmgruppen *R150*

Für das Anlegen eines neuen Ordners brauchen Sie noch keine Shell-Interfaces, sondern lediglich eine API-Funktion, die Ihnen sagt, welchem Verzeichnis des Dateisystems ein bestimmtes Ziel wie das Startmenü oder der Windows-Desktop entspricht. Diese Funktion heißt *SHGetSpecialFolderLocation* und liefert einen hier nicht weiter interessanten *PItemIdList*-Zeiger, der daraufhin mit *SHGetPathFromIDList* in einen herkömmlichen Dateisystem-Pfad umgewandelt wird:

```
var
  pidl: PItemIdList;
  FolderPath: string;
begin
  if CheckCommon.Checked then // Checkbox "für alle Benutzer"
    case RadioGroup1.ItemIndex of  // Feld "Ziel der Gruppe/des Links"
      0: SystemFolder := CSIDL_STARTMENU;
      1: SystemFolder := CSIDL_PROGRAMS;
    ...
  if SUCCEEDED(SHGetSpecialFolderLocation(0, SystemFolder, pidl)) then
  begin
    // Umformung: pidl (PItemIdList) -> FolderPath (String):
    SetLength(FolderPath, max_path);
    SHGetPathFromIDList(pidl, PChar(FolderPath));
    SetLength(FolderPath, strlen(PChar(FolderPath)));
    // Die "Programmgruppe" entspricht einem normalen Verzeichnis
    // im Dateisystem, das mit MkDir erzeugt werden kann:
    FolderPath := FolderPath + '\' + Edit1.Text;
    MkDir(FolderPath);
```

MkDir, die Prozedur zum Erzeugen eines neuen Ordners, ist keine Shell-Funktion, sondern befindet sich in Delphis *System*-Unit. Mit ihr können Sie auch beliebige andere Verzeichnisse erzeugen.

Das vollständige Beispielprogramm prüft außerdem, ob der Pfad schon existiert, und macht eine Sicherheitsabfrage, falls dies der Fall ist (damit nicht der Inhalt bestehender Ordner kommentarlos überschrieben wird).

Abbildung 8.6: Erzeugen von Programmgruppen und Verknüpfungen mit dem Beispielprogramm ShellLinks

Erzeugen eines Links *R151*

Um eine Verknüpfung auf eine Datei zu erzeugen, benötigen Sie wieder ein COM-Objekt, und zwar eines der Klasse *CLSID_ShellLink*, auf das Sie zunächst über ein Interface des Typs *IShellLinkA* und dann (zum Schreiben des Links in eine Datei) über ein Interface des Typs *IPersistFile* zugreifen. Das erste Interface wird im Beispielprogramm unter dem Namen *ShellLink* geführt und durch einen Aufruf von *CoCreateInstance* mitsamt dem zugehörigen Shell-Objekt erzeugt. Das zweite Interface trägt im Folgenden den Namen *LinkFile* und wird vom bereits existierenden Shell-Objekt mit der *QueryInterface*-Methode abgefragt:

```
var
  ShellLink: IShellLink;
  LinkFile: IPersistFile;
  widestr: PWideChar;
begin
  if SUCCEEDED(CoCreateInstance(CLSID_ShellLink, nil,
    CLSCTX_INPROC_SERVER, IID_IShellLinkA, ShellLink)) then
  begin
    // Die Verknüpfung soll auf den folgenden Pfad verweisen:
    ShellLink.SetPath(PChar(Memo1.Lines[i]));
    // Die (in der Shell nicht angezeigte) Beschreibung des Links setzen:
    ShellLink.SetDescription(PChar('Test-Link zu '+Memo1.Lines[i]));
    // Setzen: LinkFile (Interface-Variable):
    if SUCCEEDED(ShellLink.QueryInterface(IID_IPersistFile, LinkFile))
    then begin
      GetMem(widestr, max_path*2);
      try // FileName gibt den Namen der Verknüpfungsdatei an.
```

```
          (* FileName wird aus dem eventuell neu erzeugten Ordner, aus
             dem Dateinamen der verknüpften Datei (im Memo-Feld des
             Beispielprogramms) und aus der Endung .LNK gebildet: *)
          FileName := ExtractFileName(Memo1.Lines[i]);
          FileName := ChangeFileExt(FileName, '.LNK');
          FileName := FolderPath + '\' + FileName;
          // Umformung: FileName -> widestr
          MultiByteToWideChar(CP_ACP,0, PChar(FileName),-1,
                              widestr,max_path);
          LinkFile.Save(widestr, true);
        finally
          FreeMem(widestr, max_path*2);
        end;
      end; // Succeeded(ShellLink.QueryInterface)
    end; // Succeeded(CoCreateInstance)
  end;
```

Zu weiteren, die Interfaces *IShellLink* und *IPersistFile* betreffenden Details muss hier auf die Online-Hilfe zum Win32-SDK verwiesen werden.

8.4.3 Ein selbst gemachter Shell-Browser

Mit der Anwendung *ShellExplorer* können Sie den gesamten Baum der Shell-Objekte untersuchen. Im Gegensatz zum Windows-Explorer können Sie in diesem sogar in der Baumansicht zwischen kleinen und großen Icons wechseln. Dafür enthält er jedoch keinerlei Funktionen zur Manipulation der Dateien wie beispielsweise Kopieren oder Umbenennen. Lediglich einige Sortierfunktionen sind implementiert.

ShellExplorer nutzt die in Kapitel 3.6.2 und 3.6.3 behandelten Komponenten *ListView* und *TreeView* und macht intensiven Gebrauch von den API-Funktionen und COM-Objekten der Windows-Shell. Mit diesen COM-Objekten sowie einigen Shell-Grundlagen müssen wir uns als Erstes befassen. Bei der danach folgenden Beschreibung des Beispielprogramms konzentriert sich dieses Kapitel auf die Erläuterung der Gesamtstruktur des Programms und auf seine wichtigen und grundlegenden Funktionen. Zusätzliche, nicht wesentliche Informationen über die zahlreichen verwendeten Shell-Funktionen können in Delphis Online-Referenz nachgeschlagen werden.

Benennung der Shell-Objekte

Zuerst müssen wir uns mit der Art beschäftigen, wie die Windows-Shell ihre Ordner und Verzeichnisse benennt. Für Dateien der Festplatte gibt es zwar schon zwei Benennungsschemata (Pfade mit langen und mit kurzen Dateinamen), wie Sie bei einem Blick in den Explorer schnell erkennen können, ist die Verzeichnisstruktur des lokalen Computersystems jedoch nicht das Einzige, was in der Hierarchie des Explorers dargestellt wird: Alle lokalen Laufwerke sind mit anderen Objekten wie der Systemsteuerung und dem Drucker-Ordner zu einem *Arbeitsplatz* zusammengefasst.

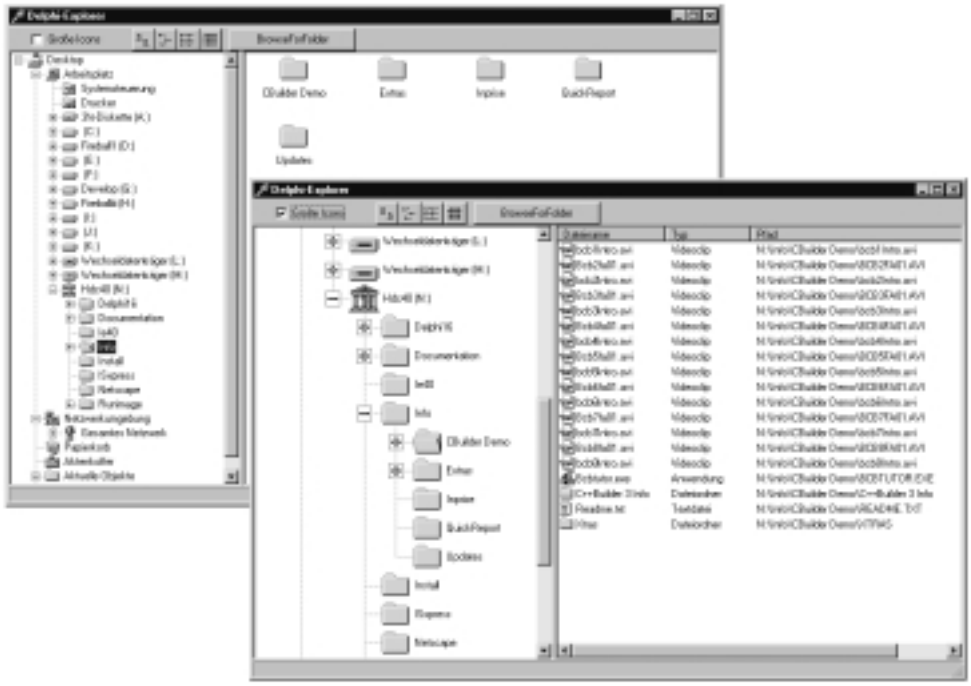


Abbildung 8.7: Der in Delphi programmierte Shell-Explorer kann auch als Grundlage für bessere Dateiauswahl-Dialoge und andere Browser-Anwendungen dienen.

Dieser ist wiederum Teil des *Desktop*-Objekts, zu dem auch die Netzwerkumgebung und die Ordner, die Sie auf Ihrem Windows-Desktop abgelegt haben, gehören.

Abgesehen davon, dass diese auf dem Desktop abgelegten Ordner sich auch in der Verzeichnisstruktur der Festplatte wiederfinden (im Verzeichnis WINDOWS\DESKTOP bzw. WINNT\PROFILES\[PROFILE-UNTERORDNER]\DESKTOP), liegen diese Objekte außerhalb der durch Verzeichnispfade benennbaren Gebiete. Schon der Arbeitsplatz lässt sich nicht durch einen Verzeichnispfad angeben, da es im Dateisystem kein Überverzeichnis gibt, das alle Wurzelverzeichnisse der einzelnen Laufwerke zusammenfasst.

Aus diesem Grund führt die Windows-Shell ein neues Benennungsschema ein, das dem der Dateipfade teilweise sehr ähnlich ist. Ähnlich ist der Aufbau eines Pfades durch Aneinanderreihung mehrerer Einzelbezeichnungen. Wenn also im Dateisystem eine Datei A im Verzeichnis B des Laufwerks C liegt, können Sie sie durch den Pfad *C:\B\A* eindeutig identifizieren. Die Pfade der Shell-Objekte werden ebenfalls durch Hintereinanderschreiben aufgebaut. Um beispielsweise den vollständigen Pfad des *Arbeitsplatz*-Ordners innerhalb des *Desktop*-Ordners anzugeben, schreiben Sie die *Shell Item Identifiers* dieser beiden Objekte einfach hintereinander (begonnen mit dem des *Desktop*-Ordners).

Shell-Objekt-Bezeichner und Bezeichnerlisten

Der große Unterschied zu den Dateipfaden besteht darin, dass die Shell-Objekte nicht durch lesbare Namen bezeichnet werden, sondern durch binäre *Shell Item Identifiers* (im Folgenden auch *Objektbezeichner* genannt), die in der Struktur *TSHItemId* teilweise definiert werden. Nur *teilweise* deshalb, weil *TSHItemId* lediglich die ersten beiden Bytes dieser Struktur festlegt; diese Bytes geben an, wie viele Bytes die gesamte Struktur verbraucht. Wie die restlichen Bytes verwendet werden, ist alleine Sache der Shell bzw. des Objekts, das den jeweiligen Ordner implementiert.

Es gibt natürlich eine Möglichkeit, einen lesbaren Namen zu einem solchen binären Bezeichner herauszufinden; da es für die Funktion des Programms jedoch erheblich wichtiger ist, richtig mit den binären Objektbezeichnern umzugehen, werden wir erst später auf die lesbaren Namen zurückkommen.

Im Programm tauchen die Objektbezeichner meistens in Form von *PItemIdList*-Variablen auf. Wie der Name schon andeutet, handelt es sich hierbei um Zeiger, und zwar um Zeiger auf eine Liste von Objektbezeichnern. Eine *PItemIdList*-Variable kann auf zwei Arten interpretiert werden:

- Wenn es sich um eine vollständige Pfadangabe wie *Desktop\Arbeitsplatz\C:* handelt, werden die Bezeichner der einzelnen Ordner einfach hintereinander geschrieben. In diesem Beispiel zeigt die Variable also auf eine Folge dreier *TSHItemId*-Strukturen, von denen die erste den *Desktop*-Ordner angibt, die zweite den *Arbeitsplatz*-Ordner, die dritte das Laufwerk C:. Da, wie schon erwähnt, die Länge jeder einzelnen *TSHItemId*-Struktur variabel ist, ist diese Liste nicht besonders einfach zu durchschauen. Das Ende der Liste wird durch zwei Null-Bytes markiert.
- Meistens handelt es sich nicht um vollständige Pfade, sondern um einzelne Bezeichner, die innerhalb eines bestimmten Verzeichnisses liegen. Eine *PItemIdList*-Variable zeigt, anders als es der Name vermuten lässt, meistens auf einen solchen Einzelbezeichner. Damit ein solcher Bezeichner einen Sinn ergibt, benötigen Sie einen Pfad als Bezugspunkt. Meistens handelt es sich bei diesem Pfad um ein Ordner-Objekt, dessen Methoden einen solchen Einzelbezeichner verstehen und erwarten.

Die IShellFolder-Schnittstelle

Die Ordner der Windows-Shell sind als Windows-Objekte nach dem Component Object Model realisiert. Wie schon beschrieben, können Sie die Funktionen eines solchen Objekts nicht über ein einzelnes Object-Pascal-Objekt, sondern nur über einzelne Schnittstellen ansprechen, zu denen mindestens die *IUnknown*-Schnittstelle gehört. Die wichtigsten Schnittstellen, die von den Shell-Ordnern unterstützt werden, sind *IShellFolder* und *IEnumIdList*. Da Sie die Shell durch eigene Ordner-Typen erweitern können,

ist die Zahl der durch Ordner unterstützten Schnittstellen nicht begrenzt. Wichtige Methoden der Standard-Schnittstelle *IShellFolder* sind in der folgenden Tabelle zusammengefasst:

Methode	Aufgabe
GetDisplayNameOf	erwartet im ersten Parameter einen *PItemIdList* für ein in diesem Ordner enthaltenes Objekt und ermittelt den zu diesem gehörenden lesbaren Namen (*Anzeigenamen*); die Rückgabe erfolgt über eine etwas kompliziertere Struktur (zur Verwendung siehe den Quelltext des Beispielprogramms).
ParseDisplayName	wandelt umgekehrt zu *GetDisplayNameOf* einen Anzeigenamen in eine *ItemIdList*.
GetAttributesOf	überprüft zahlreiche Flags des als Parameter übergebenen Unterobjekts; so besagt z.B. *SFGAO_FOLDER*, dass ein Unterobjekt wieder ein Ordner ist, oder *SFGAO_CANCOPY*, dass das Objekt kopiert werden kann.
GetUIObjectOf	ist das Tor zu weiteren Schnittstellen eines in dem Ordner enthaltenen Objekts, z.B. für Kontextmenüs oder zur Übertragung von Daten des Objekts.
EnumObjects	liefert eine andere Schnittstelle des Ordners: die *IEnumIdList*-Schnittstelle, über die Sie sich alle in dem Ordner enthaltenen Objekte aufzählen lassen können.
BindToObject	liefert eine *IShellFolder*-Schnittstelle zu einem Unterordner, den Sie über den *PItemIdList*-Parameter angeben.

Auch wenn die Tabelle die einzelnen Parameter der Methoden zur Vereinfachung weglässt, lässt sich eine Gemeinsamkeit erkennen:

- Alle Methoden bis auf *EnumObjects* arbeiten mit den Objekten, die im Ordner enthalten sind. Diese Methoden erwarten einen *PItemIdList*-Parameter, der das Objekt innerhalb des Ordners eindeutig bezeichnet.
- Einige Methoden dienen lediglich dazu, weitere Schnittstellen herauszufinden, sei es die *IEnumIdList*-Schnittstelle des Ordnerobjekts oder die Schnittstellen der im Ordner enthaltenen Objekte (*GetUIObjectOf*, *BindToObject*).

Nun benötigen wir nur noch Wege, an eine solche *IFolder*-Schnittstelle heranzukommen. Im Beispielprogramm werden wir zwei dieser Wege anwenden: Zugriff auf das Objekt für den *Desktop*-Ordner erhalten wir über die schon erwähnte API-Funktion *SHGetDesktopFolder*, alle anderen Objekte erhalten wir jeweils als Unterobjekte eines bestehenden Ordners. Diese werden von der im Folgenden beschriebenen *IEnumIDList*-Schnittstelle aufgezählt.

Aufzählen der Shell-Objekte

Um alle Objekte der Shell aufzählen zu lassen, benötigen Sie lediglich zwei Typen von Schnittstellen. Bevor das Beispielprogramm dies demonstriert, sind hier die grundlegenden Schritte aufgeführt:

- Holen Sie sich mit der Funktion *SHGetDesktopFolder* die *IShellFolder*-Schnittstelle des Desktop-Ordners. Über diese können Sie den Desktop-Ordner als Windows-Objekt ansprechen.
- *IShellFolder* besitzt die bereits erwähnte Methode *EnumObjects*, die Ihnen eine zweite Schnittstelle des *Desktop*-Ordners zurückliefert: die *IEnumIdList*-Schnittstelle.
- Diese Schnittstelle enthält die Methode *Next*, die Ihnen bei jedem Aufruf ein weiteres Shell-Objekt liefert, das sich im *Desktop*-Ordner befindet.
- Falls dieses Objekt ein Ordner ist (was Sie über die Methode *GetAttributesOf* herausfinden können), rufen Sie die Methode *BindToObject* der *IShellFolder*-Schnittstelle des Desktop-Ordners auf, um eine *IShellFolder*-Schnittstelle für diesen Unterordner zu erhalten. Für diesen Unterordner können Sie rekursiv die Schritte zwei bis vier wiederholen, statt mit dem *Desktop*-Ordner arbeiten Sie dann mit dem Unterordner.

Speicherverwaltung

Bei der Verwendung der Shell-Objekte müssen Sie sich teilweise auch um die Speicherverwaltung kümmern, denn nicht alle Daten, für die die Shell Speicher reserviert, liegen in Form von COM-Objekten vor und können automatisch freigegeben werden. Für Daten, die keine Objekte sind, verwendet die Shell die Speicherverwaltungs-Schnittstelle *IMalloc*, die ein direkter Nachkomme von *IUnknown* ist. Um derartige Objekte freizugeben, müssen auch Sie sich dieser Schnittstelle bedienen.

Für den Shell-Explorer des nächsten Kapitels benötigen wir diese Methoden nur im Zusammenhang mit den oben genannten *Shell Item Identifier Lists*. Einige der Shell-Methoden geben solche Listen zurück und erwarten, dass Sie den von diesen belegten Speicher wieder freigeben. Da sich dieser Speicher wie die Shell-Objekte außerhalb des Autoritätsbereichs der Object Pacal-Laufzeitbibliothek befindet, können Sie deren Funktionen *System.Release* und *System.FreeMem* hier nicht verwenden; statt dessen rufen Sie die Methode *Free* der *IMalloc*-Schnittstelle auf.

Zugriff auf die *IMalloc*-Schnittstelle bzw. eine Variable des Typs *IMalloc* erhalten Sie wieder über eine einfache API-Funktion: *SHGetMalloc*. Die *OnCreate*-Methode des Formulars für das Beispielprogramms zeigt, wie es genau geht.

Verknüpfung der Baumeinträge mit weiteren Daten

Für den *ShellExplorer* genügt es nicht, die Shell-Schnittstellen einmal abzufragen und dabei eine Baumstruktur aufzubauen, vielmehr ist es während des Programmablaufs ständig notwendig, erneut auf die Shell-Schnittstellen zuzugreifen (zwar bietet dieses Beispielprogramm keinerlei Funktionen an, die Objekte zu verändern, also z.B.

Dateien zu kopieren, der erneute Zugriff auf Shell-Schnittstellen ist jedoch schon beim Expandieren des Baumes erforderlich).

Es bietet sich daher an, die Schnittstellen der einzelnen Ordner-Objekte zusammen mit den *TreeView*-Einträgen zu speichern. Wir werden in diesem Beispielprogramm also die Möglichkeit der Klasse *TTreeView* nutzen, mit jedem Eintrag einen Zeiger zu speichern, der auf einen selbst definierten Datenblock zeigt.

Bei der Entwicklung von *ShellExplorer* hat es sich als vorteilhaft erwiesen, mit jedem TreeView-Eintrag nicht nur das Shell-Objekt, sondern die im Folgenden definierte Struktur zu speichern:

```
type
  PNodeInfo = ^TNodeInfo;
  TNodeInfo = record
    RelativeIDL: PItemIdList;  // ein einzelner Shell-Objektbezeichner
    AbsoluteIDL: PItemIdList;  // komplette Pfadangabe des Bezeichners
    TreeNode: TTreeNode;       // der verknüpfte TreeView-Eintrag
    // falls der Eintrag ein Ordner ist:
    ShellFolder: IShellFolder; // IShellFolder-Schnittstelle des Ordners
    Expanded: Boolean;         // gibt an, ob die Unterobjekte des Ordners
                               // schon bekannt sind
  end;
```

Auf den Sinn der einzelnen Strukturelemente kommen wir bei Gelegenheit zurück, zunächst einmal gilt es, weitere Vorbereitungen für den Programmlauf zu treffen.

Die System-ImageList

Wie Sie in Abbildung 8.7 sehen können, soll auch unser *ShellExplorer* all die schönen Icons anzeigen, die der Original-Explorer verwendet. Da diese Icons eine Eigenschaft der einzelnen Shell-Objekte sind (und nicht etwa Privatbesitz des Windows-Explorers), ist es leicht, Zugriff auf sie zu erhalten.

Wie in Kapitel 3.6.2 beschrieben, benötigt ein ListView eine Liste für die kleinen und eine für die großen Icons, ein TreeView kommt mit einer solchen Bilderliste aus. Die Icons aller Shell-Objekte sind gemeinsam in besonderen Bilderlisten, den System-Bilderlisten, versammelt, die sich in jeder normalen Windows-95/98/NT-Umgebung versteckt halten. Die Shell erlaubt jeder Anwendung, auf diese Listen zuzugreifen. Sie benötigen dazu lediglich ein Handle auf diese Bilderlisten, die Sie als Nebeneffekt von der Funktion *SHGetFileInfo* erhalten. Die Bild-Indizes, die dieselbe Funktion für die einzelnen Objekte zurückliefert, verstehen sich als Indizes in diese Systembilderlisten.

Da die Komponenten *TTreeView* und *TListView* jedoch in ihren Properties *Images*, *SmallImages* usw. nur *TImageList*-Komponenten zulassen, müssen Sie die System-Bilderlisten an der VCL vorbei in Ihre List- und TreeView-Komponenten »schmuggeln«, und zwar über den Aufruf von API-Funktionen, wie der nächste Abschnitt zeigen wird.

Initialisierung des Explorers

Wir können nun zur Initialisierung des Beispielprogramms und damit zur Methode für das *OnCreate*-Ereignis des Formulars kommen. Diese initialisiert im Wesentlichen die Bilderlisten, verknüpft sie mit List- und TreeView und setzt den obersten Eintrag des TreeViews, der den Desktop-Ordner repräsentiert. Sie verwendet viele Funktionen der Shell- und CommCtrl-APIs, die im Anschluss an die Methode in einer Tabelle kurz beschrieben sind.

```
procedure TExplorer.FormCreate(Sender: TObject);
var
  FileInfo: TSHFileInfo;
  DesktopItemIdList: PItemIdList;
  Ptr: pointer;
begin
  if Succeeded(SHGetMalloc(Allocator)) and
     Succeeded(SHGetDesktopFolder(Desktop)) then
  begin
    // Feststellen der System-Bilderlisten und des Desktop-Icons:
    SHGetSpecialFolderLocation(Handle, CSIDL_DESKTOP, DesktopItemIdList);
    SysImageListSmall := SHGetFileInfo(PChar(DesktopItemIdList), 0,
                    FileInfo, sizeof(FileInfo),
                    SHGFI_PIDL or SHGFI_SYSICONINDEX or SHGFI_SMALLICON);
    SysImageListLarge := SHGetFileInfo(PChar(DesktopItemIdList), 0,
                    FileInfo, sizeof(FileInfo),
                    SHGFI_PIDL or SHGFI_SYSICONINDEX);
    // Verbinden der Views mit den Systembilderlisten und
    // Holen der Bild-Indizes für das Symbol des Desktop-Ordners:
    TreeView_SetImageList(Tree.Handle, SysImageListSmall, TVSIL_NORMAL);
    ListView_SetImageList(ListView1.Handle, SysImageListLarge,
                          LVSIL_NORMAL);
    ListView_SetImageList(ListView1.Handle, SysImageListSmall,
                          LVSIL_NORMAL or LVSIL_SMALL);
    // Wurzel-Element setzen:
    Tree.Items[0].ImageIndex := FileInfo.iIcon; // Desktop-Icon-Index
    Tree.Items[0].SelectedIndex := FileInfo.iIcon;
    Tree.Items[0].Data := MakeNodeInfo(nil, Desktop, DesktopItemIdList);
    Tree.Items[0].HasChildren := True;
  end
  else // Erzeugen einer selbst definieren Ausnahmebedingung:
    raise EShellInitError.Create('Shell ist nicht erreichbar');
end;
```

Bezeichner	Erläuterung
Succeeded	prüft anhand des relativ komplexen *HResult*-Rückgabewertes, den viele der hier verwendeten Funktionen liefern, ob die Funktion erfolgreich ausgeführt wurde.

Bezeichner	Erläuterung
SHGetMalloc	setzt die als Parameter übergebenen *IMalloc*-Variable auf die entsprechende Schnittstelle der Windows-Shell.
SHGetDesktopFolder	liefert eine *IShellFolder*-Schnittstelle für den *Desktop*-Ordner.
SHGetSpecialFolderLocation	liefert *PItemIdList*-Bezeichner für besondere Objekte, in diesem Fall für den *Desktop*-Ordner.
SHGetFileInfo	wird später erläutert.
TreeView_SetImageList	setzt die Bilderliste eins TreeViews. Sie benötigen hierzu das Handle des TreeViews, das in der Komponente *TTreeView* (wie auch in allen anderen *TWinControl*-Komponenten) im Property *Handle* vorliegt.
ListView_SetImageList	wie *TreeView_SetImageList*, kann aber die Liste der kleinen oder großen Bilder eines ListViews setzen.
MakeNodeInfo	Funktion des Beispielprogramms, erzeugt die mit jedem *TTreeNode*-Eintrag verknüpfte Datenstruktur (siehe unten).

SHGetFileInfo

Die komplexeste der genannten Funktionen ist *SHGetFileInfo*, sie speichert zahlreiche Informationen über ein Shell-Objekt in einer *TSHFileInfo*-Struktur, von der Sie eine Variable anlegen und als Parameter übergeben müssen. Über welches Objekt Sie Informationen wünschen, teilen Sie der Funktion über einen *PItemIdList*-Zeiger oder über einen String mit. Die Arbeitsweise der Funktion steuern Sie im letzten Parameter über eine Verknüpfung verschiedener der folgenden Flags.

Flag	Wirkung
SHGFI_PIDL	Das abgefragte Objekt wird nicht als String, sondern als *PItemIdList*-Zeiger angegeben.
SHGFI_SYSICONINDEX	Die Funktion liefert das Handle der Systembilderliste als Ergebnis zurück.
SHGFI_SMALLICON	weist die Funktion in diesem Fall an, das Handle der Systembilderliste für die kleinen Icons zurückzuliefern.
SHGFI_OPENICON	dient zum Abruf des *SelectedImage* für einen TreeView-Eintrag.

Expandieren nach Bedarf

Wie schon in Kapitel 3.6.3 erwähnt, sind viele TreeView-Strukturen so komplex oder aufwändig zu erstellen, dass sie nicht schon zu Beginn vollständig aufgebaut werden sollten, sondern nur nach Bedarf, wenn der Benutzer einen Eintrag expandiert. Während der Windows-Explorer von sich aus zumindest alle Einträge expandiert, die zu einem bestimmten Pfad führen, ist das Beispielprogramm sogar so sparsam mit der Rechenzeit, dass es am Anfang nur die Wurzel des Baumes, den Desktop-Ordner, darstellt.

Beim *OnCreate*-Ereignis wurde bereits das *HasChildren*-Property dieses Wurzeleintrags auf *True* gesetzt. Das bewirkt, dass ein Plus-Symbol neben dem Eintrag angezeigt wird, obwohl noch gar keine Untereinträge im TreeView gespeichert sind. Klickt der Benutzer auf dieses Symbol, erhält das Programm mit dem *OnExpanding*-Ereignis die Chance, die Untereinträge hinzuzufügen. Welcher Eintrag expandiert werden soll, besagt der Parameter *Node* dieses Ereignisses.

Die folgende Methode bearbeitet dieses Ereignis; sie holt sich über das Property *Data* des *Node*-Eintrags die Datenstruktur, die bei der Erstellung des Eintrages mit diesem verknüpft wurde, und gibt den Expandierungs-Auftrag an die Methode *EnumFolder* weiter:

```
procedure TExplorer.TreeExpanding(Sender: TObject; Node: TTreeNode;
  var AllowExpansion: Boolean);
var
  NodeInfo: PNodeInfo;
begin
  NodeInfo := PNodeInfo(Node.Data);
  if Assigned(NodeInfo) and not NodeInfo.Expanded then begin
    EnumFolder(Node);
    NodeInfo.Expanded := True;
  end;
end;
```

EnumFolder beauftragt die *IShellFolder*-Schnittstelle des Ordners (diese ist Teil der mit dem Eintrag verknüpften Datenstruktur), die im Ordner enthaltenen Objekte aufzuzählen, wie oben im Abschnitt *Aufzählen der Shell-Objekte* beschrieben. Über den Parameter *SHCONTF_FOLDERS* erreicht sie, dass nur die Unterordner und nicht alle Objekte aufgezählt werden:

```
procedure TExplorer.EnumFolder(Node: TTreeNode);
var
  ShellFolder: IShellFolder;
  Objects: IEnumIdList;
  ItemIdList: PItemIdList;
  DummyResult: ULONG;
begin
  ShellFolder := PNodeInfo(Node.Data)^.ShellFolder;
  if Succeeded(ShellFolder.EnumObjects(Handle, SHCONTF_FOLDERS, Objects))
  then begin
    while Objects.Next(1, ItemIdList, DummyResult) = NOERROR do
      AddSubfolderToNode(Node, ItemIdList);
  end;
end;
```

Die Methode *Next* der *IEnumIdList*-Schnittstelle liefert noch keine *IShellFolder*-Schnittstelle auf den Unterordner, sondern lediglich einen *PItemIdList*-Bezeichner für den Ordner. Mit Hilfe dieses Bezeichners einen neuen Knoten für die Baumansicht zu gene-

rieren ist Aufgabe der noch zu erläuternden Methode *AddSubfolderToNode*; sie wird oben für jeden Unterordner einmal aufgerufen.

Erzeugen der TreeView-Einträge

AddSubfolderToNode muss nun einige Informationen über den neuen Unterordner sammeln. Die meisten davon werden dafür verwendet, die Properties des *TreeView*-Eintrags richtig einzustellen:

- der darstellbare Name des Ordners und die mit dem Eintrag verknüpfte selbst definierte *NodeInfo*-Struktur (beides wird schon bei der Erzeugung des Eintrags an die Methode *TTreeNodes.AddChildObject* übergeben),
- ein Flag, das angibt, ob der Ordner Unterordner hat und somit expandierbar ist (Property *HasChildren*) sowie
- zwei Icon-Indizes für die Properties *ImageIndex* und *SelectedIndex*.

Außerdem müssen Felder der an *AddChildObject* übergebenen *NodeInfo*-Struktur initialisiert werden:

- *RelativeIDL* und *TreeNode* müssen lediglich auf die Parametern der Methode *AddSubfolderToNode* gesetzt werden.
- Der Datenblock, auf den *AbsoluteIDL* zeigt, enthält im Gegensatz zu *RelativeIDL* eine vollständige Pfadangabe des Bezeichners, beginnt also mit der *TShItemId*-Struktur des *Desktop*-Objekts. Diese vollständige Pfadangabe ist zum Aufruf der Funktion *SHGetFileInfo* erforderlich (die Methoden der *IShellFolder*-Schnittstelle benötigen lediglich den Bezeichner des Objekts innerhalb des Ordners).
- *Expanded* wird bei der Erzeugung des Eintrags grundsätzlich auf *False* gesetzt, da der Eintrag erst auf Benutzeranfrage expandiert wird (bei einem erneuten *OnExpanding*-Ereignis).
- Schließlich bleibt noch das Feld *ShellFolder* für die *IShellFolder*-Schnittstelle des Ordners. Es wird mit *IShellFolder.BindToObject* durch den übergeordneten Ordner initialisiert.

Den größten Teil der Initialisierung von *NodeInfo* übernimmt die Funktion *MakeNodeInfo*, eine weitere Funktion des Beispielprogramms. Diese und zwei weitere noch nicht gezeigte Funktionen des Beispielprogramms sind im folgenden Listing kursiv dargestellt, um sie von den API-Funktionen, Shell- und VCL-Methoden deutlich zu unterscheiden:

```
procedure TExplorer.AddSubfolderToNode(
  Node: TTreeNode; // Eintrag des TreeViews, der erweitert wird.
  ItemIdList: PItemIdList); // Ordner, um den der Node erweitert wird
```

```
var
  ShellFolder: IShellFolder;
  DisplayName: string;
  TreeNode: TTreeNode;
  NodeInfo: PNodeInfo;
  FileInfo: TSHFileInfo;
begin
  ShellFolder := PNodeInfo(Node.Data)^.ShellFolder;
  // Die mit TTreeNode verknüpfte NodeInfo-Struktur erzeugen:
  NodeInfo := MakeNodeInfo(Node, nil, ItemIdList);
  // 1. Information: lesbarer Name:
  DisplayName := GetShellItemName(ShellFolder, ItemIdList);
  // mit DisplayName und NodeInfo einen neuen TTreeView-Eintrag erstellen:
  TreeNode := Node.Owner.AddChildObject(Node, DisplayName, NodeInfo);
  // 2. Information: sind Unterobjekte vorhanden?
  TreeNode.HasChildren := HasSubfolders(ShellFolder, ItemIdList);
  // 3. Information: IShellFolder-Schnittstelle des Unterordners
  // zur späteren Verwendung beim Expandieren in NodeInfo speichern:
  ShellFolder.BindToObject(ItemIdList, nil, IID_ISHELLFOLDER,
                           pointer(NodeInfo.ShellFolder));
  // 4. Information: Index des Icons
  SHGetFileInfo(PChar(NodeInfo^.AbsoluteIDL), 0, FileInfo,
                sizeof(FileInfo), SHGFI_PIDL or SHGFI_SYSICONINDEX);
  TreeNode.ImageIndex := FileInfo.iIcon;
  // 5. Information: Icon für den Eintrag in ausgewähltem Zustand:
  SHGetFileInfo(PChar(NodeInfo^.AbsoluteIDL), 0, FileInfo,
                sizeof(FileInfo),
                SHGFI_PIDL or SHGFI_OPENICON);
  TreeNode.SelectedIndex := FileInfo.iIcon;
end;
```

SHGetFileInfo und vollständige Pfade

Da wir sowieso nicht alle Details des Beispielprogramms untersuchen können, werden die Hilfsfunktionen *MakeNodeInfo*, *GetShellItemName* und *HasSubFolders* hier nicht vollständig abgedruckt.

Besprochen werden soll noch die schwierigste Aufgabe von *MakeNodeInfo*; sie besteht darin, die vollständige Pfadangabe *AbsoluteIDL* zu konstruieren. *MakeNodeInfo* braucht dazu zwar lediglich den Bezeichner des neuen Objekts (im Folgenden die *PItemIdList*-Variable *Child*) an die vollständige Pfadangabe des übergeordneten Eintrags (hier *ParentPath*, ebenfalls vom Typ *PItemIdList*) anzuhängen, muss dafür aber einen neuen Speicherbereich verwenden, den sie von der *IMalloc-Schnittstelle* der Shell erhält (zur Initialisierung dieser Schnittstelle in der Variablen *Allocator* siehe die oben gezeigte Methode *FormCreate*):

```
if Assigned(ParentFolder) then PathLen := GetPathLen(ParentPath)
  else PathLen := 0;
Result^.AbsoluteIDL := Allocator.Alloc(PathLen+Child^.mkId.cb+2);
```

Die Ermittlung der Länge einer Shell-Pfadangabe ist in *GetPathLen*, einer weiteren Hilfsfunktion, ausgelagert; der von dieser Funktion ermittelten Länge wird in den obigen Zeilen nur noch die Länge des anzuhängenden Objektbezeichners und zwei Bytes für die abschließenden Null-Zeichen hinzuaddiert.

Dies soll als exemplarischer Einblick in die unerquicklichen Verwaltungsaufgaben von *MakeNodeInfo* genügen. Die Freigabe des mit *Alloc* reservierten Speichers wird unten in dem Abschnitt *Aufräumen des Baums der Shell-Objekte* gezeigt.

Die ListView-Komponente

Grundsätzlich sind für die ListView-Komponente ähnliche Schritte erforderlich wie für die Baumdarstellung. Jedes Mal wenn der Benutzer einen anderen Baumeintrag selektiert (Ereignis *TTreeView.OnChange*), muss das ListView mit den Unterobjekten des selektierten Ordners gefüllt werden. Die Methode für dieses Ereignis erhält den selektierten Baumeintrag im Parameter *Node* und kann sich aus der verknüpften *NodeInfo*-Struktur die *IShellFolder*-Schnittstelle holen, die beim Aufbauen des Baums (Methode *AddSubFolderToNode*, Aufruf von *BindToObject*) dort gespeichert wurde:

```
procedure TExplorer.TreeChange(Sender: TObject; Node: TTreeNode);
var
  NodeInfo: PNodeInfo;
begin
  NodeInfo := PNodeInfo(Node.Data);
  if Assigned(NodeInfo) then
    if Assigned(NodeInfo.ShellFolder) then
      FillListView(NodeInfo.ShellFolder, Node);
end;
```

Von der *FillListView*-Methode soll hier ein Ausschnitt genügen:

```
ListView1.Items.BeginUpdate;
ListView1.Items.Clear;
try
  if ShellParentOfItem.EnumObjects(Handle, SHCONTF_FOLDERS or
    SHCONTF_NONFOLDERS or SHCONTF_INCLUDEHIDDEN, Objects)=NOERROR
  then begin
    while Objects.Next(1, ItemIdList, DummyResult)=NOERROR do begin
       ...
    end;
  end;
finally
  ListView1.Items.EndUpdate;
end;
```

Die oben gezeigten Zeilen weisen im Vergleich zum Expandieren des Baums drei Besonderheiten auf:

- Das ListView wird jedes Mal vollständig neu aufgebaut, muss also am Anfang mit *Ites.Clear* gelöscht werden.
- Die Anzeige am Bildschirm soll nicht nach jedem hinzugefügten Eintrag erneuert werden, sondern erst, nachdem alle Einträge hinzugefügt wurden. Für die Dauer des Einfügens wird daher die Aktualisierung mit *Items.BeginUpdate* ausgeschaltet; der zugehörige Aufruf von *Items.EndUpdate* soll auch im Falle einer Exception stattfinden, weshalb er im *finally*-Teil eines *try*-Blocks erwähnt wird.
- Während der *EnumObjects*-Aufruf beim Expandieren des Baums nur die Unterordner auflisten sollte, sind hier alle Unterobjekte erwünscht; die beiden Konstanten *SHContf_NonFolders* (Nicht-Ordner) und *SHContf_IncludeHidden* (versteckte Objekte) teilen dies der Methode *EnumObjects* mit.

Innerhalb der *while*-Schleife werden auf ähnliche Weise Informationen über das per *Next*-Aufruf erhaltene Objekt eingeholt wie in der Methode *AddSubfolderToNode*. Kleine Unterschiede ergeben sich z.B. daraus, dass die Einträge des ListViews kein *SelectedIndex*-Property haben und dass das Beispielprogramm in der Spalte *Typ* den Typ des Objekts aufführt (der ebenfalls von *SHGetFileInfo* geliefert wird, siehe CD).

Sortieren

Beide Hälften des *ShellExplorer*-Fensters verfügen über ein eigenes Popup-Menü, das Menüpunkte zum Sortieren der Einträge beinhaltet. Zunächst folgt eine Zusammenfassung der Sortierfunktionen von *TListView* und *TTreeView*; beide Komponenten verfügen jeweils über eine automatische und eine explizit aufrufbare Sortierung:

- Die automatische Sortierung sortiert per Voreinstellung alphabetisch nach der Beschriftung der Einträge; Sie können jedoch auch das Ereignis *OnCompare* mit einer Vergleichsmethode verknüpfen, die zwei Einträge nach beliebigen anderen Kriterien vergleicht. Die automatische Sortierung verläuft dann nach diesem Kriterium. Automatische Sortierung findet je nach der Einstellung des Properties *SortType* nie (*stNone*), bei Änderung des Eintrags-Textes (*stText*), bei Änderung der mit dem Eintrag verknüpften Daten (*stData*) oder bei beidem (*stBoth*) statt. Sie können diese automatische Sortierung über die Methode *AlphaSort* explizit aufrufen.
- Neben der automatischen Sortierung können Sie mit der Methode *CustomSort* nach weiteren Kriterien sortieren. Sie übergeben der Methode eine Vergleichsfunktion ähnlich der Methode, die mit dem *OnCompare*-Ereignis verknüpft ist.

Im Beispielprogramm ist zunächst das Property *SortType* des ListViews auf *stText* eingestellt. Die Komponente sortiert die Einträge also, wenn sich deren Text ändert, d.h. zunächst beim Einfügen der Elemente und außerdem, wenn deren Text editiert wurde. Da das *OnCompare*-Ereignis nicht bearbeitet wird, sortiert die Komponente alphabetisch nach dem Eintragstext.

Da das *SortType*-Property des TreeViews bei *stNone* belassen wurde, müssen wir die Baumansicht manuell sortieren, und zwar über den Popup-Menüpunkt SORTIEREN NACH NAME. Baumansichten werden niemals vollständig sortiert. Vielmehr können Sie jeweils nur die direkten Untereinträge eines ausgewählten Eintrags sortieren. Die Methode für den Menüpunkt sortiert den gewählten Eintrag mit der Standardmethode:

```
procedure TExplorer.SortierennachName1Click(Sender: TObject);
begin
  Tree.Selected.AlphaSort;
end;
```

Schließlich erlaubt Ihnen das Beispielprogramm, das ListView nach dem Typ der Elemente zu sortieren (Popup-Menü: SORTIEREN NACH TYP). Hierzu müssen wir eine eigene Vergleichsfunktion definieren, die die erste Spalte der Zusatzinformationen (*SubItems[0]*) zweier übergebener Listeneinträge mit der Funktion *CompareStr* vergleicht. Die Vergleichsfunktion soll an die Sortiermethode *CustomSort* übergeben werden; ihre Parameter müssen daher wie folgt deklariert sein:

```
function CompareType(Item1, Item2: TListItem;
                     Data: Integer): integer; stdcall;
begin
  if (Item1.SubItems.Count>0) and (Item2.SubItems.Count>0)
    then CompareType:=CompareStr(Item1.SubItems[0], Item2.SubItems[0]);
end;
```

Anders als eine mit dem *OnCompare*-Ereignis verknüpfte Methode wird diese Funktion von Windows aufgerufen. Sie darf daher keine Methode des Formulars sein und muss außerdem der Aufrufkonvention *stdcall* folgen.

Die Methode für den Menüpunkt SORTIEREN NACH TYP ruft nun statt *AlphaSort* die Methode *CustomSort* auf und übergibt ihr die Adresse der Vergleichsfunktion. Der zweite Parameter wird bei jedem Vergleich an den *Data*-Parameter der Vergleichsfunktion weitergegeben, er wird hier nicht benötigt:

```
procedure TExplorer.SortierennachTyp1Click(Sender: TObject);
begin
  ListView1.CustomSort(@CompareType, 0);
end;
```

Aufräumen des Baums der Shell-Objekte

Zum Abschluss der Shell-Programmierung wenden wir uns der bisher kaum beachteten Freigabe des von der Shell reservierten Speichers zu. Praktischerweise kann diese im vorliegenden Beispielprogramm in einer einzigen Methode stattfinden, nämlich in einer solchen, die mit dem *TTreeView*-Ereignis *OnDeletion* verknüpft wird. Diese wird vor dem Löschen eines jeden Eintrags erzeugt. Alle *PItemIdList*-Zeiger, für die vorher Speicher reserviert wurde, sind in der *NodeInfo*-Struktur gespeichert, so dass sie wie folgt freigegeben werden können:

```
procedure TExplorer.TreeDeletion(Sender: TObject; Node: TTreeNode);
var
  NodeInfo: PNodeInfo;
begin
  NodeInfo := PNodeInfo(Node.Data);
  if Assigned(NodeInfo) then begin
    if (Assigned(NodeInfo.RelativeIDL)) then
      Allocator.Free(NodeInfo.RelativeIDL);
    if (Assigned(NodeInfo.AbsoluteIDL)) then
      Allocator.Free(NodeInfo.AbsoluteIDL);
    Dispose(NodeInfo);
  end;
end;
```

Ähnliche Maßnahmen sind für das Ereignis *OnDeletion* des ListViews erforderlich. Dabei soll noch einmal erwähnt werden, dass die Freigabe der *IShellFolder*-Schnittstelle, die sich in *NodeInfo.ShellFolder* befindet, automatisch erfolgt, und zwar bei der Freigabe von *NodeInfo* über *Dispose*.

8.5 Programmierung eigener COM-Klassen

Während Kapitel 8.4 die Verwendung der von Windows zur Verfügung gestellten COM-Klassen beschrieben hat, geht dieses Kapitel den umgekehrten Weg: Es zeigt, wie Sie Object-Pascal-Interfaces schreiben, die von anderen Anwendungen und der Windows-Shell benutzt werden. Dafür sollten Sie bereits die in Kapitel 2.7 erläuterten Interface-Grundlagen kennen.

8.5.1 Typen von COM-Objekten

Für einen besseren Überblick seien hier einmal die verschiedenen Arten von COM-Objekten zusammengefasst, die Sie mit Object Pascal erzeugen können:

- Objekte, die Interfaces besitzen und nur innerhalb Ihrer Anwendung verwendet werden, brauchen gar nicht unbedingt als COM-Objekte bezeichnet zu werden. In der Terminologie der VCL sind es »Interfaced Objects« , denn sie werden normalerweise in Klassen implementiert, die von *TInterfacedObject* abgeleitet sind.

- Einfache COM-Objekte, die Sie in einer DLL implementieren und die in anderen Anwendungen genutzt werden sollen. Diese Objekte müssen sich in ihrem binären Aufbau an den COM-Standard halten (dafür sorgt alleine Delphis Compiler) und werden durch Klassen implementiert, die Sie von *TComObject* statt von *TInterfacedObject* ableiten (oder auch von der Klasse *TTypedComObject*, wenn Sie eine Typenbibliothek verwenden). Der in Delphi 4 Professional eingeführte Experte, der ein Gerüst für ein solches COM-Objekt bereits automatisch erzeugt (DATEI | NEU | WEITERE-Dialog auf der Seite ActiveX), wurde in Delphi 6 dahingehend erweitert, dass Sie nun ein COM-Objekt für eine bereits unter Windows registrierte Schnittstelle erzeugen können. Wenn Sie den Schalter LISTE innerhalb des Experten drücken, erhalten Sie eine Liste aller zusammen mit einer Typbibliothek registrierten Schnittstellen, die unter Umständen extrem lang ist, da mittlerweile sehr viele Anwendungen Gebrauch von solchen Schnittstellen machen, ohne dass man es als reiner Benutzer solcher Anwendungen ahnen würde.
- COM-Automationsobjekte basieren auf den einfachen COM-Objekten, was sich dadurch zeigt, dass ihre Implementierungsklassen von einem Nachkommen der Klasse *TTypedComObject* abgeleitet werden, und zwar von *TAutoObject*. Dass Sie diese Implementierungsklassen nicht mehr von Hand ableiten müssen, zeigt Kapitel 8.7. Allerdings machen die Typenbibliotheken dieser Objekte die Professional-Version von Delphi erforderlich, da die Standard-Version keinen TLB-Editor enthält. Trotzdem ist es möglich, bereits existierende Programme mit Automationsobjekten (wie etwa die Beispiele aus Kapitel 8.7) mit der Standard-Version zu kompilieren.
- ActiveX-Steuerelemente bauen auf Automationsobjekten auf, indem ihre Implementierungsklasse vom *TAutoObject*-Nachkommen *TActiveXControl* abgeleitet werden. Für den speziellen Fall der ActiveForms enthält die VCL auch noch die Klasse *TActiveFormControl*.

In diesem Kapitel geht es lediglich um den zweiten Typ. Die anderen Typen werden in den Kapiteln 2.7, 8.6, 8.7 und 6.8 ausführlich behandelt. Während bei Anwendungsinternen COM-Objekten (Kapitel 2.7) keine Verwaltungsaufgaben wie die Registrierung der Schnittstellenklassen anfallen, sind solche Aufgaben schon bei Automationsobjekten so umfangreich, dass Sie sich besser durch Delphis Experten helfen lassen sollten (Kapitel 8.7). Bei den einfachen COM-Objekten dieses Kapitels können Sie einen Mittelweg gehen und die wenigen Verwaltungsaufgaben per Hand durchführen.

Unser besonderes Augenmerk gilt in diesem Kapitel aber auch anderen Themen wie den Erweiterungen der Windows-Shell und dem Debuggen von DLLs, wobei wir auch einen Blick hinter die Kulissen der automatischen Freigabe von COM-Objekten werfen werden.

8.5.2 Erweiterungen der Windows-Shell

Bevor wir zu den selbst entwickelten COM-Objekten kommen, mit denen wir die Shell um einen Kontextmenü-Handler erweitern werden, soll ein allgemeiner Überblick über die Erweiterungsmöglichkeiten der Shell gegeben werden, von denen COM-Objekte nur ein kleiner Teil sind.

Überblick über Shell-Erweiterungen

Die Erweiterbarkeit der Explorer-Shell von Windows beruht zunächst einmal auf ihrem dokumentorientierten Aufbau: Während beim Programm-Manager aus früheren Windows-Versionen Programme die wesentlichen Objekte waren, konzentriert sich die aktuelle Shell auf die Dokumente. Zwar konnten Sie auch schon in Windows 3.1 einen Dateityp mit einer Anwendung verknüpfen, so dass diese Anwendung automatisch gestartet wurde, wenn Sie ein Dokument dieses Typs im Dateimanager öffneten, und Sie konnten auch festlegen, wie ein bestimmter Dokumenttyp gedruckt werden sollte, aber die Möglichkeiten der aktuellen Shell gehen weit darüber hinaus. Das Starten der Anwendung und Drucken sind nur zwei Spezialfälle davon, dass Sie das Kontextmenü eines Dokumenttyps beliebig durch neue Menüpunkte erweitern können, wie z.B. zum Abspielen einer Klangdatei oder zum Entpacken eines Archives. Neben dem Kontextmenü zur Datei können Sie auch das *Neu*-Untermenü erweitern, das erscheint, wenn Sie im Explorer *nicht* auf eine Datei klicken, sondern z.B. auf den Desktop-Hintergrund.

Da es dem Benutzer völlig egal sein kann, von welchem Programm jeder dieser Menüpunkte bearbeitet wird, steht für ihn an dieser Stelle wirklich das Dokument im Vordergrund. Die Vorgehensweise in der Windows-Shell hat damit auch große Ähnlichkeit mit der Arbeitsweise in einem OLE-Verbunddokument: Das *Neu*-Untermenü erinnert an das Erstellen eines neuen Objekts über den Befehl OBJEKT EINFÜGEN... in der OLE-Clientanwendung; das Gegenstück zu den Kontextmenü-Erweiterungen sind die Verben eines OLE-Objekts, wobei der OLE-Container diese Verben häufig ebenfalls in einem Kontextmenü zur Wahl stellt.

Sie können die bisher beschriebenen Shell-Erweiterungen bereits durch normale Programme realisieren, wenn Sie die passenden Aufrufbefehle für diese Programme an der richtigen Stelle der Registry eintragen (siehe z.B. im Index von Delphis Win32-Hilfedatei das Kapitel *Modifying the Context Menu for a File Class* für Kontextmenü-Erweiterungen und für das *Neu*-Untermenü das Kapitel *Modifying the New Submenu*). Mit der in Kapitel 4.2 beschriebenen Klasse *TRegistry* können Sie auch Installationsprogramme schreiben, die alle benötigten Registrierungsschlüssel automatisch anlegen.

Shell-Erweiterungen über In-Process-Server

Obwohl Sie auf die erwähnte Weise bereits beliebige Funktionen in einem Kontextmenü unterbringen können, ist ein auf diese Weise entstandenes Menü noch nicht so flexibel, wie man es von einem Kontextmenü erwarten könnte: Der *Kontext*, auf den sich das Menü bezieht, ist alleine der Datei*typ*, denn eine Kontextmenüerweiterung wird immer für eine bestimmte Dateinamenserweiterung registriert. Die Auswahl der Menüpunkte ist noch in keiner Weise vom Inhalt der Datei abhängig.

Um dies zu ändern, bietet Ihnen die Windows-Shell eine weitaus flexiblere Möglichkeit der Kontextmenüerweiterung: Sie legen die Kontextmenüpunkte nicht in der Registry fest, sondern geben dort lediglich ein Programm an, das die Menüpunkte zur Laufzeit aufbaut. Das Programm »weiß« dabei, für welche Datei genau das Kontextmenü sein soll, kann diese Datei also schon vor der Anzeige des Menüs untersuchen, um dann ein wirklich situationsabhängiges Kontextmenü anzeigen zu können.

Dieses »Programm« muss in Form einer In-Process-Server-DLL vorliegen, die ein COM-Objekt zur Verfügung stellt, welches wiederum das *IContextMenu*-Interface implementiert. Bevor wir zu einem konkreten Object-Pascal-Beispielprogramm kommen, sei noch auf die anderen Einsatzgebiete für In-Process-Server in der Windows-Shell hingewiesen:

- Erweiterung des *Eigenschaftsdialogs*, der bei Auswahl des Menüpunkts *Eigenschaften* aus dem lokalen Kontextmenü eines Objekts erscheint, um neue Seiten.
- Flexible Festlegung des *Icons* für den Dateityp mit einer Icon-Handler-Erweiterung. In einer solchen können Sie zur Laufzeit zwischen verschiedenen Icons für ein und denselben Dateityp wählen oder ein situationsabhängiges Icon zeichnen.
- Überwachung von *Kopieroperationen*: Über die *ICopyHook*-Schnittstelle fragt die Windows-Shell bei verschiedenen Dateioperationen alle registrierten *ICopyHook*-Handler um Erlaubnis, ob die Operation durchgeführt werden darf.
- Erweiterung des *Drag&Drop-Kontextmenüs* (das Kontextmenü, das beim Loslassen einer Datei nach dem Ziehen mit der rechten Maustaste erscheint).
- Festlegen der Operation, die ausgeführt wird, wenn das Dokument auf einem anderen Dokument abgelegt wurde (*Drop-Handler*).
- Darstellung eines Dokuments in anderen Formaten, wenn das Dokument auf einem anderen Dokument abgelegt wird (*Daten-Handler*).

Bei den meisten der genannten Erweiterungen genügt es nicht, wie bei der Überwachung von Kopieroperationen eine einzige Schnittstelle (wie etwa die *ICopyHook*-Schnittstelle) zu unterstützen; so müssen Daten-Handler z.B. *IPersistFile* und *IDataObject* unterstützen, Kontextmenü-Handler *IShellExtInit* und *IContextMenu*. Während wir

Kontextmenü-Handler in einem Beispielprogramm genauer untersuchen werden, sei für die anderen Handler auf die Win32-Online-Hilfe und auf Spezialliteratur verwiesen.

Aufruf der Kontextmenü-Erweiterung

Für das Beispielprogramm ist es von Bedeutung, wie genau die dynamische Kontextmenüerweiterung in der Windows-Shell abläuft:

- Wenn der Benutzer mit der rechten Maustaste auf ein Objekt klickt, sucht Windows zunächst an verschiedenen Stellen der Registry nach passenden Kontexmenü-Handlern (Genaueres dazu erfahren Sie in Kapitel 8.5.4).
- Alle Kontextmenü-Handler inklusive unseres Beispielprogramms liegen in Form von DLLs vor, die Windows dann erst einmal laden muss, sofern sie sich nicht bereits im Speicher befinden.
- Als Nächstes fordert Windows jede DLL dazu auf, ein COM-Objekt für die Kontextmenü-Erweiterung zu erzeugen und an Windows zu übergeben.
- Zur Erfüllung der eigentlichen Aufgabe – das Kontextmenü zu erweitern – wendet sich Windows daraufhin ausschließlich an dieses COM-Objekt.

Ablauf der Kontextmenü-Erweiterung

Die Zusammenarbeit mit dem COM-Objekt besteht aus folgenden Schritten:

- Als Erstes initialisiert Windows das Objekt über den Aufruf einer *Initialize*-Methode, die im *IShellExtInit*-Interface definiert ist. Dieser Methode übergibt Windows bereits die Objekte, für die das Kontextmenü aufgerufen wurde (meistens ist das nur ein Objekt, der Benutzer kann aber auch mehrere Objekte markieren und ein Kontextmenü für alle diese Objekte aufrufen).
- Im zweiten Schritt fordert Windows das COM-Objekt dazu auf, seine Menüpunkte in ein vorgegebenes Kontextmenü einzufügen. Das Menü sowie die Position, an die die Menüpunkte geschrieben werden sollen, und weitere Daten gibt Windows an die *QueryContextMenu*-Methode, die im *IContextMenu*-Interface definiert ist. Nach Aufruf dieser Methode zeigt Windows das Kontextmenü an.
- Falls der Benutzer sich für einen der Menüpunkte des COM-Objekts interessiert und ihn markiert, fragt die Windows-Shell das COM-Objekt noch nach einem Hilfstext, der in der Statuszeile des aktuellen Ordners angezeigt werden soll (Methode *GetCommandString* im *IContextMenu*-Interface).
- Falls der Benutzer sich entschließt, den Menüpunkt auch noch auszuführen, kommt die Shell ein weiteres Mal auf das COM-Objekt zurück, in dem es seine *IContextMenu::InvokeCommand* aufruft.

Ein COM-Objekt zur Kontextmenü-Erweiterung muss also zwei Schnittstellen unterstützen: *IShellExtInit* und *IContextMenu*. Natürlich wäre es den Entwicklern der Windows-Shell auch möglich gewesen, alle notwendigen Methoden in nur einer Schnittstelle zu definieren (dazu hätte *IContextMenu* lediglich um die *Initialize*-Methode von *IShellExtInit* erweitert werden müssen), aber durch die Auslagerung der Initialisierung in die *IShellExtInit*-Schnittstelle können mehrere Shell-Erweiterungen für die Initialisierung auf dieselbe Schnittstellen-Definition zurückgreifen, was für den Entwickler eine bessere Übersicht bedeuten kann.

8.5.3 Ein COM-Objekt als Kontextmenü-Handler

Das Beispielprogramm *ComDemo* soll am Beispiel einer Kontextmenü-Erweiterung und eines CopyHook-Handlers für die Windows-Shell zeigen, wie Sie in Object Pascal eigene COM-Objekte erzeugen. Aus Platzgründen kann hier jedoch nur auf die Kontextmenü-Erweiterung eingegangen werden. Dabei werden die Grundbegriffe *Interfaces*, *COM-Objekte*, *Co-Klassen* und *IUnknown* aus Kapitel 2.7 als bekannt vorausgesetzt.

Damit das Beispiel nicht völlig sinnlos ist, sollte es das Menü wirklich *dynamisch* erweitern, also mit Menüpunkten, die nicht für alle Dateiobjekte gleich sind (denn solche statischen Menüpunkte können Sie ja über einfache Registry-Einträge installieren). Um nicht zu kompliziert zu werden und trotzdem eine »maximale Dynamik« zu erreichen, zeigt das Beispielprogramm in seinen Menüpunkten Informationen über die selektierten Dateiobjekte an, für die das Kontextmenü aufgerufen wurde: die Zahl und Gesamtgröße der Dateiobjekte sowie den Namen des ersten Objekts (siehe Abbildung 8.8). Diese Menüpunkte werden grau dargestellt, da sie dem Benutzer nur zur Information dienen.

Um die Menüerweiterung ein- und auszuschalten sowie genauer zu konfigurieren, kommt noch der nicht grau dargestellte Menüpunkt MENÜERWEITERUNG... hinzu, über den Sie den ebenfalls in Abbildung 8.8 gezeigten Einstellungsdialog aufrufen können.

Hinweis: Zur Installation der Kontextmenü-Erweiterung auf Ihrem System verwenden Sie bitte nach der Installation der CD-ROM-Dateien auf der Festplatte das Programm *ComDemoSetup*. Es besteht aus dem in der Abbildung gezeigten Dialog *Shell-Erweiterungen registrieren*.

Abbildung 8.8: Ein durch das Beispielprogramm erweitertes Kontextmenü und der dazugehörige Installations-/ Konfigurationsdialog

ActiveX-Bibliotheken

Der wesentliche Unterschied zwischen den hier entwickelten COM-Objekten und den COM-Objekten aus Kapitel 2.7 (die zur besseren Unterscheidung auch als »Objekte mit Schnittstellen« oder *InterfacedObjects* bezeichnet werden könnten), besteht darin, dass die Objekte des vorliegenden Kapitels über Anwendungsgrenzen hinweg aufgerufen werden (bzw. über die Grenzen zwischen einer Anwendung und einer DLL).

Normalerweise müssen Sie für solche »echten« COM-Objekte sehr viel Verwaltungsaufwand betreiben, wozu die Erzeugung einer Class Factory und die Bereitstellung von vier DLL-Routinen gehört. Dank der Unit *ComServ* entfallen diese beiden Arbeitsschritte in Delphi glücklicherweise. Um den Start in ein neues COM-Projekt noch einfacher zu machen, finden Sie ab Delphi Professional auch noch eine Schablone namens *ActiveX-Bibliothek* auf der Seite *ActiveX* der Objektablage (DATEI | NEU). Durch sie erhalten Sie den folgenden Quelltext, den Sie in der Standard-Version von Delphi manuell eingeben müssen:

```
library Project1;

uses
  ComServ;
```

```
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

Bevor wir uns weiter mit den oben in der *exports*-Klausel aufgelisteten Einträgen und mit den noch übrig bleibenden – nicht von der Unit *ComServ* übernommenen – Verwaltungsaufgaben befassen, werden wir die Co-Klasse für die Shell-Erweiterungs-Schnittstellen deklarieren und implementieren.

Erzeugen neuer COM-Objekte in Delphi *R152*

In Kapitel 2.7.4 wurde bereits gezeigt, wie Sie in Object Pascal ein COM-Objekt erzeugen. Sie benötigen als Erstes eine Co-Klasse, die alle erforderlichen Schnittstellen implementiert. In Kapitel 2.7.4 begann die Klassendeklaration mit der folgenden Zeile:

```
TFormContainer = class(TInterfacedObject, IContainer)
```

TInterfacedObject war dabei eine Klasse der VCL, die die Routineaufgaben der Referenzzählung und der *QueryInterface*-Methode erledigt. Für Shell-Erweiterungen und andere COM-Objekte, die nicht wie in Kapitel 2.7.4 nur innerhalb der Anwendung, sondern auch von außerhalb genutzt werden können, stellt die VCL die Klasse *TComObject* zur Verfügung. Zusammen mit den beiden zu unterstützenden Schnittstellen *IContextMenu* und *IShellExtInit* sieht die Deklaration der Co-Klasse für die Kontextmenü-Erweiterung wie folgt aus:

```
TContextMenuExtender = class(TComObject, IContextMenu, IShellExtInit)
  DataObj: IDataObject;
(* IShellExtInit *)
    function IShellExtInit.Initialize=ShellExtInitialize;
    // TComObject hat bereits eine Initialize-Methode
    function ShellExtInitialize(pidlFolder: PItemIDList;
      lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
 (* IContextMenu *)
  function QueryContextMenu(Menu: HMENU;
    indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
  function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult;
    stdcall;
  function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
    pszName: LPSTR; cchMax: UINT): HResult; stdcall;
end;
```

Die Methodendeklarationen wurden einfach aus den entsprechenden *interface*-Deklarationen der mit Delphi mitgelieferten Unit *ShlObj* kopiert. Die *Initialize*-Methode aus dem *IShellExtIni*-Interface wird jedoch unter dem Namen *ShellExtInitialize* implementiert, damit keine Verwechslung mit der *Initialize*-Methode von *TComObject* auftritt.

Die Variable *DataObj* dient dazu, die in der *Initialize*-Methode übergebenen Daten für die spätere Verwendung in *QueryContextMenu* zwischenzuspeichern.

Hinweis: Der in Kapitel 8.5.1 erwähnte *COM-Objekt*-Experte aus der Objektgalerie kann das Gerüst für unser COM-Objekt automatisch erzeugen. Dazu gehört sowohl der Rumpf der oben gezeigten Klassendeklaration als auch die in Kapitel 8.5.4 beschriebene Registrierung dieses Objekts im *initialization*-Teil der Unit. Allerdings ist die in Delphi 6 neu eingeführte Funktion, automatisch alle in der vorgegebenen Schnittstelle (hier etwa *IContextMenu*) definierten Funktionen in die Klasse zu übernehmen, in diesem Fall nicht verfügbar, denn weder *IContextMenu* noch *IShellExtInit* besitzen die für diese Funktion erforderliche Typbibliothek. Besitzer der Personal-Version von Delphi können übrigens auch ohne diesen Experten dieselben Quelltexte per Hand eingeben und kompilieren.

Implementierung der Schnittstellen-Methoden

Für Kontextmenü-Erweiterungen ist nur ein Parameter der *Initialize*-Methode wichtig: eine *IDataObject*-Schnittstelle, über die das COM-Objekt von der Windows-Shell erfragen kann, für welche Dateien und Objekte das Menü gelten soll. Im Beispielprogramm wird diese Schnittstelle in der schon erwähnten Variablen *DataObj* zwischengespeichert:

```
function TContextMenuExtender.ShellExtInitialize(pidlFolder: PItemIDList;
  lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
begin
  DataObj := lpdobj;
  Result := NoError;
end;
```

Etwas mehr Arbeit erhält die *QueryContextMenu*-Methode: Ihr übergibt Windows das Handle des zu erweiternden Menüs (*Menu*), den Index, an dem die Erweiterung beginnen soll (*indexMenu*), den ersten und letzten freien Befehlscode (*idCmdFirst* und *idCmdLast*) sowie ein paar Flags. Der Befehlscode wird später dazu benötigt, einen vom Benutzer aufgerufenen Menüpunkt zu identifizieren (jedoch nicht in diesem Beispielprogramm, das ja sowieso nur einen einzigen aktivierbaren Menüpunkt besitzt).

Alle Parameter von *QueryContextMenu* außer den Flags werden dazu verwendet, die Windows-API-Funktion *InsertMenu* aufzurufen. *InsertMenu* fügt einen neuen Menüeintrag an einer bestimmten Position des im ersten Parameter angegebenen Menüs ein. Den Text für diesen Menüeintrag bereitzustellen ist eine der Hauptaufgaben eines Kontextmenü-Handlers.

```
function TContextMenuExtender.QueryContextMenu(Menu: HMENU;
  indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
var
  FirstName: String;
  FileCount, Size: Integer;
  MenuText: String;
  AddCount: Integer;
begin
  LoadConfig; // eventuell geänderte CtxMenuActive-Einstellung laden
  AnalyzeData(Size, FileCount, FirstName);
  // erster Menüpunkt wird auf jeden Fall eingefügt:
  InsertMenu(Menu, indexMenu, MF_STRING or MF_BYPOSITION
             or (MF_CHECKED * byte(CtxMenuActive)),
             idCmdFirst, PChar('Menüerweiterung...'));
  AddCount := 1;
  // weitere Menüpunkte nur wenn die Funktion aktiviert ist:
  if CtxMenuActive then begin
    MenuText := 'Größe von ' + IntToStr(FileCount) + ' Dateien: '
                +IntToStr(Size)+' Bytes.';
    InsertMenu(Menu, indexMenu + 1, MF_GRAYED or MF_STRING or
               MF_BYPOSITION, idCmdFirst, PChar(MenuText));
    MenuText := '('+FirstName+')';
    InsertMenu(Menu, indexMenu + 2, MF_GRAYED or MF_STRING or
               MF_BYPOSITION, idCmdFirst, PChar(MenuText));
    inc(AddCount, 2);
  end;
  // Die Shell möchte wissen, wie viele Menüpunkte hinzugekommen sind.
  // Diese Zahl (AddCount) muss mit MakeResult im Funktionsergebnis
  // "verpackt" werden. Da ein HResult-Funktionsergebnis intern aus
  // drei Bestandteilen besteht, muss eine Hilfsfunktion bedient werden:
  Result := MakeResult(SEVERITY_SUCCESS, FACILITY_NULL, AddCount);
end;
```

Die beiden Methoden *LoadConfig* und *AnalyzeData* sowie die Variable *CtxMenuActive* gehören ebenfalls zum Beispielprogramm. *CtxMenuActive* gibt an, ob die grauen Datei-Infos im Menü angezeigt werden sollen, und wird über den Schalter *Kontextmenüerweiterung aktiv* im Konfigurationsprogramm festgelegt (siehe Abbildung 8.8). *AnalyzeData* ermittelt die im Menü angezeigten Daten durch Abfrage der zwischengespeicherten *IDataObject*-Schnittstelle:

Die wichtigste noch verbleibende Methode ist *InvokeCommand*, die so viele Daten als Parameter erhält, dass diese in einer eigenen *TCMInvokeCommandInfo*-Struktur zusammengefasst werden. Da das Beispielprogramm aber nur einen anwählbaren Menü-

punkt erzeugt, benötigt es diese Struktur nicht zur Abfrage des angewählten Menüpunkts, sondern nur um das Handle eines Fensters zu erhalten (*lpici.hWnd*), das im Falle eines Fehlers als Elternfenster für ein Fehlermeldungsfenster dienen kann:

```
function TContextMenuExtender.InvokeCommand(var lpici:
            TCMInvokeCommandInfo): HResult; stdcall;
var
  SetupProgram: string;
  Registry: TRegistry;
begin
  Registry := TRegistry.Create;
  Registry.Rootkey := HKEY_CURRENT_USER;
  Registry.Openkey('Software\DelphiBuchEW\COMObjects', True);
  Registry.WriteInteger('Kontextmenü-Schalter', byte(not CtxMenuActive));
  Registry.Free;
  // aktivieren und Konfigurationsprogramm starten
  SetupProgram := HomeDir+'ComDemoSetup.EXE';
  if (WinExec(PChar(SetupProgram), SW_ShowNormal) < 32) then
    // bei der Verwendung der VCL-Funktion MessageDlg kommt es
    // nach einer Weile zu Abstürzen des Explorers, daher wird hier
    // die API-Funktion für Meldungsfenster verwendet:
    MessageBox(lpici.hWnd, 'Konfigurationsprogramm kann nicht '+
               'gestartet werden.', 'Shell-Erweiterungs-Demo', MB_OK);
  Result := NoError; // NoError ist eine vordefinierte Konstante
end;
```

Da der ausführbare Menüpunkt als An-/Aus-Schalter für die Datei-Info-Menüpunkte dienen soll, besteht die wichtigste Aktion dieser Methode darin, die Variable *CtxMenuActive* umzuschalten. Da die DLL kurz nach dem Schließen des Kontextmenüs wieder aus dem Speicher entfernt wird und so der Wert von *CtxMenuActive* immer wieder verloren geht, muss das Programm diesen auf eine andere Weise speichern, und zwar verwendet es dazu die Registry.

Hinweis: Wenn Sie im Beispielprogramm auch die *ICopyHook*-Handler-Funktion aktivieren, lädt der Explorer die DLL sofort bei seinem nächsten Start in den Speicher und gibt sie nicht mehr frei.

Außerdem ruft *InvokeCommand* noch den Konfigurationsdialog auf, der sich nicht in der DLL, sondern in einer separaten EXE-Datei befindet und bei dem es sich um nichts weiteres als das schon erwähnte Installationsprogramm *ComDemoSetup* handelt. Die Verlagerung des Dialogs in eine eigene DLL hat zwei Vorteile: Der Dialog muss nicht als modaler Dialog aufgerufen werden und blockiert daher nicht die weitere Arbeit mit dem Explorer, und außerdem stürzt der Explorer nicht mit schöner Regelmäßigkeit nach ungefähr 15 weiteren Mausklicks ab (wie er es in den Tests des Autors tat).

Der Vollständigkeit halber sei auch noch die letzte der vier Methoden vorgestellt:

```
function TContextMenuExtender.GetCommandString(idCmd, uType: UINT;
  pwReserved: PUINT; pszName: LPSTR; cchMax: UINT): HResult; stdcall;
begin
  pszName[0] := #0;
  Result := NoError;
end;
```

Das Beispielprogramm verzichtet auf die Bereitstellung von Hilfstexten für die Statuszeile und setzt daher *pszName* in jedem Fall auf einen Leerstring.

Da bereits der *ShellExplorer* aus Kapitel 8.4.3 gezeigt hat, wie Sie über Shell-Schnittstellen an die Daten der Objekte in der Windows-Shell gelangen, würde eine Besprechung der *AnalyzeData*-Methode hier zu weit führen, denn sie macht von anderen, noch nicht erwähnten Shell-Schnittstellen und -Datenstrukturen Gebrauch, um an die Zahl und Größe der selektierten Dateien sowie den Namen der ersten Datei zu gelangen.

Ebenfalls nicht abgedruckt wird die *LoadConfig*-Methode, denn sie liest mit Hilfe der VCL-Klasse *TRegistry* lediglich zwei Einträge aus der Windows-Registry aus (den Wert für die Variable *CtxMenuActive* sowie *HomeDir*, das Verzeichnis, in der sich das Konfigurationsprogramm befindet).

Debuggen von DLLs und Shell-Erweiterungen *R142*

Eine DLL ist alleine, also ohne eine EXE-Datei, von der sie verwendet wird, nicht ausführbar, daher kann sie auch nicht alleine im Debugger untersucht werden. Das Debuggen einer Shell-Erweiterung verläuft ebenso wie das Debuggen jeder anderen DLL nach dem folgenden Prinzip: Sie starten auch im Debugger nicht die DLL selbst, sondern die EXE-Datei, von der die DLL geladen wird. Dazu geben Sie diese EXE-Datei zunächst unter START | PARAMETER im Feld *Host-Anwendung* an. Durch die Ausführung des Menüpunkts START | START wird nun diese Host-Anwendung gestartet, und durch diese wird dann die DLL geladen.

Am einfachsten erhalten Sie die Kontrolle des Debuggers, indem Sie Haltepunkte in die DLL setzen. Es ist jedoch auch möglich, die DLL schon vor dem Starten der EXE-Datei in das Modul-Fenster des Debuggers zu laden (ANSICHT | DEBUG-FENSTER | MODULE, lokales Menü: MODUL HINZUFÜGEN...) und dann im lokalen Menü durch BEIM LADEN ANHALTEN zu bewirken, dass die DLL sofort nach dem Laden im DLL-Startcode angehalten wird.

Die hier vorliegende Shell-Erweiterung ist bereits ein Spezialfall, denn sie wird nicht von einem selbst entwickelten Programm, sondern vom Windows-Explorer gestartet. Glücklicherweise gibt es eine klar definierte EXE-Datei, welche die DLLs der Shell-Erweiterungen lädt: `explorer.exe` im Windows-Hauptverzeichnis (also beispielsweise `c:\winnt\explorer.exe`). Der interessante Umstand, dass normalerweise immer Delphi

vom Explorer aus gestartet wird, und nicht umgekehrt, ist jedoch kein Hindernis. Um die `comdemo.dll` in Delphis integriertem Debugger ablaufen zu lassen, führen Sie die folgenden Schritte durch:

- Geben Sie als *Host-Anwendung* um START | PARAMETER-Dialog die Datei `explorer.exe` mit dem auf Ihrem System gültigen Pfad an.
- Über eine »unsichtbare« Funktion ist es möglich, den bereits laufenden Windows-Explorer zu schließen, ohne die laufenden Anwendungen zu beenden oder den Computer herunterzufahren: Wählen Sie BEENDEN aus dem Start-Menü, halten Sie die Tasten Strg+Alt+Shift und drücken Sie währenddessen mit der Maus den Schalter ABBRECHEN. Die Taskbar, alle offenen Explorer-Fenster und die Explorer-Icons auf dem Desktop-Hintergrund verschwinden, aber die Anwendungen laufen weiter (unsichtbare Anwendungen erreichen Sie über Alt+Tab).
- Setzen Sie eventuell einen Haltepunkt in einen *initialization*-Abschnitt der DLL, in diesem Beispiel etwa den der Unit *ShellHandlers*, wenn Sie das Laden der DLL im Debugger untersuchen wollen.
- Über START | START können Sie nun den Explorer starten. Dieser lädt dann Ihre Erweiterungs-DLL, im Falle von *ComDemo* je nach Konfiguration erst, wenn ein von der DLL erweitertes Kontextmenü aufgerufen wird. Falls Sie bis dahin einen Haltepunkt im Startcode der DLL gesetzt haben, unterbricht Delphi die Ausführung an dieser Stelle.

8.5.4 Verwaltungsaufgaben für ein COM-Objekt

Wir kommen nun wieder zu den speziellen Verwaltungsaufgaben für COM-Objekte zurück, die laut der Übersicht in Kapitel 8.5.1 nur für die COM-Objekte des einfachen Typs anfallen. Zur Erinnerung sei noch einmal gesagt, dass wir am Anfang von Kapitel 8.5.3 damit begonnen haben, eine ActiveX-Bibliothek anzulegen, die zu einer DLL kompiliert wird. Auf dieser Grundlage bauen die weiteren Schritte auf.

Wichtigster Schritt beim Installieren einer COM-Klasse auf einem Windows-System ist die Registrierung des Objekts in der Windows-Registry. Diese Registrierung ist beispielsweise erforderlich, damit andere Anwendungen COM-Objekte dieser Klasse vom Betriebssystem anfordern können und die COM-Objekte dann auch durch das Betriebssystem erzeugt werden können.

Kernbestandteil einer Registrierung ist, einer Klasse einen eindeutigen Bezeichner zuzuteilen, um keine Verwechslung mit anderen Klassen aufkommen zu lassen. Für diesen eindeutigen Bezeichner verlässt sich das COM nicht auf den Einfallsreichtum der Entwickler, sondern fordert, dass dieser Bezeichner maschinell so erstellt wird,

dass er weltweit und für alle Zeit eindeutig bleibt. Für die Kontextmenüerweiterung des Beispielprogramms lautet dieser Bezeichner, der auch als GUID (global unique identifier) bezeichnet wird, beispielsweise *{22910AA1-C028-11D0-9E2F-444553540000}*.

Herkunft des GUID

In Delphi erzeugen Sie einen solchen 128-Bit-GUID einfach über das Tastenkürzel Shift + Strg + G im Quelltexteditor oder (ab der Professional-Version), indem Sie eine neue Typenbibliothek anlegen (DATEI | NEU, Seite *ActiveX*, *Typbibliothek*). Der daraufhin angezeigte Typbibliotheks-Editor enthält in seinem Feld *GUID* einen solchen eindeutigen Bezeichner, der selbst, wenn Sie ihn nicht verwenden, kein weiteres Mal von einem solchen GUID-Generator irgendwo auf der Welt erzeugt werden sollte, denn in seine Berechnung gehen sowohl die aktuelle Systemzeit als auch spezielle Daten der System- oder Netzwerkkonfiguration ein.

Wenn Sie die neue Typenbibliothek nicht brauchen, können Sie diesen Bezeichner also direkt verwenden, ansonsten muss er natürlich der Typenbibliothek als Kennung vorbehalten bleiben. In diesem Fall leiten Sie aus dem Bezeichner der Typenbibliothek einen weiteren Bezeichner ab, indem Sie die letzte Ziffer im ersten Teil des Bezeichners erhöhen. So geht auch Delphi vor, wenn es innerhalb einer ActiveX-Bibliothek beispielsweise eine neue COM-Klasse für ein ActiveX-Control erzeugt.

Das Beispielprogramm verwendet eine eigene Unit namens *ComDemoConsts*, um zum GUID der Kontextmenü-Erweiterung zwei passende Konstanten zu definieren:

```
const
  CLSID_ContextMenu: TGUID = '{22910AA1-C028-11D0-9E2F-444553540000}';
  CLSID_ContextMenuStr = '{22910AA1-C028-11D0-9E2F-444553540000}';
```

Im Gegensatz zur ersten kann die zweite Konstante als String verwendet werden. Sie wird später zur Registrierung des COM-Objekts als Shell-Erweiterung in der Registry benötigt; bei der Implementierung von COM-Objekten, die keine Shell-Erweiterungen sind, ist eine solche Stringkonstante normalerweise nicht erforderlich.

Registrieren des COM-Objekts

Nach der Erzeugung eines GUID müssen Sie für normale COM-Objekte noch einen zweiten Schritt durchführen: Sie erzeugen einen »Objektgenerator«, von dem das Betriebssystem später die einzelnen COM-Objekte anfordern wird. Hierzu genügt ein einziger Aufruf der VCL im *initialization*-Teil der Unit, in der das COM-Objekt implementiert wird:

```
initialization
  TComObjectFactory.Create(
    ComServer, // Name des Servers, hier als Stringkonstante implementiert
    TContextMenuExtender, // Object-Pascal-Klasse (Co-Klasse)
```

```
      CLSID_ContextMenu, // GUID des Objekts
      'EWDemoContextMenuHandler', // Name des Objekts
      'COM-Objekte zur Kontextmenü-Erweiterung', // Beschreibung
      ciMultiInstance, // Instanziierungsmodus (siehe COM-Automation)
      tmApartment); // Threading-Model (siehe Online-Hife)
  end.
```

Durch den obigen *Create*-Aufruf wird die VCL darüber informiert, dass die Klasse *TContextMenuExtender* eine COM-Klasse mit der angegebenen GUID implementiert. Die VCL übernimmt daraufhin alle weiteren Verwaltungsaufgaben. Zu diesen gehört, dass die VCL die DLL mitsamt ihrem vollständigen Pfad in der Windows-Registry unter *HKEY_CLASSES_ROOT\CLSID\{22910AA1-...}\InProcServer32* einträgt, wenn jemand von außen die Funktion *DllRegisterServer* aufruft. Nach dieser Registrierung steht die neue COM-Klasse systemweit zur Instanziierung bereit, wobei die VCL die für diese Instanziierung (Erzeugung eines neuen Objekts dieser Klasse) notwendige Class Factory zur Verfügung stellt.

Bei dem »jemand«, der *DllRegisterServer* aufruft, kann es sich z.B. um die Delphi-IDE handeln, und zwar müssen Sie dazu nur den Menüpunkt START | ACTIVEX-SERVER EINTRAGEN aufrufen. Eine allgemeintauglichere Möglichkeit ist jedoch, dass Sie *DllRegisterServer* in einem Installationsprogramm aufrufen. Hierzu kommen wir im nächsten Abschnitt.

Hinweis: Der letzte Parameter im Aufruf von *TComObjectFactory.Create* dient nur dem Zweck, der VCL mitzuteilen, welches Threading-Model im oben erwähnten Registry-Knoten *InProcServer32* registriert werden soll. Es ist Sache des Entwicklers, sicherzustellen, dass die DLL sich auch an das registrierte Threading-Model hält. Shell-Erweiterungen werden nur geladen, wenn sie für das *Apartment*-Model registriert wurden. Das Beispielprogramm funktioniert ohne besondere Vorkehrungen mit diesem Threading-Model (zu weiteren Details bezüglich *tmApartment* siehe die Online-Hilfe).

Registrierung der Shell-Erweiterungen

Bisher ist unsere DLL also als COM-Server registriert. Die Windows-Shell *könnte* also jetzt die in der DLL implementierten Kontextmenü-Handler-Objekte erzeugen, wenn sie nur wüsste, dass diese DLL zur Kontextmenü-Erweiterung gedacht ist.

Die DLL muss also auch noch als Shell-Erweiterung registriert werden. Für Kontextmenü-Erweiterungen erzeugen Sie dazu einen Schlüssel namens *ShellEx\ContextMenuHandlers*, unter dem Sie die Class-ID Ihrer Erweiterung entweder direkt als Unterschlüssel eintragen oder als Standardwert eines Unterschlüssels, dessen Name eine besser lesbare Form hat als die Class-ID.

Unter welchem Schlüssel der Registry der Schlüssel *ShellEx\ContextMenuHandlers* eingeordnet wird, hängt vom Einsatzgebiet der Kontextmenü-Erweiterung ab:

- Menüs, die nur bei einem bestimmten Dateityp aufgerufen werden sollen, werden unter dem Namen des Dateityps registriert (dieser Name ist wiederum unter der Dateiendung dieses Dateityps registriert, z.B. unter *Hkey_Classes_Root\.tvf* für die Dateien des TreeDesigners, siehe hierzu auch die später folgenden Codeauszüge und den Abschnitt *Registrieren einer Anwendung* in Kapitel 4.2.3).
- Für spezielle Arten von Shell-Objekten (Ordner der Shell, Laufwerke, Verzeichnisse des Dateisystems und Drucker) verwenden Sie die Schlüssel *Folder*, *Drive*, *Directory* und *Printers*, die sich direkt unter der Wurzel *Hkey_Classes_Root* befinden.
- Wenn Sie eine Erweiterung für alle diese Typen und Dateiobjekte der Shell registrieren wollen, tragen Sie sie unter '*' ein (ebenfalls nur eine Ebene unter der Wurzel).

Abbildung 8.8 zeigt, dass Sie im Dialog des *ComDemo*-Installationsprogramms (Projekt *ComDemoSetup*) die Wahl zwischen verschiedenen Einsatzbereichen der beiden Shell-Erweiterungen der DLL haben. Das Programm passt die entsprechenden Systemeinstellungen an, wenn Sie den Dialog mit *Ok* schließen. Dabei wiederholt es grundsätzlich jedes Mal die gesamte Konfigurierung, so dass sich dieser Dialog auch als Installationsprogramm verwenden lässt. Im DATEI-Menü des Dialogs finden Sie auch eine Option zum Deregistrieren aller Einträge.

Für seine Registrierungs- und Deregistrierungsmaßnahmen verwendet das Programm drei verschiedene Werkzeuge:

- Die Funktionen *DllRegisterServer* und *DllUnregisterServer* der DLL *ComDemo*, die ganz einfach in eine Anwendung importiert werden können (zum Import von DLL-Funktionen siehe auch Kapitel 8.3),
- die VCL-Klasse *TRegistry* (siehe Kapitel 4.2) und
- die VCL-Routinen *CreateRegKey* und *DeleteRegKey*, die im Falle des *HKEY_CLASSES_ROOT*-Baumes komfortabler handzuhaben sind als *TRegistry*.

Code-Auszüge *R153*

Von der ohne die Formularklasse ungefähr 100 Zeilen langen Unit des Installationsprogramms soll hier zur Veranschaulichung ein kleiner Ausschnitt genügen, der letztendlich zur Verknüpfung des Menu-Handlers mit den TVF-Dateien führt. Zunächst werden die Funktionen der Server-DLL importiert:

```
function DllRegisterServer: HResult; external 'COMDEMO.DLL';
function DllUnregisterServer: HResult; external 'COMDEMO.DLL';
```

Beim Drücken des OK-Schalters registriert das Programm die DLL mit deren *DllRegisterServer*-Funktion und versieht die TreeDesigner-Dateierweiterung `.tvf` mit einem Namen:

```
if DllRegisterServer <> NoError then
  MessageDlg('Fehler bei der Autoregistrierung der Server-DLL.',
             mtError, [mbOk], 0);
CreateRegKey('CLSID\' + CLSID_ContextMenuStr + '\InProcServer32',
             'ThreadingModel', 'Apartment');
CreateRegKey('.tvf', '', 'TreeDesignerDoc');
```

Unter dem Schlüssel der Dokumentbezeichnung wird nun die *ContextMenuHandler*-Information hinzugefügt oder gelöscht, je nachdem, ob der entsprechende Markierungsschalter (*KeTVF*) im Dialog gewählt ist oder nicht. Da es im Dialog sehr viele Registrierungsschalter gibt, wird diese Abfrage in eine eigene Unterroutine ausgelagert, die dann für die TVF-Dateien wie folgt aufgerufen wird:

```
UpdateRegKey(KeTVF, 'TreeDesignerDoc\shellex\'+
                    'ContextMenuHandlers\DelphiBuchEWDemo',
                    '', CLSID_ContextMenuStr);

procedure UpdateRegKey(CreateCheck: TCheckBox;
                       Key, ValueName, Value: string);
begin
  if CreateCheck.Checked then CreateRegKey(Key, ValueName, Value)
    else DeleteRegKey(Key);
end;
```

Die Parameter von *CreateRegKey* und *DeleteRegKey* entsprechen den Parametern der *TRegistry*-Methoden. Siehe hierzu Kapitel 4.2 und die Online-Hilfe.

8.5.5 Selbst definierte COM-Schnittstellen

In den Kapiteln 2.7 und 5.7.5 haben wir selbst definierte Interfaces bereits innerhalb einer Delphi-Anwendung verwendet. An dieser Stelle gehen wir noch einen kleinen Schritt weiter und definieren ein eigenes Interface, das auch von anderen Anwendungen aufgerufen werden kann. Anhand dieses Interfaces soll auch gezeigt werden, wie die Referenzzählung von COM-Objekten funktioniert. Es handelt sich jedoch nicht um ein Interface, das von anderen Anwendungen sinnvoll genutzt werden könnte.

Eine neue Schnittstelle zum Testen der Referenzzählung

Das neue Interface wird in der Unit *RefCount* unter dem Namen *ITestReferenceCount* definiert und im Programm *ComDemo* verwendet, das in Kapitel 8.5.3 schon zur Erweiterung der Windows-Shell verwendet wurde. Dieses Kapitel wird demnach auf einige bisher noch nicht gezeigte Stellen des *ComDemo*-Quelltextes eingehen.

Wir beginnen mit der Schnittstellendefinition:

```
type
  TReleaseNotifyProc = procedure;
  ITestReferenceCount = interface(IUnknown)
    ['{22910AA3-C028-11D0-9E2F-444553540000}']
    function GetRefCount: Integer;
    procedure RegisterReleaseNotifyProc(P: TReleaseNotifyProc);
  end;
```

Die Aufgabe der beiden Methoden ist relativ einfach:

- Von *GetRefCount* soll eine Anwendung, die das COM-Objekt verwendet, Informationen über den aktuellen Stand von dessen Referenzzähler erhalten.
- *RegisterReleaseNotifyProc* entspricht dem Installieren eines Ereignishandlers für ein Ereignis im Objektinspektor: Sie erwartet als Parameter eine Prozedur, die aufgerufen werden soll, wenn ein bestimmtes Ereignis eintritt. Dieses Ereignis ist die Freigabe des COM-Objekts, wenn der Referenzzähler den Wert 0 erreicht. Ziel dieser Benachrichtigung ist es, etwas Licht in das Dunkel der völlig automatischen Verwaltung der COM-Objekte durch die VCL zu bringen.

Hinweis: Die mit dem Ereignis verknüpfte Prozedur ist nicht zu verwechseln mit einer Ereignisbehandlungs-*Methode*, für die *TReleaseNotifyProc* als *procedure of object* hätte deklariert werden müssen (zur Deklaration von Events siehe Kapitel 6.3.2).

Ein GUID für das Interface

Der entscheidende Unterschied zwischen dem oben gezeigten Interface und den früheren Interface-Deklarationen aus Kapitel 2.7 und 5.7.5 liegt in der – zumindest für einen Object-Pascal-Quelltext – ungewöhnlich schwer verständlich aussehenden Angabe *{22910AA3-C028-11D0-9E2F-444553540000}*, die sehr große Ähnlichkeit zu den schon in Kapitel 8.5.4 erzeugten Class-IDs aufweist. Bei diesem GUID handelt es sich nicht um einen Bezeichner für die Klasse, sondern um einen Bezeichner für das Interface. Wie die COM-Klassen eines In-Process-Servers brauchen auch die selbst definierten Interfaces einen solchen Bezeichner.

Überschreiben von Implementierungsmethoden

Die Schnittstelle *ITestReferenceCount* soll der Einfachheit halber von der Co-Klasse aus Kapitel 8.5.3 implementiert werden. Dazu wird *TContextMenuHandler* zunächst um eine Variable zur Speicherung der zu benachrichtigenden Prozedur erweitert. Außerdem muss *TContextMenuHandler* etwas unternehmen, damit sie von der automatischen Freigabe des COM-Objekts erfährt, denn bei dieser Gelegenheit soll sie ja die Benachrichtigungs-Prozedur aufrufen.

Da ein COM-Objekt nur anlässlich des Aufrufs der *_Release*-Methode automatisch freigegeben wird, brauchen wir hierzu nur eine neue *_Release*-Methode zu schreiben, die natürlich die alte (von der VCL vordefinierte) *_Release*-Methode aufrufen muss, und dann – falls der Referenzzähler zu Beginn auf 1 steht – die Benachrichtigung absenden kann.

Dieser Vorgang hat große Ähnlichkeit mit dem Überschreiben von virtuellen Methoden normaler Object-Pascal-Klassen, weist aber auch einige Unterschiede dazu auf. Im Listing der neuen *_Release*-Methode fällt zunächst auf, dass die neue Methode einen anderen Namen haben kann als die alte:

```
function TContextMenuExtender.New_Release: Integer;
begin
  if RefCount = 1 then
    if Assigned(TestNotifyProc) then TestNotifyProc;
  Result := _Release;
end;
```

Der andere Name wurde verwendet, damit keine Verwechslung mit den geerbten Methoden aufkommen kann. Denn selbst wenn wir den ursprünglichen Namen *_Release* verwenden würden, wäre es damit noch nicht getan – zum Überschreiben der alten Methode müssten wir die Interface-Methode in der Klassendeklaration auf jeden Fall explizit mit der neuen Methode verknüpfen.

Dazu kommt noch, dass die *_Release*-Methode über *IUnknown* an alle Schnittstellen weitervererbt wird. Die Co-Klasse braucht zwar nur eine einzige *_Release*-Methode, diese muss aber mit allen Schnittstellen verknüpft werden:

```
TContextMenuExtender =
  class(TComObject, IContextMenu, IShellExtInit,
        IUnknown, ITestReferenceCount)
  (* ITestRefCount (selbst definiertes Interface) *)
  public
    TestNotifyProc: TReleaseNotifyProc;
    function IUnknown._Release = New_Release;
    function IContextMenu._Release = New_Release;
    function IShellExtInit._Release = New_Release;
    function ITestReferenceCount._Release = New_Release;
    function GetRefCount: Integer;
    procedure RegisterReleaseNotifyProc(P: TReleaseNotifyProc);
    function New_Release: Integer; stdcall;
  (* IShellExtInit *) // siehe Kapitel 8.5.3
     ...
  (* IContextMenu *) // siehe Kapitel 8.5.3
     ...
  end;
```

Würden wir beispielsweise die Verknüpfung *IContextMenu._Release = New_Release* weglassen, würde *New_Release* zwar automatisch aufgerufen werden, sobald eine *ITest-ReferencCount*-Variable ungültig wird, nicht aber, wenn eine *IContextMenu*-Variable freigegeben wird.

Die beiden weiteren Methoden der Co-Klasse brauchen nicht weiter erläutert zu werden:

```
function TContextMenuExtender.GetRefCount: Integer;
begin
  Result:=RefCount; // RefCount ist eine Variable der
                    // Erbklasse TInterfacedObject
end;

procedure TContextMenuExtender.
  RegisterReleaseNotifyProc(P: TReleaseNotifyProc);
begin
  TestNotifyProc := P;
end;
```

Test der neuen Methoden

Das Beispielprogramm *TestClient* stellt zunächst einmal eine Prozedur zur Verfügung, die bei der Freigabe der zu überwachenden COM-Objekte aufgerufen werden soll:

```
procedure ReleaseNotify;
begin
  MessageDlg('Das Objekt wurde freigegeben.', mtInformation, [mbOK], 0);
end;
```

Diese Prozedur ist zwar nicht optimal, weil die Dialogbox nur kurz aufblinkt, falls die Freigabe des COM-Objekts beim Beenden des Programms geschieht, aber sie soll für unsere Zwecke genügen.

Abbildung 8.9: Die Oberfläche des Programms TestClient

In der Formularklasse des Testprogramms sind die folgenden privaten Variablen deklariert:

```
RefCountedObject: ITestReferenceCount;
AddRefs: array[1..100] of ITestReferenceCount;
AddRefCount: Integer;
```

Das Programm arbeitet mit einem einzigen COM-Objekt, das beim Drücken des Schalters *Objekt erzeugen* erstellt werden soll:

```
procedure TForm1.BtnInitialisierenClick(Sender: TObject);
var
  LocalCtxMenu: IContextMenu;
  Obj: IUnknown;
begin
  Obj := CreateComObject(CLSID_ContextMenu);
  LocalCtxMenu := OBJ as IContextMenu; // << nur zum Test
  RefCountedObject := OBJ as ITestReferenceCount;
  RefCountedObject.RegisterReleaseNotifyProc(ReleaseNotify);
  DisplayRefCount;
end;
```

Die Erzeugung des Objekts findet in der Methode *CreateComObject* statt, *CreateComObject* liefert die *IUnknown*-Schnittstelle des Objekts zurück. Da in der Klassenvariable aber eine *ITestReferenceCount*-Schnittstelle benötigt wird, muss diese noch über den *as*-Operator erfragt werden. Zur Demonstration erfragt die Prozedur auch noch eine *IContextMenu*-Schnittstelle des Objekts. Die Frage ist nämlich nun, wie hoch der Referenzzähler des COM-Objekts am Ende der Prozedur zum Zeitpunkt des Aufrufs der Methode *DisplayRefCount* ist.

```
procedure TForm1.DisplayRefCount;
begin
  if not Assigned(RefCountedObject) then
    RefCountDisplay.Caption := '-'
  else RefCountDisplay.Caption := IntToStr(RefCountedObject.GetRefCount);
end;
```

Nach dem Drücken des Schalters zur Laufzeit bestätigt sich die Annahme, dass der Referenzzähler den Wert 3 erhalten hat. Zuerst wird er im Verlauf der Routine *CreateComObject* mit 1 initialisiert, danach zweimal um eins erhöht, und zwar jedes Mal, wenn eine Schnittstelle des Objekts einer anderen Variablen zugewiesen wird (also hier *LocalCtxMenu* und *RefCountedObject*).

Die automatische Freigabe findet erst im *end* der Methode statt, weshalb der Aufruf von *DisplayRefCount* vor diesem *end* noch keine Auskunft über die korrekte Freigabe liefert. Hierzu drücken Sie den Schalter *Aktualisieren*, bei dem *DisplayRefCount* erneut aufgerufen wird. Der Referenzzähler ist zu diesem Zeitpunkt wieder bei 1 angelangt, da am Ende der Methode *BtnInitialisierenClick* sowohl die Variable *Obj* als auch die Variable *LocalCtxMenu* ungültig geworden sind.

Referenzen in Arrays

Das Beispielprogramm zeigt das Verhalten des Referenzzählers auch für den Fall, dass Sie noch viel mehr Referenzen auf dasselbe Objekt in einem Array speichern. Mit einem Druck auf den Schalter *Zusätzliche Referenz über Array* führen Sie die folgende Methode aus:

```
procedure TForm1.BtnArrayReferenzClick(Sender: TObject);
begin
  if (AddRefCount < 100) then begin
    inc(AddRefCount);
    AddRefs[AddRefCount] := RefCountedObject;
    DisplayRefCount;
  end;
end;
```

Sie kopiert einfach die Variable *RefCountedObject* in ein noch unbenutztes Feld des Arrays und aktualisiert die Ausgabe des Referenzzählers, so dass deutlich wird, dass dieser bei jedem Schalterdruck um eins erhöht wird. Im weiteren Programmverlauf wird kein einziges Mal explizit auf die so gesetzten Array-Felder zugegriffen.

Mit diesem Array wird auch demonstriert, wie leistungsfähig und zuverlässig Delphis automatische Verwaltung der COM-Objekte ist: Selbst für jede einzelne Interface-Variable in einem teilweise beschriebenen Array wird automatisch *_Release* aufgerufen, sobald das Array ungültig wird (da das Array *AddRefs* als statisches Array innerhalb des Formulars deklariert ist, wird es zusammen mit dem Formular ungültig, wenn das Formular bzw. die Anwendung geschlossen wird).

Freigabe bei RefCount = 0

Wie schon erwähnt können Sie die Dialogbox der Methode *ReleaseNotify* kaum sehen, da das Programm unmittelbar nach deren Erscheinen beendet ist. Um die endgültige Freigabe eines Objekts einmal in Ruhe »beobachten« zu können, drücken Sie den letzten Schalter des Testformulars: *Zweites Objekt lokal verwenden*. Die *OnClick*-Methode zu diesem Schalter konstruiert ein zweites COM-Objekt, bewahrt aber keine Referenzen auf dieses Objekt in statischen Variablen auf, sondern nur in der lokalen Variable *LocalObject*. Wenn diese im *end* der Prozedur ungültig wird, erhält der Referenzzähler dieses lokalen Objekts den Wert 0, die VCL von *TestClient* ruft *_Release* auf, dies bewirkt den Aufruf unserer oben geschriebenen Methode *New_Release*, die die Benachrichtigung über die Objektfreigabe aussendet und anschließend die Methode *_Release* der VCL aufruft (diesmal handelt es sich um die VCL innerhalb der DLL), die das Objekt dann zu guter Letzt mit *TObject.Free* freigibt.

```
procedure TForm1.BtnLokaleVerwendungClick(Sender: TObject);
var
  LocalObject: ITestReferenceCount;
  Obj: IUnknown;
```

```
begin
  LocalObject:=CreateComObject(CLSID_ContextMenu) as ITestReferenceCount;
  LocalObject.RegisterReleaseNotifyProc(ReleaseNotify);
  // hier hat RefCount noch den Wert 1:
  MessageDlg('Das Objekt wird verwendet, RefCount = '
    +IntToStr(LocalObject.GetRefCount)+'.', mtInformation, [mbOK], 0);
end;
```

Debuggen des COM-Objekts

Am Ende von Kapitel 8.5.3 wurde bereits beschrieben, wie Sie die *ComDemo.DLL* als Shell-Erweiterung debuggen können. Mit dem hier verwendeten Programm *TestClient* steht nun eine alternative Host-Anwendung zur Verfügung, während deren Ausführung Sie die DLL ebenfalls debuggen können. Aktivieren Sie dazu das `ComDemo.dll`-Projekt und geben Sie statt `explorer.exe` im Feld *Host-Anwendung* des START | PARAMETER-Dialogs die Datei `testclient.exe` an. Nachdem Sie dann die Anwendung gestartet haben, können Sie die DLL *ComDemo* nicht nur von außen auf die Funktion des Referenzzählers testen, sondern auch »von innen«. Wir hätten daher gar keine Benachrichtigungsprozedur gebraucht, um die Freigabe der COM-Objekte überwachen zu können; ein Breakpoint im Code der Funktion *New_Release* hätte genügt (dabei wäre es wieder empfehlenswert gewesen, die Optimierung des Compilers auszuschalten).

8.6 COM-Automation: Clients und Internet-Explorer

Auch die COM-Automation (früher auch als OLE-Automation bezeichnet) basiert auf COM-Objekten, die seit Delphi 3 direkt, also ohne zusätzliche Vermittlung durch die VCL, in eigenen Programmen verwendet werden können. In diesem Kapitel geht es zunächst um einfache Automations-Clients (alternativ auch als Automations-Controller bezeichnet), Kapitel 8.7 wird sich dann den Automations-Servern widmen.

Abschnitt 8.6.1 gibt einen leichten Einstieg in die Verwendung von Automationsobjekten und Varianten. Abschnitt 8.6.2 führt die Typenbibliotheken ein, durch die zumindest zum Teil auf die Varianten verzichtet werden kann. Diese Abschnitte verwenden in den (sehr einfachen) Beispielen Microsoft Word als Automations-Server. Falls Sie Word nicht verwenden wollen, können Sie diese einfachen Techniken auch am Beispiel des TreeDesigner-Automations-Servers (Kapitel 8.7) ausprobieren. Die Abschnitte 8.6.3 und 8.6.4 verwenden schließlich als Beispiel den Internet-Explorer, weil dies wohl die am weitesten verbreitete viel genutzte Automations-Server-Anwendung sein dürfte.

8.6.1 COM-Automationsobjekte und Varianten

Mit Hilfe der COM-Automation kann eine Anwendung auf objektorientierte Weise die Dienste einer anderen Anwendung in Anspruch nehmen. Durch diese Objektorientie-

rung hebt sich die Automation deutlich von vielen anderen Möglichkeiten der Kooperation zwischen Anwendungen ab (wie z.B. der Steuerung über DDE-Makros).

Automation

Der Begriff der »Automation« bezieht sich auf die automatische Ausführung von Routineaufgaben in Anwendungen. Eine häufig benutzte Möglichkeit, Routineaufgaben zu automatisieren, besteht im Schreiben oder Aufzeichnen von Makros. Der Benutzer kann so später mehrere aufgezeichnete Arbeitsschritte auf einen Schlag automatisch ausführen lassen. Da die aufgezeichneten Arbeitsschritte nur ohne Variation durchgeführt werden können, sind derartige Makros noch nicht besonders flexibel. Daher stellen viele größere Standard-Anwendungen, insbesondere im Office-Bereich, auch noch eine Programmiersprache bereit, in der die Arbeitsschritte durch Kontrollstrukturen wie Bedingungsabfragen und Schleifen gesteuert werden können.

Mit Hilfe der COM-Automation müssen Sie nun keine eigene Programmiersprache entwickeln, um Ihre Anwendung automatisch steuerbar zu machen, sondern es genügt, die Fähigkeiten Ihrer Anwendung in Automationsobjekten zur Verfügung zu stellen. Dadurch wird Ihre Anwendung zu einem *Automations-Server* gemacht und kann aus beliebigen Programmiersprachen oder auch aus anderen Anwendungen, die über eine Programmiersprache verfügen, gesteuert werden (diese sind dann die *Automations-Clients* oder *Automations-Controller*).

Automation in Delphi

Unter Delphi können Sie die COM-Automation auf zwei verschiedene Arten nutzen:

- ohne direkten Kontakt mit den COM-Schnittstellen der Automation, wenn Sie die schon seit Delphi 2 bekannten Varianten verwenden
- seit Delphi 3 werden COM-Interfaces auch direkt von Object Pascal unterstützt und bieten einen effizienteren Zugang zur Automation.

Im einfachsten Fall entsteht die objektorientierte Sichtweise der COM-Automation dadurch, dass der gesamte Automations-Server als ein Objekt betrachtet wird, das Sie in der Client-Anwendung wie jedes andere Object-Pascal-Objekt verwenden können. Erst Beispiele lassen erkennen, wie enorm einfach die Steuerung anderer Anwendungen auf diese Weise ist.

COM-Automations-Clients *R169*

Bevor wir in Kapitel 8.7.1 unseren eigenen COM-Automations-Server schreiben, den wir dann natürlich auch aus einem eigenen Automations-Client steuern werden, sehen wir uns hier ein Beispiel an, wie eine häufig verwendete kommerzielle Anwendung, Microsoft Word, per COM-Automation gesteuert werden kann.

Auch eine große Anwendung wie Word bekommen Sie mit einem einzigen Aufruf von *CreateOleObject* »in den Griff«, als OLE-Objekt geben Sie hier *Word.Basic* an:

```
uses ComObj; // enthält die Funktion CreateOleObject
...
var
  MsWord: Variant;
begin
  MsWord := CreateOleObject('Word.Basic');
```

Glücklicherweise ist die Bezeichnung »Basic« eine leere Drohung. In Delphi verwenden Sie von Word Basic bzw. Visual Basic für Word lediglich die Namen der Methoden, können sich aber ansonsten in der Object-Pascal-Syntax wohl fühlen. Um beispielsweise die Namen aller Textmarken des gerade in Word 7 geöffneten Dokuments in ein Memo-Feld der Delphi-Anwendung zu schreiben, genügen die drei folgenden Zeilen:

```
TMCount := MsWord.ZählenTextmarken;
for i := 1 to TMCount do
  Memo.Lines.Add(MsWord.TextMarkeName(i));
```

Das folgende Beispiel verwendet Word 8 (Word 97) dazu, ein Dokument zu laden (*OpenDialog1.FileName*) und alle Absätze, die mit der Formatvorlage *FormatName.Text* formatiert sind, in das Memo-Feld zu kopieren, wobei *App* eine zweite Variable des Typs *Variant* ist:

```
App := CreateOleObject('Word.Application.8');
App.Documents.Open('TESTDATEI.DOC');
MsWord.StartOfDocument;
// Den Formatvorlagen-Filter für den nächsten Suchvorgang setzen:
MsWord.EditFindStyle(Style := FormatName.Text);
repeat
  MsWord.EditFind(Find := ''); // Find = '' -> beliebigen Text suchen:
  if MsWord.EditFindFound then
    Memo.Lines.Add(MsWord.Selection); // in Word markierter Text -> Memo
until not MsWord.EditFindFound;
MsWord.EditFindStyle(Style := ''); // Filter zurücksetzen
MsWord.AppClose;
```

Wie Sie sehen, kommt in diesen Beispielen nur die freie und einsehbare Pascal-Syntax vor – alle und nicht nur manche Parameter werden in Klammern geschrieben, keine »$«-Zeichen verunzieren den Namen von String-Variablen / Methoden, und auch sonst gibt es keine Spur vom zeilenorientierten Basic, in dem die Syntax einer *if*-Anweisung davon abhängig ist, ob Sie sie in eine Zeile schreiben oder auf mehrere Zeilen verteilen.

Eine etwas ausführlichere Version dieses Programms, in der Sie die zu durchsuchende Datei aus einem Dialog auswählen, die zusätzliche Fehlerüberprüfungen durchführt und auch Rückmeldungen gibt, finden Sie auf der CD-ROM unter dem Namen *WordCtrl*. Es enthält auch eine Word-7-Version des Beispiels, bei der die Word-Befehle noch

in deutscher Sprache gehalten sind. (In Word 95 musste man seine Programme noch übersetzen, damit sie auf anderssprachigen Word-Versionen funktionierten!)

Hinweis: Das obige Beispiel nutzt die von Word 97 neu eingeführten Objekte eher halbherzig, indem es zwar mit *Documents.Open* einmal eines dieser Objekte verwendet, ansonsten aber alle weiteren Aktionen über das globale, schon in Word 7 (Word 95) enthaltene *Word.Basic*-Objekt (im Programm also die *MsWord*-Variante) abwickelt. Ein Beispiel für den ausgiebigen Einsatz der neueren Word-Objekte wird mit Delphi mitgeliefert und befindet sich im Verzeichnis DEMOS\ACTIVEX\OLEAUTO\WORD8; aber auch der nächste Abschnitt wird noch einmal etwas mehr von diesen anderen Objekten Gebrauch machen.

Varianten

Die bisher gezeigten Aufrufe von Automatisierungs-Methoden sehen zwar an sich ganz logisch aus, mit der Object-Pascal-Sprachdefinition von Delphi wäre so etwas jedoch nicht zu machen gewesen. So enthalten die obigen Funktionsaufrufe Umlaute (wie in *ZählenTextmarken*), die Sie normalerweise nicht in Bezeichnern verwenden dürfen; die Parameter können als Zuweisung geschrieben werden (wie oben bei *EditFindStyle*), und überhaupt werden die Namen der Word-Funktionen in keiner von Delphis Units deklariert.

Um dies und noch mehr möglich zu machen, erweiterte Borland die Sprache Object Pascal in Delphi 2 um die *Varianten*. Varianten sind Variablen des vordefinierten Typs *Variant* (wie oben die Variable *MsWord*), für die Compiler und Laufzeitbibliothek eine Unmenge von Sonderbehandlungen durchführen müssen, denn diese Variablen können zur Laufzeit nicht nur einen nahezu beliebigen Typ annehmen, sie können ihn auch noch beliebig oft wechseln.

Hinter dieser flexiblen Fassade steckt aus technischer Sicht eine einfache Record-Struktur des Namens *TVarData*, die dem *TVarRec* (siehe Kapitel 2.5.3) ähnelt, aber mit 16 Bytes etwas platzaufwändiger ist. Sie ist in der Unit *System* deklariert und definiert den inneren Aufbau einer jeden Varianten-Variablen. Für Werte, deren Größe vier Bytes übersteigen, speichert die Variante lediglich einen Zeiger auf einen dynamisch verwalteten Speicherbereich.

Falls Sie Informationen über den Inhalt einer Variante benötigen, können Sie Ihre *Variant*-Variable per Typenumwandlung als *TVarData* darstellen, oder Sie können den Typencode der Variante (das erste Feld von *TVarData*) mit der Funktion *VarType(Variant)* abfragen. *VarType* liefert einen der ebenfalls in *System* deklarierten Konstanten wie *varSmallint*, *varString*, *varEmpty* oder *varUnknown*.

Verwendung von Varianten *R140*

Der eigentliche Zweck der Varianten ist jedoch, dass Sie sich bei der Programmierung gerade *nicht* um diese Details kümmern müssen, sondern dass Sie die Variante wie eine normale Variable verwenden können, z.B. ihr Werte zuweisen, mit ihr rechnen können und sie als Funktionsparameter übergeben können.

Da Sie mit Varianten fast alles machen können, braucht ihre Verwendung nicht besonders beschrieben zu werden. Wichtig ist nur, dass Ihre Anwendung den Geschwindigkeitsvorteil von Delphi nicht ausnutzt, sondern um ein Vielfaches langsamer als theoretisch möglich abläuft, wenn sie Operationen wie die folgende ausführt:

```
// Multiplizieren von String-Varianten:
v := Variant('2') * Variant('2');
if v = '4' then Experiment_Erfolgreich('!');
```

Zur Art, wie Delphi bei der Kombination von normalen Variablen und Varianten vorgeht, und welche Umwandlungen erlaubt sind, befinden sich in der Online-Hilfe von Delphi weitere Informationen (in der Object-Pascal-Referenz unter *Datentypen, Variablen und Konstanten*, Unterknoten *Variante Typen*).

Hinweis: Die obige Typenumwandlung *Variant('2')* ist erforderlich, weil der Compiler den Operator »*« nur für Zahlen und Varianten zulässt, *'2'* ohne Typenumwandlung aber noch als String ansieht.

Außer einzelnen Werten kann eine Variante sogar Arrays von Varianten enthalten (da jedes Element einen anderen Typ haben kann, kann ein Varianten-Array ähnlich vielfältig strukturiert sein wie ein Record). In Kapitel 7.4.1 wurde ein Varianten-Array bereits als Parameter für die Methode *Locate* der Klasse *TDataSet* verwendet:

```
Files.Locate('Feld1;Feld2', VarArrayOf([3, 'A']), ...);
```

Wenn Sie ein Varianten-Array nicht an eine Funktion weitergeben, sondern einzelne Elemente selbst ansprechen wollen, behandeln Sie es wie eine normale Array-Variable (z.B. *Variant[x] oder Variant[x, y]*).

Während *VarArrayOf* ein vorinitialisiertes Array erzeugt, können Sie mit *VarArrayCreate* auch ein nicht-initialisiertes Varianten-Array erzeugen. Ein nützliches Anwendungsbeispiel dafür gibt Kapitel 8.7.5.

Erwähnenswert sind schließlich zwei spezielle Variantenwerte, die dem *nil*-Wert eines Zeigers ähnlich sind:

- Im Gegensatz zu anderen Variablen werden alle Varianten vom Compiler bzw. Programmcode automatisch mit einem vordefinierten Wert initialisiert, und zwar mit dem Wert *Unassigned*.

- Der Wert *Null* gibt Ihnen die Möglichkeit, »unbekannte oder fehlende« Daten zu kennzeichnen, ohne dafür den Wert *Unassigned* verwenden zu müssen.

Hinweis: Seit Delphi 6 müssen Sie die Unit *Variants* einbinden, wenn Sie Varianten-Funktionen wie etwa *VarArrayOf* verwenden wollen.

Varianten als COM-Objekte

Weitere Zusatzregeln für Varianten erlauben es schließlich, dass Sie Varianten wie Objekte verwenden können, indem Sie Methoden aufrufen, wobei die Parameter dieser Methoden aus Zuweisungen bestehen können. Da der Compiler zur Übersetzungszeit nicht wissen kann, ob eine Variante zur Laufzeit ein COM-Automatisierungsobjekt enthält, erlaubt er diese Art des Methodenaufrufs bei jeder Variante. Zur Laufzeit wird allerdings eine Exception erzeugt, wenn Sie versuchen, eine Methode für eine ungeeignete Variante (bzw. eine Methode, die nicht vom Automationsserver unterstützt wird) aufzurufen.

OleVariant

Außer dem Typbezeichner *Variant* kennt Delphi auch noch die *OleVariant*, die im Gegensatz zur *Variant* nur OLE-kompatible (COM-kompatible) Typen enthalten kann (also *Byte*, *Currency*, *Real*, *Double*, *LongInt*, *Integer*, *Single*, *SmallInt*, *WideString*, *TDateTime*, *Variant*, *OleVariant*, *WordBool* und alle Interface-Typen). *Variant* und *OleVariant* sind miteinander zuweisungskompatibel. Bei der Zuweisung einer *Variant* an eine *OleVariant* werden allerdings Typen, die nicht COM-kompatibel sind, in COM-kompatible Typen konvertiert, so z.B. der Typ *AnsiString* in *WideString*. Bei umgekehrter Zuweisung einer *OleVariant* an eine *Variant* treten keine Änderungen auf.

8.6.2 Typenbibliotheken

Die im letzten Abschnitt beschriebene Variantentechnik ist leicht anzuwenden, hat aber doch entscheidende Nachteile:

- die fehlende Möglichkeit der Typenüberprüfung durch den Compiler, weil dieser ja nicht weiß, ob das erst zur Laufzeit festgelegte COM-Objekt tatsächlich über die aufgerufenen Methoden verfügt, und welche Parameter diese erwarten bzw. welche Ergebnisse sie liefern,
- die geringere Geschwindigkeit zur Laufzeit, weil dann ja noch all diese Typenüberprüfungen nachgeholt werden müssen, und das unter Umständen noch mehrfach (z.B. wenn eine COM-Methode in einer Schleife aufgerufen wird).

Eine Typenbibliothek liefert dem Compiler die fehlenden Informationen, damit er die Aufrufe von COM-Objekten so schnell gestalten kann wie die Aufrufe von normalen Methoden. Damit ist eine Typenbibliothek so etwas Ähnliches wie der Interface-Teil einer Unit, wobei die Implementierung dieser »Unit« durch eine andere Anwendung gegeben ist, die meistens sogar in einer anderen Sprache als Object Pascal geschrieben wurde.

Typenbibliotheken in Object Pascal

Da Delphis Compiler jedoch Typenbibliotheken nicht in ihrem ursprünglichen binären Format verstehen kann, müssen diese zuerst in die Sprache Object Pascal *importiert* werden. Hierzu bietet die Delphi-IDE den Menüpunkt PROJEKT | TYPBIBLIOTHEK IMPORTIEREN an, der Sie in eine Dialogbox führt, die zunächst einmal alle in der Windows-Registry angemeldeten Typenbibliotheken auflistet (Abbildung 8.10). Wenn sich die gesuchte Bibliothek nicht in dieser Liste befindet, aber bereits auf Ihrem System als Datei vorliegt, können Sie sie mit HINZUFÜGEN angeben. Der Dateiauswahldialog zeigt bereits, dass eine Typenbibliothek sich in verschiedenen Arten von Dateien verbergen kann: Als `*.tlb` und `*.olb` sowie als Teil der Datei, in der auch der Code des COM-Servers liegt (`*.dll`, `*.exe` und `*.ocx`).

Abbildung 8.10: Der Dialog zum Importieren von Typenbibliotheken, hier mit der in Kapitel 8.6.3 benötigten MSHTML-Bibliothek

Haben Sie im Import-Dialog eine Bibliothek aus der Liste ausgewählt bzw. eine neue hinzugefügt, nennt die zweite Liste des Dialogs die in dieser Bibliothek enthaltenen Klassen, wobei eine Bibliothek nicht unbedingt eine Klasse enthalten muss, sondern auch nur aus Schnittstellenbeschreibungen (Interfaces) und der Definition einfacher Typen bestehen kann. Wenn Sie eine ausführlichere Übersicht über eine Typenbibliothek erhalten wollen, müssen Sie den Import-Dialog verlassen und den Typenbibliotheksedi tor öffnen (DATEI | ÖFFNEN, Dateityp TYPBIBLIOTHEK).

Die Klassenliste im Import-Dialog hat jedoch noch eine kleine Zusatzfunktion, und zwar ist sie editierbar. Dies hat den Zweck, dass Sie Klassen umbenennen können, falls die vorgegebenen Namen mit Klassen, die bereits in der Komponentenpalette installiert sind, in Konflikt geraten.

Wenn Sie den Import-Dialog mit INSTALLIEREN... oder UNIT ANLEGEN beenden, übersetzt Delphi die komplette Typenbibliothek in eine Object-Pascal-Unit, die neben der reinen Übersetzung auch noch zusätzlichen Code enthält, durch den die Co-Klassen der Bibliothek in Object-Pascal-Klassen verpackt werden, die Sie dann in Ihrem Code ähnlich wie normale Object-Pascal-Klassen instanziieren können.

Hinweis: Aus sprachlichen Gründen kann Delphi nicht alle Namen der Typenbibliothek unverändert in die Pascal-Unit übernehmen. Reservierte Bezeichner wie *Type* werden z.B. zu *Type_* erweitert und die Standardschnittstellen von Co-Klassen, die als Komponente in die Komponentenpalette aufgenommen werden sollen, werden um den Namen der Anwendung erweitert, so dass beispielsweise *Document* zu *WordDocument* wird. Eine Dokumentation aller Umbenennungen finden Sie als Kommentar im Quelltext der von Delphi erzeugten Unit, welche vorzugsweise in den Verzeichnissen DELPHI\OCX\SERVERS (für die vorgegebenen Units) oder DELPHI\IMPORTS (bei neu importierten Bibliotheken) zu finden ist.

Die erweiterte Import-Funktion

Seit der Version 5 kann Delphi COM-Co-Klassen auf Knopfdruck in Komponenten für die Komponentenpalette umwandeln. Dies funktioniert im Prinzip wie das Verpacken eines ActiveX-Controls in eine Komponente (siehe Kapitel 6.8, insbesondere der Blick hinter die Kulissen in 6.8.4). Und tatsächlich kann man ein ActiveX-Control ja einfach als spezielle Co-Klasse, die bestimmte wohldefinierte Schnittstellen unterstützt, ansehen.

Um die neue Funktion zu aktivieren, stellen Sie sicher, dass im Import-Dialog die Option KOMPONENTEN-WRAPPER GENERIEREN gewählt ist. Wenn Delphi nun also Komponentenhüllen für die CoKlassen erzeugt, bietet es sich an, dass diese Komponenten gleich in der Palette installiert werden. Zu diesem Zweck bietet der Dialog weitere Elemente an, die Sie wahrscheinlich schon aus dem Dialog zur Installation von Kompo-

nenten kennen: die Angabe der Seite, auf der die Komponente installiert werden soll, und der bei der Erzeugung eines Packages verwendete Suchpfad. Sie können auch nur eine Unit erzeugen und trotzdem Komponentenhüllen erhalten, wenn Sie die KOMPONENTEN-WRAPPER-Option wählen, aber den Dialog mit UNIT ANLEGEN abschließen.

Die COM-Server der Komponentenpalette

Falls Sie jetzt häufig verwendete COM-Server wie aus Microsoft Office oder dem Internet Explorer einsetzen wollen: Freuen Sie sich nicht zu früh, dass Sie nun die komfortable Import-Funktion anwenden können, denn ab der Professional-Ausgabe (in Delphi 5 schon in der Standard-Ausgabe) hat Borland diese Arbeit bereits für Sie erledigt und die Gelegenheit genutzt, um die Zahl der Komponenten in der Komponentenpalette zu erhöhen: Auf der Seite SERVER finden Sie all die Komponenten, die Sie auch beim manuellen Importieren der entsprechenden Typenbibliotheken erhalten würden. Allerdings ist zu beachten, dass diese vorgegebenen Komponenten einen bestimmten Entwicklungsstand der COM-Server konservieren. Sollten danach neue Versionen der Office-Anwendungen oder des Internet Explorers mit erweiterten COM-Schnittstellen herauskommen, so müssen Sie diese manuell importieren, um sie nutzen zu können.

TOleServer

Jede von Delphi erzeugte Hüllklasse für einen COM-Server (genauer für eine Co-Klasse eines COM-Servers) ist von *TOleServer* abgeleitet und erbt von dieser die folgenden wichtigen Properties, die festlegen, auf welche Weise die Komponente ihre Verbindung mit dem COM-Server aufbauen soll, und die Sie im Objektinspektor einstellen können. Dabei gibt *ConnectKind* die grundsätzliche Art der Verbindungsaufnahme an:

- Bei *ckRunningOrNew* wird sozusagen eine Verbindung zum nächstbesten COM-Server aufgebaut. Das heißt, dass eine laufende Instanz des Server-Programms verwendet wird, falls vorhanden, und dass ansonsten eine neue Instanz gestartet wird.
- Bei *ckNewInstance* wird in jedem Fall eine neue Instanz des COM-Servers gestartet.
- Bei *ckRunningInstance* wird vorausgesetzt, dass der COM-Server bereits läuft, und die Verbindung wird nur zu einem solchen schon gestarteten Server aufgebaut.
- *ckRemote* besagt, dass ein Server auf einem anderen Computer verwendet werden soll. Hier kommt das Property *RemoteMachine* ins Spiel, in dem Sie den Namen dieses Computers angeben.

- Eine grundsätzlich andere Verbindungsart ist *ckAttachToInterface*, denn hier wird die Verbindung nicht mit der Co-Klasse des Servers, sondern nur mit einem Interface hergestellt, wenn Sie dieses zur Laufzeit auf irgendeine Weise bekommen haben und dann an die Methode *ConnectTo* weitergeben.

Wenn Sie das Property *AutoConnect* einschalten, wird die Verbindung sogar automatisch beim Programmstart aufgebaut (in der Verbindungsart *ckAttachToInterface* geht das nicht).

Verwenden der COM-Server-Komponenten *R173*

Das auf der CD enthaltene Beispielprogramm *Wordctrl2* steuert Word unter Zuhilfenahme einer Typenbibliothek bzw. unter Verwendung der vorgefertigten Server-Komponenten. Es kann fünf ganz einfache Aktionen ausführen, die im Folgenden in der Reihenfolge der zugehörigen Aktionsschalter erläutert werden.

Hinweis: Um unter Delphi 6 Personal oder einer beliebigen Ausgabe von Delphi 3-4 auf die Automationsschnittstelle von Word zugreifen zu können, müssen Sie Words Typenbibliothek `Msword8.olb` manuell importieren (PROJEKT | TYPBIBLIOTHEK IMPORTIEREN). Sie befindet sich im OFFICE-Verzeichnis; bei einer Word-Einzel-Installation ist dieses dem Word-Verzeichnis untergeordnet.

Abbildung 8.11 zeigt das Formular des Programms zur Entwurfs- und zur Laufzeit. Zur Entwurfszeit sind je eine Komponente der Klassen *TWordDocument* und *TWordApplication* sowie zwei *TWordParagraphFormat*-Komponenten sichtbar. Alle vier wurden aus der *Server*-Seite der Komponentenpalette entnommen.

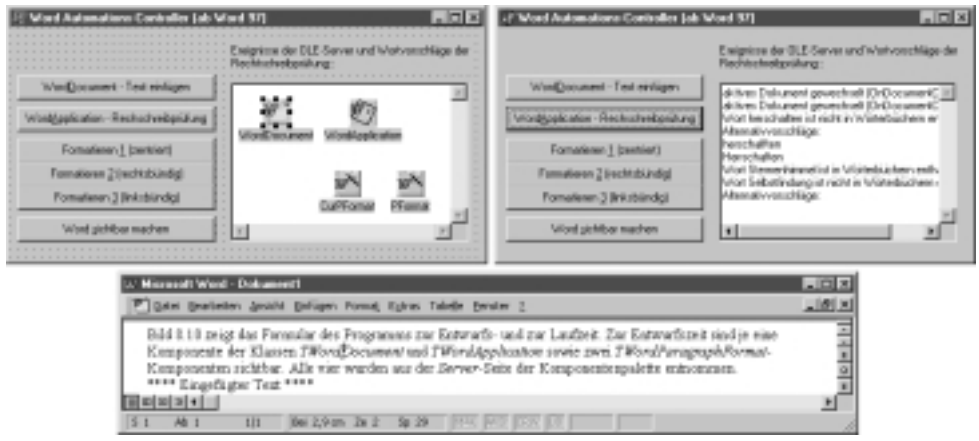


Abbildung 8.11: WordCtrl2 nutzt zwei der neuen COM-Server-Hüllkomponenten.

Am einfachsten haben Sie die Programmierung, wenn Sie die Voreinstellungen von *AutoConnect = True* und *ConnectKind = ckRunningOrNew* unverändert lassen; in *WordCtrl2* wurde dies bei den ersten beiden Server-Komponenten so gemacht. Im Falle von *WordApplication* und *WordDocument* können Sie dann im Programmcode einfach davon ausgehen, dass diese Objekte zur Verfügung stehen, und müssen sich nicht um Initialisierung oder Freigabe kümmern.

Hinweis: Wenn *AutoConnect* auf *False* eingestellt ist, stellt die Komponente die Verbindung auch automatisch her, aber erst, wenn Sie die erste Methode der Komponente aufrufen oder ein Property auslesen, das eine Verbindung erfordert (z.B. *DefaultInterface*). Manchmal kann es nützlich sein, dass eine Verbindung schon vorher aufgebaut wird: Wenn das Beispielprogramm eine neue Instanz von Word erzeugt, enthält diese z.B. noch kein Dokument und das Property *WordApplication.ActiveDocument* wird erst durch die Verbindungsaufnahme der *WordDocument*-Komponente gültig. Der Zugriff auf *WordApplication.ActiveDocument* würde dann zu einem Fehler führen, wenn nicht vorher manuell ein Dokument-Objekt erzeugt werden würde, z.B. durch manuelle Verbindungsaufnahme der Dokument-Komponente (*WordDocument.Connect*).

Für das *WordDocument* bedeutet die automatische Verbindungsaufnahme oder die manuelle Verbindungsaufnahme mit *Connect*, dass beim ersten Zugriff auf dieses Objekt ein neues Word-Dokument angelegt wird. Sollte Word bis dahin noch nicht gestartet sein, wird dies automatisch nachgeholt.

Das folgende Listing zeigt die mit dem ersten Aktionsschalter des Beispielprogramms verknüpfte Methode, die einen Text in dieses neue Word-Dokument einfügt.

```
procedure TForm1.DocumentActionClick(Sender: TObject);
var
  Text: String;
begin
  // bei AutoConnect = False vorher WordDocument.Connect ausführen!
  if InputQuery(Caption, 'Einzufügender Text:', Text) = True then
    WordApplication.ActiveDocument.Content.InsertAfter(Text);
end;
```

Ein Word-Dokument besteht natürlich nicht einfach nur aus Text, sondern hat eine ziemlich komplexe Struktur, die in der Automationsschnittstelle durch viele verschiedene, teilweise geschachtelte COM-Objekte repräsentiert wird. Der Inhalt des Dokuments lässt sich beispielsweise über das Objekt/Property *Content*, aber auch über die in der nächsten Beispielmethode verwendete Kollektion *Paragraphs* ansprechen. Da die Beschreibung der Word-Programmierung eigene umfangreiche Bücher erforderlich

macht, muss sich dieses Kapitel mit kleinen Beispielen begnügen, bei denen Sie die verschiedenen Elemente der Word-Schnittstelle ohne große Erläuterung hinnehmen müssen.

Beim Schreiben einer Methode wie der obigen können Sie erfreulicherweise hinter jedem nach einem Objektbezeichner folgenden Qualifizierungspunkt die Programmierhilfe des Delphi-Editors in Anspruch nehmen und sich die in diesem Objekt enthaltenen Elemente in einer Liste anzeigen lassen. Dies erleichtert den Umgang mit umfangreichen Schnittstellen wie der Word-Automationsschnittstelle sehr und ist ein weiterer Vorteil gegenüber der in Abschnitt 8.6.1 praktizierten Verwendung von Varianten.

Hinweis: Um Zugriff auf ein bereits geöffnetes Word-Dokument zu erhalten, verwenden Sie das *Documents*-Property des *Application*-Objekts von Word. Das erste Dokument wäre dann z.B. *WordApplication.Documents.Item(Index)*. Es kann mit der Methode *Activate* aktiviert und dann über *Application.ActiveDocument* verändert werden.

Interne Arbeitsweise

An dieser Stelle ist vielleicht ein kurzer Blick hinter die Kulissen interessant. Es soll verdeutlicht werden, was es bedeutet, dass die Klasse *TWordApplication* nur eine von Delphi erzeugte Hüllklasse für die Co-Klasse von Word ist. Es bedeutet zunächst, dass Word keine Klasse namens *TWordApplication* definiert, also kann es auch kein Property *TWordApplication.ActiveDocument* geben. Word definiert allerdings eine Schnittstelle namens *_Application*, die über ein solches Property verfügt.

Beim Importieren der Typenbibliothek hat Delphi auch diese Schnittstelle in eine Object-Pascal-Deklaration übersetzt. Die Hüllkomponente *TWordApplication* arbeitet intern mit einer solchen *_Application*-Schnittstelle, die Sie übrigens über das Property *TWordApplication.DefaultInterface* abfragen können.

Aufrufe von Word-Methoden oder Zugriffe auf Word-Properties leitet *TWordApplication* an diese Schnittstelle weiter, z.B. im Falle des Lesezugriffs auf das Property *ActiveDocument* (den vollständigen Quelltext finden Sie ab Delphi 5 in der Datei `delphiverzeichnis\ocx\servers\word97.pas`, sonst in der beim manuellen Import erzeugten Datei):

```
function TWordApplicationProperties.Get_ActiveDocument: WordDocument;
begin
  Result := DefaultInterface.Get_ActiveDocument;
end;
```

Get_ActiveDocument liefert nun ein *WordDocument* zurück, und dies ist nicht wieder ein Objekt einer von Delphi erzeugten Hüllklasse (wie könnte Word auch Objekte von Del-

phi-Klassen erzeugen), sondern ein COM-Interface, das Sie unter Delphi wie jedes andere COM-Interface ansprechen können. *WordDocument* ist ein von Delphi erzeugtes Alias für das Original-Interface *_Document*. Das Property *_Document.Content* liefert eine Schnittstelle des Typs *Range*, und diese verfügt schließlich über eine Methode, die oben wie folgt aufgerufen wurde:

```
WordApplication.ActiveDocument.Content.InsertAfter(Text);
```

Zweite Beispielmethode

Der zweite Schalter fordert Sie zum Eingeben eines Wortes auf, das dann an die Rechtschreibprüfung von Word weitergegeben wird. Falls diese das Wort nicht als korrekt verifizieren konnte, werden Alternativvorschläge von Word erbeten und gegebenenfalls im Memo-Feld des Programms aufgelistet. Die hier in Anspruch genommenen Funktionen werden nicht vom Word-Dokument, sondern von der Word-Anwendung, also vom Objekt *WordApplication* angeboten:

```
if WordApplication.CheckSpelling(TestWort) then
   Memo1.Lines.Add('Wort ist in Wörterbüchern enthalten.')
else begin
   Memo1.Lines.Add('Wort nicht gefunden, Alternativen:');
   WortListe := WordApplication.GetSpellingSuggestions(TestWort);
// WortListe ist eine Schnittstelle des Typs SpellingSuggestions
   for i := 1 to WortListe.Count do // Nummerierung beginnt bei 1
     Memo1.Lines.Add(WortListe.Item(i).Name);
end;
```

Vorsicht: In Word werden Objektsammlungen (Collections) normalerweise mit Eins beginnend durchnummeriert. Falls Sie gewöhnt sind, Schleifen über VCL-Arrays von 0 bis *Count* – 1 durchzählen zu lassen, können hier leicht Fehler entstehen.

Verbinden der COM-Server-Komponenten mit Interfaces R174

Als Letztes soll *WordCtrl2* noch zeigen, was es mit dem Verbindungsmodus *ckAttachToInterface* auf sich hat. Hierzu zeigen die drei mittleren Schalter des Beispielformulars drei Möglichkeiten, den ersten Absatz im *WordDocument* zu formatieren (jeweils über dieselben Elemente der Word-Programmierschnittstelle).

Hintergrundwissen zur Word-Programmierschnittstelle: Die Ausrichtung des Absatzes kann über das Property *Alignment* eingestellt werden. Dieses ist ein Element des *Format*-Objekts, von dem es für jeden einzelnen Absatz ein Exemplar gibt. Der erste Absatz, auf den wir es hier abgesehen haben, lässt sich durch *Paragraph.Item(1)* ermitteln.

Für eine einfache Änderung wie die Anpassung der Ausrichtung wäre es natürlich nahe liegend, einfach

```
WordDocument.Paragraphs.Item(1).Format.Alignment := ...;
```

aufzurufen, aber Ziel soll hier sein, die *Format*-Schnittstelle in einer Variablen des Programms zwischenzuspeichern, was bei umfangreicheren Operationen durchaus Sinn machen kann. Der Schnittstellentyp des *Format*-Properties ist *_ParagraphFormat*, und da dies die Standardschnittstelle der Co-Klasse *ParagraphFormat* ist, hat Delphi eine Hüllkomponentenklasse *TWordParagraphFormat* erzeugt.

Ziel soll es nun sein, eine Komponente des Typs *TWordParagraphFormat* in das Formular einzufügen und mit dem oben angesprochenen *Format* zu verbinden. Der Verbindungsmodus *AutoConnect* ist hier sicher nicht mehr sinnvoll, denn während bei den bisherigen Objekten immer klar war, welche Word-Anwendung bzw. welches Word-Dokument gemeint war (immer die »nächstbeste« Word-Instanz und ein neues Dokument), gibt es viele verschiedene *Format*-Objekte, und wir müssen genau angeben, welches davon gemeint ist. Die Properties der Server-Komponente müssen dazu auf *AutoConnect = False* und *ConnectKind = AttachToInterface* eingestellt werden.

Im Formular von *WordCtrl2* wurde die erste *TWordParagraphFormat*-Komponente mit dem Namen *CurPFormat* versehen. Der Schalter *Formatieren 1* führt folgenden Code aus:

```
procedure TForm1.Format1Click(Sender: TObject);
begin
  CurPFormat.ConnectTo(WordDocument.Paragraphs.Item(1).Format);
  CurPFormat.Alignment := wdAlignParagraphRight;
end;
```

Entscheidend ist der Aufruf der Methode *ConnectTo*. Vor diesem Aufruf lässt sich das Objekt nicht verwenden, da ja nicht schon automatisch eine Verbindung aufgebaut wurde wie in den früheren Fällen. Nach dem Verbindungsaufbau verläuft alles wie gehabt.

Hinweis: Nichtsdestotrotz kann der automatische Verbindungsmodus auch bei *TWordParagraphFormat* eingesetzt werden. Falls Sie dies tun, erhalten Sie nämlich einfach ein völlig neues *ParagraphFormat*-Objekt – mit anderen Worten ein Absatzformat, welches noch keinem Absatz zugewiesen ist. Dies soll durch den zweiten Formatieren-Schalter des Beispielprogramms demonstriert werden.

Das Formular von *WordCtrl2* enthält eine zweite *TWordParagraphFormat*-Komponente mit *AutoConnect = True* und *ConnectKind = ckRunningOrNew*. Auf Druck des Schalters

Formatieren 2 führt das Beispielprogramm die folgenden Zeilen aus, die dieses neue Absatzformat einrichten und dann dem ersten Absatz zuweisen:

```
PFormat.LeftIndent := 10; // PFormat: TWordParagraphFormat
PFormat.Alignment := wdAlignParagraphCenter;
WordDocument.Paragraphs.Item(1).Format := PFormat.DefaultInterface;
```

Verzicht auf die Komponenten

Der dritte und letzte Schalter demonstriert schließlich, dass es im Verbindungsmodus *ckAttachToInterface* leicht fällt, ganz auf Delphis Hüllkomponenten zu verzichten. Wie erwähnt, hat das *Format*-Property den Interface-Typ *_ParagraphFormat*. Delphi erlaubt es ja, Variablen dieses Typs zu deklarieren:

```
procedure TForm1.Format3Click(Sender: TObject);
var
  PFormatIntf: _ParagraphFormat;
begin
  PFormatIntf := WordDocument.Paragraphs.Item(1).Format;
  PFormatIntf.Alignment := wdAlignParagraphLeft;
end;
```

Da man sich dank der automatischen Referenzzählung der COM-Objekte um die Freigabe dieser Schnittstelle keine Gedanken machen muss, ist dies gewissermaßen noch einfacher als die Verwendung der Hüll-Komponenten, denn da muss man ja noch den Verbindungsmodus bedenken. Der Einsatz der Hüll-Komponenten ist also besonders dann sinnvoll, wenn der automatische Verbindungsmodus Anwendung findet, und außerdem, wenn es um die Bearbeitung von Events geht, was in Kapitel 8.6.3 am Beispiel des Internet Explorers demonstriert werden wird.

Hinweis: Oben verwendete Konstanten wie *wdAlignParagraph* sind ebenfalls in der Typenbibliothek von Word definiert. Um mit dem Inhalt dieser Bibliothek vertrauter zu werden, ihren großen Umfang abschätzen zu können und das Angebot dieser Bibliothek von den durch Delphi hinzugefügten Extras unterscheiden zu können (z.B.: *ParagraphFormat* und *_ParagraphFormat* sind in der Bibliothek zu finden, nicht aber *TWordParagraphFormat*), ist es empfehlenswert, die Original-Bibliothek (`Msword8.olb`) einmal mit DATEI | NEU im Typenbibliotheks-Editor von Delphi anzuzeigen.

Vorzeitiges Beenden des Automations-Servers

Wenn zwei eigenständige Anwendungen sich gegenseitig aufrufen, gilt es den Fall zu bedenken, dass der Benutzer eine der Anwendungen schließt. Wenn es dann in der anderen Anwendung nicht zu Programmfehlern kommen soll, muss diese Anwendung vom Beenden der Partneranwendung informiert werden. Word erzeugt in die-

sem Fall ein *OnQuit*-Ereignis, welches das Beispielprogramm dazu nutzt, die Verbindung von allen vier OleServer-Komponenten zu trennen. Das *OnQuit*-Ereignis wird auch von *TWordApplication* nach außen weitergegeben und kann ganz einfach über den Objektinspektor mit einer Methode verknüpft werden:

```
procedure TForm1.WordApplicationQuit(Sender: TObject);
begin
  Memo1.Lines.Add('Quit');
  WordDocument.Disconnect;
  WordApplication.Disconnect;
  PFormat.Disconnect;
  CurPFormat.Disconnect;
end;
```

Wenn im umgekehrten Fall der Client zuerst beendet wird, erfährt Word dies dank der Referenzzählung des COM automatisch, so dass hier keine Fehler entstehen. Falls Word erst durch den Client gestartet wurde, wird Word durch das Beenden des Clients automatisch beendet.

8.6.3 Einbindung des Internet Explorers

Die Macht der COM-Automation soll nun in diesem Abschnitt anhand eines etwas größeren und auch sinnvolleren Beispielprogramms demonstriert werden. Dabei soll diesmal der Internet Explorer das Ziel unserer Programmierabsichten sein, da dieser sich aufgrund seiner generellen Verfügbarkeit und seiner häufigen Anwendung wohl am besten für ein solches Beispiel eignet (er wird zwar erst seit Delphi 5 in der Komponentenpalette mitgeliefert, kann aber auch in Delphi 3 und 4 noch manuell über die Import-Funktion für ActiveX-Controls eingebunden werden).

Das Beispielprogramm (Abbildung 8.12) soll folgende Eigenschaften haben:

- eine Grundfunktionalität zum »Surfen« im Web, also die Möglichkeit zum Abrufen von Web-Seiten, Verfolgen von Links und zum Zurückblättern.
- eine individuelle Benutzerschnittstelle, die für den Vollbildmodus ausgelegt ist, unter platzsparendem Verzicht auf Titelleiste, Menüs und Symbolleisten, dafür aber mit einem hierarchischen TreeView anstelle der einfachen History-List zum Zurückblättern (ein- und ausblendbar über das Menü oder Shift+F5).
- einige demonstrative Funktionen wie das Auflisten verschiedener Inhalte der HTML-Dokumente und die Anzeige der vom Internet Explorer erzeugten Events (ein- und ausblendbar über das Menü oder F5).
- eine Notizfunktion, mit der man Bereiche von verschiedenen Web-Seiten in einem globalen Speicher für »Notizen« zusammentragen kann. Dies soll so ablaufen, dass man einen Bereich jeweils markiert, wenn die Seite geladen ist, und dann den Menüpunkt MARKIERTEN BEREICH NOTIEREN aufruft (F8). Dabei sollen die Notizen

nicht im Textformat übertragen werden (denn dazu könnte man ja jeden beliebigen Texteditor verwenden), sondern inklusive Formatierung im HTML-Format.

- die Möglichkeit, die zusammengetragenen Notizen wie ein neues HTML-Dokument anzusehen und in einer Datei zu speichern (NOTIZEN ANZEIGEN, NOTIZEN SPEICHERN).

Das Beispielprogramm ist auf der CD unter dem Namen *WebCollector* zu finden.

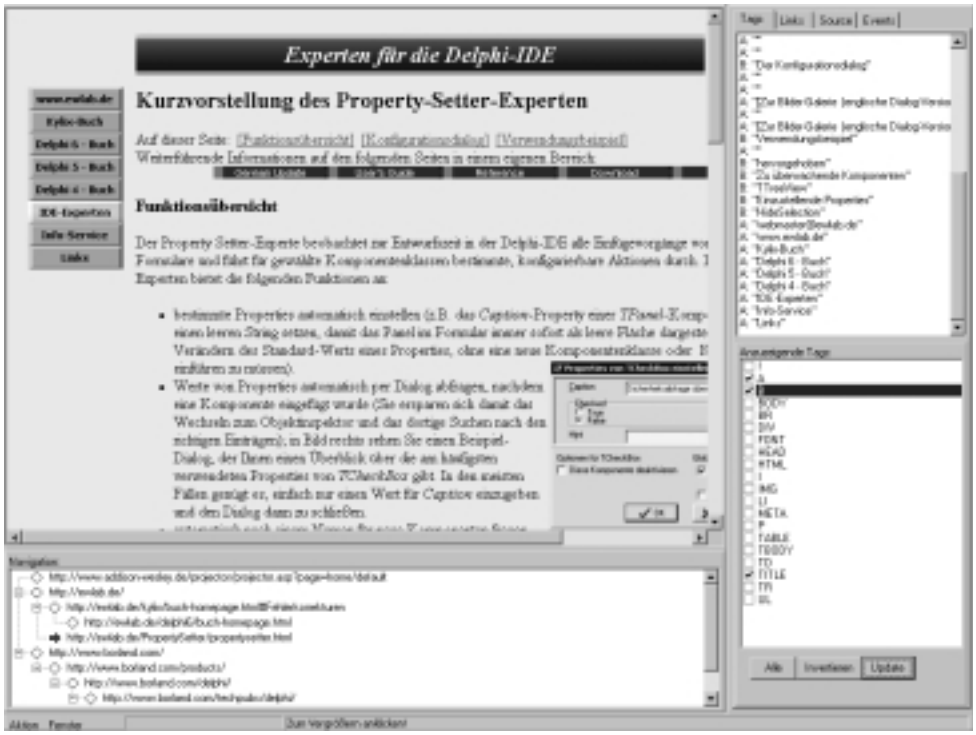


Abbildung 8.12: Ein in Delphi mit Hilfe des Internet-Explorer-Controls entwickelter Web-Browser mit Spezialfunktionen

Ein kostenloser Web-Browser

Eine der angenehmsten Eigenschaften des Internet Explorers ist, dass man den Bereich seines Fensters, in dem die HTML-Seiten angezeigt werden, ganz leicht in eigene Programme übernehmen kann. Zudem erlaubt Microsoft die freie Weitergabe des Internet Explorers mit eigenen Programmen (sofern man ihn als komplett installierbares Originalpaket weitergibt), so dass auch die Softwareentwickler hier ihren kostenlosen Web-

Browser gefunden haben (auf die nicht ganz uneigennützigen Motive von Microsoft, die hinter dieser kostenlosen Freistellung stecken, braucht wohl nicht näher eingegangen zu werden).

Die Typenbibliothek des Internet Explorers

Die Interface-Definitionen des Internet-Explorer-Controls wie z.B. die *IWebBrowser2*-Schnittstelle befinden sich in der DLL `shdocvw.dll`, die im System-Verzeichnis von Windows liegt. Weitere Interfaces für den Zugriff auf das HTML-Dokument gibt es in der Datei `mshtml.dll`. Auch diese werden im Beispielprogramm verwendet.

Wenn Sie die Standard- oder Personal-Version von Delphi 5 oder 6 oder eine ältere Delphi-Version (ab Delphi 3) verwenden, müssen Sie für die erstgenannte DLL den Menüpunkt KOMPONENTE | ACTIVEX IMPORTIEREN, für die letztgenannte dagegen PROJEKT | TYPBIBLIOTHEK IMPORTIEREN verwenden. Da beide DLLs bereits in der Systemregistrierung verzeichnet sein sollten, brauchen Sie wahrscheinlich nicht die Dateien selbst auszuwählen, sondern können aus den Listen der registrierten ActiveX-Controls bzw. Typenbibliotheken wählen. Sie werden dort unter den folgenden Titeln genannt:

- *Microsoft Internet Controls*, Version 1.1 oder höher, für `shdocvw.dll`
- *Microsoft HTML Object Library*, Version 4.0 oder höher, für `mshtml.dll`

Im Object-Pascal-Quelltext müssen dann die durch das Importieren erzeugten Units *ShDocVw_TLB* und *MsHtml_TLB* in die *uses*-Zeile der Unit eingebunden werden.

Die Originaldokumentation zu diesen Bibliotheken ist übrigens leider nicht im Lieferumfang von Delphi enthalten, aber von Microsoft als Teil des *Internet Client SDK* kostenlos zu beziehen bzw. online unter *msdn.microsoft.com/library* abrufbar (für ein kurzes Nachschlagen genügt bereits die Suchfunktion, etwa nach einer der hier verwendeten Methoden wie *IWebBrowser2.Navigate*).

Hinweis: Im Importieren-Dialog für die *HTML Object Library* ist es übrigens weniger empfehlenswert, die Option KOMPONENTEN-WRAPPER GENERIEREN anzuschalten, weil dies eine wahre Schwemme von Komponenten auslösen würde, die überhaupt nicht benötigt werden, da die direkte Verwendung der Interfaces hier effizienter ist.

Der ferngesteuerte Internet Explorer *R162*

Sie können auch den kompletten, in einem eigenen Fenster ablaufenden Internet Explorer über die COM-Automationsschnittstelle fernsteuern. Hierzu müssen Sie lediglich die in *ShDocVw* definierte Co-Klasse *CoInternetExplorer* instanziieren und erhalten dadurch eine *IWebBrowser2*-Schnittstelle, über die Sie alle Methoden und Pro-

perties verwenden können, die im Beispielprogramm dieses Kapitels für das Web-Browser-Control aufgerufen werden. Besonders wichtig ist das *Visible*-Property, das Sie erst einschalten müssen, damit der Explorer sichtbar wird:

```
uses ShDocVw;
...
var
  IE: IWebBrowser2;
begin
  IE := CoInternetExplorer.Create;
  IE.Navigate(WebBrowser.LocationURL, EmptyParam,
              EmptyParam, EmptyParam, EmptyParam);
  IE.Visible := True;
...
```

Im Folgenden werden wir uns jedoch nur mit dem im Fenster der Delphi-Anwendung integrierten Web Browser beschäftigen.

Der Internet Explorer im WebBrowser-Control *R54*

Bevor wir zu den speziellen Funktionen des Beispielprogramms kommen, sei hier kurz zusammengefasst, wie sich die Basisausstattung eines Browsers realisieren lässt. Als Erstes müssen Sie den Browser natürlich in ein Formular einfügen. Ab Delphi 5 Professional finden Sie auf der Seite *Internet* der Komponentenpalette den *WebBrowser*, in anderen Versionen und Ausgaben von Delphi müssen Sie ihn zuerst – wie oben beschrieben – als ActiveX-Control importieren. Die *WebBrowser*-Komponente ist im Gegensatz zu den in Abschnitt 8.6.2 verwendeten Word-Komponenten auch zur Laufzeit sichtbar, denn es handelt sich um ein vollwertiges ActiveX-Control. Im Beispielprogramm (Abbildung 8.12) wurde die WebBrowser-Komponente per *Align = alClient* auf dem Haupt-Panel des Formulars ausgerichtet.

Die weitere Implementierung findet im Object-Pascal-Code statt. Am wichtigsten dürfte wohl die Möglichkeit sein, eine bestimmte Web-Seite anzusteuern. Erwartungsgemäß bietet der WebBrowser dafür eine Methode an. Sie heißt *Navigate* und erwartet eine Reihe von Parametern, von denen wir hier nur den ersten verwenden, um die Adresse der Seite anzugeben. Da die restlichen Parameter ein wenig umständlich zu handhaben sind, verpackt das Beispielprogramm die *Navigate*-Methode des Browsers in eine eigene *Navigate*-Methode mit nur einem Parameter:

```
procedure TForm1.Navigate(URL: string);
begin
  // Die nach der URL folgenden var-Parameter Flags, TargetName,
  // PostData und Headers müssen angegeben werden, zumindest durch
  // die vordefinierte leere Variante EmptyParam:
  WebBrowser.Navigate(URL, EmptyParam, EmptyParam,
                           EmptyParam, EmptyParam);
end;
```

Diese Methode kann beispielsweise bei Programmstart dazu verwendet werden, eine Startseite aufzurufen. Das Beispielprogramm verzichtet übrigens auf ein Feld für die Eingabe einer neuen Adresse, erlaubt aber über das Menü und einen Eingabedialog, eine neue Seite aufzurufen.

Weitere wichtige Methoden von *TWebBrowser* sind:

- *GoBack* und *GoForward* rufen die vorherige bzw. nächste Seite in der History-Liste auf.
- *GoHome* und *GoSearch* gehen zu den entsprechenden im Einstellungsdialog des Internet Explorers festgelegten Seiten (zur Startseite bzw. zu der Seite, die die Suchfunktion bereitstellen soll).
- *Refresh* lädt das aktuelle Dokument neu. Alternativ können Sie auch *Refresh2* verwenden, deren Parameter die Gründlichkeit der Erneuerung angibt: 0 für einen normalen Refresh, 1 für einen bedingten normalen Refresh (Voraussetzung ist, dass die Gültigkeit der Seite abgelaufen ist) und 2 für einen Refresh unter Ausschluss von Cache-Speichern eventueller Proxy-Server.
- *Stop* bricht den aktuelle Ladevorgang ab.

Mit diesen Methoden ist es ein Leichtes, die entsprechenden Symbolleistenschalter des eigenständigen Internet Explorers in einer Delphi-Anwendung nachzubilden. Bei den Schaltern *Back* und *Forward* sollte allerdings zusätzlich berücksichtigt werden, ob sie überhaupt anwendbar sind. Falls nicht, sollte dies dem Benutzer durch eine Deaktivierung des Schalters veranschaulicht werden. Auch dies ist leicht zu realisieren, indem Sie das Event *OnCommandStateChange* der WebBrowser-Komponente in etwa wie folgt bearbeiten:

```
procedure TForm1.WebBrowserCommandStateChange(Sender: TObject;
  Command: Integer; Enable: WordBool);
begin
  if Command = CSC_NAVIGATEFORWARD then ForwardButton.Enabled := Enable;
  if Command = CSC_NAVIGATEBACK then BackButton.Enabled := Enable;
end;
```

(Im Beispielprogramm sind *ForwardButton* und *BackButton* keine Buttons, sondern Menüpunkte. Aufgrund der immer sichtbaren History-Liste braucht das Programm nämlich keine Vorwärts- und Rückwärtsschalter, so dass es diese Menüpunkte nur zur Demonstration implementiert.)

Ansonsten brauchen Sie sich meist nicht viel um die Navigation zu kümmern. Klickt der Benutzer einen Hyperlink an, so ruft das WebBrowser-Control von selbst die entsprechende Seite auf. Einige später noch zu besprechende Events geben Ihnen hier allerdings weitere Eingriffsmöglichkeiten.

Die aktuell geladene Web-Seite können Sie übrigens über zweierlei Properties abfragen:

- *LocationName* gibt den Titel der Webseite an, der auch in der Titelzeile des Browser-Fensters angezeigt werden würde.
- *LocationURL* gibt die Adresse in einem ausführlichen Standardformat an (also z.B. mit *http*-Präfix).

Ansonsten verfügt das WebBrowser-Control nur über wenige interessante Properties wie etwa *Busy* (»lade gerade eine Web-Seite«) oder *Silent* (keine Meldungsfenster öffnen). Das Property *Document* wird uns später in die Interna des HTML-Dokuments führen.

Von den Events werden im Folgenden die wichtigsten erläutert, welche auch im Beispielprogramm bearbeitet werden (für *OnCommandStateChange* siehe die obige Beispielmethode).

Events für die Aktualisierung der Fensteroberfläche

Zwei Events ermöglichen Ihrer Anwendung, die Statuszeile des Internet Explorers nachzubilden: Von *OnStatusTextChange* erfahren Sie den Text, den auch der Internet Explorer in der Statuszeile anzeigen würde; von *OnProgressChange* die Position des Fortschrittsbalkens. Sie benötigen also nicht mehr als eine *TStatusBar*-Komponente mit *SimpleText* = *True* und eine *TProgressBar*-Komponente, um dann in diesen beiden Events deren Properties wie folgt auf die Parameter der Events zu setzen:

```
StatusBar.SimpleText := Text; // in OnStatusTextChange
ProgressBar.Position := Progress; // in OnProgressChange
ProgressBar.Max := ProgressMax;
```

In ähnlicher Weise gibt es das Event *TitleChange* für die Aktualisierung der Titelleiste; außerdem *OnToolBar*, *OnMenuBar* und *OnStatusBar*, die Ihnen einige aus dem HTML-Dokument begründete Gelegenheiten mitteilen, bei denen der Internet Explorer seine Toolbar, Menüzeile oder Statuszeilen verstecken würde (z.B. weil entsprechende Anweisungen in einem Script ausgeführt werden) und Sie dies in Ihrem Browser ebenfalls tun könnten. Schließlich gibt es noch die Events *OnVisible*, *OnFullScreen* und *OnTheaterMode*, die sich auf das (mögliche) Verhalten des gesamten Fensters beziehen.

Events für die Überwachung der Navigation

Zum Eingreifen in die Navigation gibt Ihnen das WebBrowser-Control vielfältige Möglichkeiten. Sie können z.B. vor dem Ansteuern einer Seite die Daten einer http-Post-Transaktion ergänzen, das Ansteuern der Seite unterbinden oder bestimmen, dass die Seite entgegen der Voreinstellung in ein neues Fenster (oder gerade nicht) geladen

werden soll, und Sie erhalten natürlich auch Nachricht, wenn irgendwelche Navigationsvorgänge abgeschlossen sind.

- *OnBeforeNavigate2* tritt vor jedem Navigationsvorgang auf – also nicht nur vor dem Laden einer neuen Seite oder eines neuen Frames, sondern auch vor dem Hyperlink-gesteuerten Springen innerhalb einer Seite.
- Wenn der Navigationsvorgang in ein anderes Fenster führt, wird zunächst ein *OnNewWindow2* gemeldet; im WebBrowser-Control des neuen Fensters gibt es dann ein *OnBeforeNavigate2*.
- Am Ende eines Navigationsvorgangs steht grundsätzlich das Ereignis *OnNavigateComplete2*.
- Nach dem Laden von neuen Seiten oder Frames gibt es zusätzlich das Ereignis *OnDocumentComplete*.
- Download-Vorgänge werden durch die Events *OnDownloadBegin* und *OnDownloadComplete* eingerahmt, dazu zählen auch der Beginn und das Ende des Downloads von Seiten.

Im Folgenden werden als Beispiel die Methoden für die Events *NewWindow2* und *DocumentComplete* aus dem Beispielprogramm gezeigt.

OnDocumentComplete

Das Ereignis *DocumentComplete* eignet sich gut für Aufgaben, die nach jedem abgeschlossenen Seitenaufbau durchgeführt werden sollen. Das Beispielprogramm nimmt dann etwa die neue URL in das Navigations-TreeView auf und aktualisiert die verschiedenen Listen, die auf der rechten Seite über ein PageControl aufgeschlagen werden können.

Zu beachten ist allerdings, dass *DocumentComplete* auch dann auftritt, wenn das Laden eines Frames der aktuellen Seite abgeschlossen wurde. Um nur die Ereignisse herauszufiltern, die sich auf das komplette Dokument beziehen, inspizieren wir den Parameter *pDisp*. Dieser zeigt auf die *IDispatch*-Schnittstelle des Fensters, in dem das geladene Dokument angezeigt wird.

Wenn sich das Ereignis nur auf ein untergeordnetes Frame bezieht, weist *pDisp* auf ein Fensterobjekt, das dem WebBrowser-Control untergeordnet ist. Bezieht es sich auf die gesamte Seite, weist *pDisp* auf die *IDispatch*-Schnittstelle des WebBrowser-Controls. Die *TWebBrowser*-Komponente macht diese Schnittstelle über ihr Property *ControlInterface* zugänglich. Stimmt diese also mit dem Event-Parameter *pDisp* überein, ist die gesamte Seite mit allen eventuellen Unter-Frames fertig geladen:

```
procedure TForm1.WebBrowserDocumentComplete(Sender: TObject;
  const pDisp: IDispatch; var URL: OleVariant);
```

```
begin
  if (pDisp = WebBrowser.ControlInterface)
  then begin
    NavLog(URL); // Hinzufügen der neuen Seite zum Log-TreeView
    HTMLScan;    // Anzeigen der Infos im rechten Fensterbereich
  end;
end;
```

OnNewWindow2

Das wichtigste Event für eine Anwendung, die ein echter Web-Browser-Ersatz sein will, ist *NewWindow2*. Es tritt vor dem Öffnen eines neuen Browser-Fensters auf, beispielsweise wenn der Benutzer einen Link explizit in einem solchen öffnen will (über den Popup-Menüpunkt IN NEUEM FENSTER ÖFFNEN oder über das Anklicken des Links in Verbindung mit Shift). Wenn Sie dieses Ereignis nicht bearbeiten, öffnet das Web-Browser-Control nämlich eine neue Instanz des kompletten Internet Explorers, und dieser bringt ja seine eigene Benutzerschnittstelle mit. Durch Bearbeiten von *NewWindow2* können Sie dies verhindern und auch für das neue Browser-Fenster Ihre eigene Anwendung einsetzen.

Das Beispielprogramm erzeugt für das neue Fenster einfach eine weitere Instanz des Hauptformulars. Da sich in diesem Formular ja eine *TWebBrowser*-Komponente befindet, wird auf diese Weise automatisch ein neues WebBrowser-Control erzeugt. Die über *DefaultInterface* abfragbare Schnittstelle dieses Controls wird über den *OnNewWindow2*-Parameter *ppDisp* an das alte WebBrowser-Control zurückgegeben. Daran erkennt dieses, dass es sich nicht selbst um die Erzeugung eines neuen Controls kümmern muss, und verzichtet darauf, eine neue Internet-Explorer-Instanz anzulegen.

```
procedure TForm1.WebBrowserNewWindow2(Sender: TObject;
  var ppDisp: IDispatch; var Cancel: WordBool);
var
  NewWindow: TForm1;
 begin
  if  MessageDlg('Öffnen von neuem Fenster erlauben?',
       mtConfirmation, [mbYes, mbNo], 0) = mrYes then
  begin
    NewWindow := TForm1.Create(nil);
    NewWindow.Show;
    NewWindow.IsSecondaryWindow := True;
    NewWindow.WebBrowser.OnNewWindow2 := WebBrowserNewWindow2;
    ppDisp := WebBrowser.DefaultInterface;
  end else
    Cancel := True;
end;
```

Wichtig ist auch, dass in dem WebBrowser-Control, das über *ppDisp* zurückgegeben wird, noch keine Seite angezeigt wird, sonst wird es vom alten Control nicht angetastet und die neue Seite wird nicht geladen. Das Beispielformular lädt normalerweise beim

OnShow-Ereignis eine Startseite. Dies muss für Fenster, die bei *OnNewWindow2* geöffnet werden, jedoch unterbunden werden (also für alle Fenster außer dem Hauptfenster). Damit das Formular sein Verhalten anpassen kann, falls es nur ein solches »Zweitfenster« ist, besitzt es im Beispielprogramm eine Variable *IsSecondaryWindow*, die im obigen Code-Auszug auf *True* gesetzt wird.

Als weitere Maßnahme setzt der gezeigte Code noch das *OnNewWindow2*-Ereignis des neuen Fensters auf die Bearbeitungsmethode des alten Fensters. In diesem Fall hat das noch keine konkrete Auswirkung, es wäre aber denkbar, dass dieses (oder auch andere) Ereignisse von Zweitfenstern alle zentral im Hauptfenster bearbeitet werden sollen (z.B. um dort ein für alle Fenster geltendes Navigations-Logbuch zu führen).

Die obige Methode demonstriert außerdem noch, dass man das Öffnen eines neuen Fensters auch abbrechen kann. Durch Setzen von *Cancel* auf *True* wird die Seite, die eigentlich angezeigt werden sollte, gar nicht angezeigt (also auch nicht im »alten« WebBrowser-Control).

Das Abbrechen einer Navigation beim *BeforeNavigate2*-Event funktioniert ebenfalls über das Setzen eines *Cancel*-Flags. Das Beispielprogramm führt zur Demonstration unter *Links* eine Liste mit allen Seiten, die nicht geladen werden sollen, und bearbeitet das *BeforeNavigate2*-Event entsprechend.

Anmerkung: Die Intention, im Beispielprogramm den kompletten Navigationsverlauf in einem TreeView anzuzeigen, wird durch das Öffnen weiterer Browser-Fenster eingeschränkt, denn diese verfügen über ihr eigenes Navigations-Logbuch.

Wünschenswert wäre, dass das Beispielprogramm alle Vorgänge in nur einem TreeView anzeigt, was z.B. dadurch realisiert werden könnte, dass für ein neues Browser-Fenster keine neue Instanz des Hauptformulars angelegt, sondern ein neues WebBrowser-Control in das Hauptfenster eingefügt wird. Zum Umschalten zwischen mehreren Browser-Controls könnte z.B. eine TabControl-Komponente dienen.

8.6.4 Das HTML-Dokument

Mit den bisher besprochenen Properties und Methoden der WebBrowser-Komponente kommen wir im Beispielprogramm *WebCollector* noch nicht besonders weit, aber *TWebBrowser* (bzw. die dahinter stehende Schnittstelle *IWebBrowser2*) hat noch ein weiteres wichtiges Property, das als Tor in eine ganz andere Welt führt: in die Welt des im Browser angezeigten Dokuments.

Das Property heißt erwartungsgemäß *Document* und ist vom Typ *IDispatch*. Der WebBrowser kann nicht nur HTML-Dokumente anzeigen, sondern ist als ActiveX-Container auch zur Beherbergung anderer Anwendungen wie z.B. Word in der Lage.

Demzufolge kann sich hinter *Document* nicht nur ein HTML-Dokument, sondern auch das Dokument einer solchen Anwendung verbergen, also z.B. ein Dokument der schon in Abschnitt 8.6.2 verwendeten Word-Dokumentklasse. *Document* ist damit gewissermaßen ein polymorphes Property, dessen genauer Typ erst zur Laufzeit festgelegt wird. Wir setzen hier jedoch voraus, dass nur HTML-Dokumente geladen werden. Dies bedeutet, dass das *Document*-Property die Interfaces *IHTMLDocument* und *IHTMLDocument2* unterstützt.

Grundsätzliches zu MSHTML

Wenn Sie sich in die Programmierschnittstelle von *MSHTML* einarbeiten wollen, lassen Sie sich nicht durch die Pascal-Datei, die Delphi aus der Typenbibliothek von `mshtml.dll` erzeugt, abschrecken. Allein der Interface-Teil dieser Unit enthält, wenn Sie die Option KOMPONENTEN-WRAPPER GENERIEREN weglassen, ca. 30.000 Zeilen; mit Komponenten-Wrapper wächst die Gesamt-Unit unter Delphi 6 auf über 300.000 Zeilen. Auch wenn man bedenkt, dass im Interface-Teil viele Methoden doppelt deklariert werden, einmal als *interface* und einmal als *dispinterface*, und manche noch ein drittes Mal als Methode einer Hüllkomponente, bleibt es bei einem sehr großen Angebot verschiedener Schnittstellenelemente.

Jedoch handelt es sich bei diesen Methoden und Klassen zu einem großen Teil nicht um neue Erfindungen, sondern diese Schnittstellen bilden lediglich das Dynamic HTML-Objektmodel des Internet Explorers nach, das Sie auch über JavaScript im HTML-Code steuern können. Wenn Sie also JavaScript kennen, werden Sie wahrscheinlich einen leichten Einstieg in die Programmierung mit *MSHTML* haben. Grob gesagt kann man mit *MSHTML* nämlich all das machen, was man sonst in Dynamic HTML programmieren kann.

Das Grundprinzip zur Strukturierung der DHTML-Programmierschnittstelle ist die Schachtelung von Objekten in einer Besitzhierarchie. So gibt es etwa das aus DHTML bekannte *window*-Objekt, welches das Fenster repräsentiert, in dem eine Web-Seite angezeigt wird. Es enthält ein *document*-Objekt für das Dokument, eine *frames*-Kollektion für alle eventuell enthaltenen Frames und verschiedene Hilfsobjekte wie *history* für die Adress-History-Liste oder *screen* für die Abfrage von Informationen über den verwendeten Bildschirmmodus. Das *document*-Objekt enthält alle Elemente des HTML-Dokuments und jedes HTML-Elementobjekt wiederum eine Kollektion seiner Kindelemente.

Eine Vererbungshierarchie, wie sie von Delphis VCL bekannt ist, gibt es im DHTML-Objektmodell und im Internet Explorer nicht – zumindest ist diese nicht nach außen sichtbar, denn im COM gibt es nur die Vererbung von Schnittstellen (Interfaces), aber keine Vererbung von Implementierungen (Co-Klassen).

Zugriff auf das HTML-Dokument R165

Wie schon die Word-Programmierung in Kapitel 8.6.2, so erfordert auch die Programmierung im DHTML-Objektmodell (JavaScript) eigene Bücher, so dass sich die folgende Darstellung auf einige Beispiele für die Verwendung dieses Objektmodells aus einer Delphi-Anwendung heraus beschränken wird. Bei den Code-Auszügen, die dem Beispielprogramm *WebCollector* entnommen sind, wird uns unter anderem wieder der Unterschied zwischen *OleVariant*-Variablen und Interface-Zeigern begegnen, der schon beim Übergang von Kapitel 8.6.1 auf 8.6.2 genauer erläutert wurde.

Alle wichtigen, im Folgenden verwendeten Elemente des *Document*-Objekts sind Interface-Variablen:

- *All* - eine Kollektion, die Zugriff auf alle HTML-Elemente des Dokuments gibt. Mit ihrer Methode *Tags* können Sie die Elemente mit einem von Ihnen gewählten *Tag* herausfiltern (das Ergebnis dieser Methode ist also eine neue Kollektion).
- *Links* - eine Kollektion aller Hyperlinks des Dokuments.
- *Frames* - eine Kollektion aller in dem Dokument enthaltenen Frames, falls das Dokument ein Frame-Set ist.
- *Selection* - ein Objekt, das entweder den markierten Bereich eines Dokuments angibt oder einen Punkt, der als Einfügemarkierung dient.
- *Body* - das HTML-Element, das dem *Body*-Tag des HTML-Quelltextes entspricht.

Auf die Typen dieser Interfaces (*IHTMLElementCollection*, *IHTMLFrameCollection2*, *IHTMLSelection* und *IHTMLElement*) gehen die folgenden Erläuterungen bei Bedarf ein.

Von den einzelnen HTML-Elementen, d.h. von der *IHTMLElement*-Schnittstelle, benötigen wir die Eigenschaften *InnerText*, *TagName* und *HRef*, außerdem am Ende des Kapitels in Verbindung mit dem *Body*-Element und mit Textbereichs-Objekten die Methode *insertAdjacentHTML*.

Zunächst einmal ist es eine einfache Übung, Zugriff auf das Dokument-Objekt zu erhalten. Hier sollte man lediglich den »abstrakten« Datentyp *IDispatch*, den das *Document*-Property im Browser hat, in die *IHTMLDocument2*-Schnittstelle konvertieren (sofern man sich sicher ist, dass es sich beim Dokument um ein HTML-Dokument handelt):

```
// var Doc: IHTMLDocument2;
Doc := WebBrowser.Document as IHTMLDocument2;
```

Listen von Hyperlinks und HTML-Tag-Arten

Das Beispielprogramm soll alle im Dokument befindlichen Hyperlinks in der Listbox *LBLinks* auflisten, und zwar jeweils mit Angabe der Zieladresse. Um alle Links zu finden, genügt es, die Kollektion *Links*, die das *Document*-Objekt als Property zur Verfügung stellt, zu durchlaufen. Die Kollektion selbst wird über die Schnittstelle *IHTMLElementCollection* angesprochen. So erhält man mit *Links.item(0, varEmpty)* z.B. das erste Element der Kollektion.

Die Elemente der *Links*-Kollektion können auf verschiedene HTML-Tags zurückzuführen sein, z.B. auf *LINK*- und *AREA*- und *A*-Tags. Zwar implementiert in diesem Objektmodell grundsätzlich jedes HTML-Element das allgemeine *IHTMLElement*-Interface, darüber hinaus unterstützen die Elemente von verschiedenen Tags aber auch verschiedene spezialisierte Schnittstellen – bei den genannten Tags sind dies etwa *IHTMLLinkElement*, *IHTMLAreaElement* und *IHTMLAnchorElement*. Alle drei verfügen über ein Property namens *HRef*, das die Zieladresse enthält, die in der Listbox dargestellt werden soll.

JavaScript-Programmierer werden gewohnt sein, sich gar nicht um den genauen Elementtyp kümmern zu müssen, sondern einfach so auf das *HRef*-Property zugreifen zu können. Dies ist auch in Delphi möglich, und der folgende Code-Auszug demonstriert die Unterschiede:

```
// (aus TForm1.ScanLinks)
// var
//   Links: IHTMLElementCollection;
//   AnchorElement: IHTMLAnchorElement;
//   AnElementVar: OleVariant;
Links := Doc.Links;
Len := Links.Length;
for i := 0 to Len-1 do begin
  try
    AnchorElement := Links.Item(i, varEmpty) as IHTMLAnchorElement;
    VisibleText := (AnchorElement as IHTMLElement).InnerText;
    LBLinks.Items.Add(AnchorElement.HRef+' ('+VisibleText+')');
  except
    try
      AnElementVar := Links.Item(i, varEmpty);
      LBLinks.Items.Add('Kein IHTMLAnchorElement für: '
              + AnElementVar.TagName+', ' + AnElementVar.HRef);
    except
    end;
  end;
end;
```

Links.Item liefert für die einzelnen Elemente lediglich *IDispatch*-Schnittstellen zurück. Sofern das Element die *IHTMLAnchorElement*-Schnittstelle unterstützt, ermittelt das Programm einen typisierten Zeiger darauf (*AnchorElement*). Wird die Schnittstelle nicht

unterstützt, kommt es bei der Anwendung des *as*-Operators zu einer Exception. Das Programm könnte nun versuchen, eine *IHTMLAreaElement*-Schnittstelle zu erhalten, geht aber einen anderen Weg: Es speichert den von *Links.Item* zurückgelieferten *IDispatch*-Zeiger in einer *OleVariant*-Variablen (*AnElementVar*). Über diese kann es ohne Typumwandlung auf die Properties *HRef* und *TagName* (Name des HTML-Tags) zugreifen, allerdings wird dann erst zur Laufzeit überprüft, ob das Element diese Properties wirklich unterstützt. Für die Arbeit in Delphis Editor bedeutet dies, dass die Programmierhilfen bei der Variablen *ElementVar* nicht zur Verfügung stehen – im Gegensatz zur Variablen *AnchorElement*.

Auch an anderen Stellen verwendet *WebCollector* einen Kompromiss zwischen früher und später Bindung:

- Element-Kollektionen können problemlos früh gebunden und über das *IHTMLElementCollection*-Interface angesprochen werden.
- Die einzelnen Elemente lassen sich, wenn man ihre Properties schon kennt und die Programmierhilfe nicht benötigt, einfacher als *OleVariant* ansprechen, denn dann spart man sich die Typumwandlung.

Im Folgenden wird das *TagName*-Property ausgelesen, um eine Liste aller im Dokument vorkommenden HTML-Tags zu bilden:

```
// Elements: IHTMLElementCollection;
// AnElement: OleVariant;
Elements := Doc.All;
for i := 0 to Elements.Length - 1 do
  Tags.Add(Elements.Item(i, varEmpty).TagName);
```

Die Liste der Tags wird im Beispielprogramm in eine *CheckListBox* übertragen, in der Sie auswählen können, welche Tags in einer zweiten Liste angezeigt werden sollen (Abbildung 8.12). Von jedem Element werden *TagName* und *InnerText* angezeigt. Der entsprechende Code birgt keine Neuigkeiten, verbleibt also unabgedruckt auf der CD-ROM.

Durchsuchen von Dokumenten mit Frames *R70*

Die oben gezeigten Code-Auszüge weisen noch einen wesentlichen Mangel auf: Sie berücksichtigen keine Frames. Wenn eine HTML-Seite aus Frames besteht, enthält das *Document*-Objekt nämlich nur die Elemente des den Frames übergeordneten Dokuments. Im Beispielprogramm wäre es aber durchaus sinnvoll, dass die Listen im rechten Fensterbereich alle auf der gesamten Seite vorkommenden Elemente berücksichtigen. Dazu ist es erforderlich, jedes Frame einzeln zu durchsuchen.

Eine Liste der Frames ist schnell gefunden. Sie befindet sich in *Document.Frames*. Jedes Element dieser Liste ist selbst ein Dokument mit einer *IHTMLDocument*-Schnittstelle

und eventuell weiteren Unter-Frames. Die folgende Methode *CollectFrames* sammelt rekursiv alle Frames des Dokuments (inklusive des Gesamtdokuments) und ordnet sie in der Liste *FAllFrames* ein (diese ist in der Formularklasse deklariert und hat die VCL-Klasse *TList*, hat also nichts mehr mit *MSHTML* zu tun):

```
procedure TForm1.CollectFrames;
  procedure CollectFor(Doc: IHTMLDocument2);
  var
    i, Len: Integer;
    SubDoc, Param: OLEVariant;
    SubDocDispatch: IDispatch;
    Frames: IHTMLFramesCollection2;
  begin
    AddFrame(Doc); // (zu AddFrame siehe nächster Abschnitt)
    Frames := Doc.Frames;
    Len := Doc.Frames.Length;
    for i := 0 to Len-1 do begin
      Param := i;
      SubDoc := Doc.Frames.item(Param).Document;
      if VarType(SubDoc) = varDispatch then begin
        SubDocDispatch := IDispatch(TVarData(SubDoc).VDispatch);
        CollectFor(SubDocDispatch as IHTMLDocument2);
      end;
    end;
  end;
var
  Doc: IHTMLDocument2;
begin
  ClearFrameList;
  Doc := WebBrowser.Document as IHTMLDocument2;
  if Assigned(Doc) then
    CollectFor(Doc);
end;
```

Eine Besonderheit in dieser Methode ist die Typumwandlung, mit der die *IHTMLDocument*-Schnittstelle eines Frames ermittelt wird. Die Methode *Frames.item* liefert ihr Ergebnis nämlich als *OleVariant*. Über diese könnten wir ebenfalls alle Methoden von *IHTMLDocument* aufrufen, allerdings dann wieder nur über die späte Bindung, ohne Typüberprüfung des Compilers. Damit dies funktioniert, muss die *OleVariant* jedoch eine *IDispatch*-Schnittstelle enthalten. Und wenn diese Schnittstelle zu dem HTML-Dokumentobjekt gehört, das wir suchen, lässt sie sich mit dem *as*-Operator (intern: *IUnknown.QueryInterface*) in ein *IHTMLDocument2* umwandeln. Vorher benötigen wir jedoch Zugriff auf die *IDispatch*-Schnittstelle, wofür wir die *OleVariant* als *TVarData* darstellen müssen, um ihr *VDispatch*-Feld auslesen zu können:

```
SubDocDispatch := IDispatch(TVarData(SubDoc).VDispatch);
SubDoc := SubDocDispatch as IHTMLDocument2;
```

Um die zuvor gezeigten Code-Auszüge auf die Berücksichtigung von Frames umzustellen, genügt es nun einfach, Schleifen »um sie herum zu bauen«, in denen die Liste der Frames durchlaufen wird. Innerhalb dieser Schleife wird noch die Zuweisung der Dokument-Schnittstelle geändert:

```
// statt
Doc := WebBrowser.Document as IHTMLDocument2;
// heißt es nun:
Doc := Frame[frameIndex]; { Frame ist ein Property, das Zugriff auf die
  oben mit AddFrame in eine Liste eingefügten IHTMLDocument2-Interfaces
  gibt }
```

Ansonsten bleibt der Code unverändert. Ein komplettes Bild der Methoden können Sie sich über die CD-ROM verschaffen.

Verwaltung einer Interface-Liste *R135*

Eine Besonderheit verbirgt sich noch in der oben aufgerufenen Methode *AddFrame*, die im Folgenden abgedruckt ist:

```
type
  TFrame = class
  private
    FDoc: IHTMLDocument2;
  public
    constructor Create(Adoc: IHTMLDocument2);
    property Doc: IHTMLDocument2 read FDoc;
  end;

constructor TFrame.Create(Adoc: IHTMLDocument2);
begin
  inherited Create;
  FDoc := ADoc;
end;

procedure TForm1.AddFrame(Doc: IHTMLDocument2);
var
  frameWrapperObj: TFrame;
begin
  frameWrapperObj := TFrame.Create(Doc);
  FAllFrames.Add(frameWrapperObj);
end;
```

Ziel des Programms ist es ja, eine Liste von *IHTMLDocument2*-Zeigern zu speichern. Ein *IHTMLDocument2*-Zeiger muss aber, damit die automatische Referenzzählung der COM-Objekte funktioniert, auch in einer Variablen des Typs *IHTMLDocument2* abgespeichert werden. Eine direkte Speicherung in einer *TList* ist nicht möglich, da dies nur unter Umwandlung in einen *Pointer* funktionieren würde. Dies würde aber die Referenzzählung umgehen, und die *MSHTML*-Bibliothek wüsste nichts davon, dass das

Beispielprogramm noch eine Referenz auf diese Schnittstelle aufbewahrt. Also wird jeder *IHTMLDocument2*-Zeiger in ein Objekt der Klasse *TFrame*[22] eingebettet, und nur Objekte dieser Klasse werden in die Liste *FAllFrames* eingefügt. Der vom Compiler automatisch erzeugte Code sorgt dafür, dass der Referenzzähler eines *IHTMLDocument2* erhöht wird, wenn *TFrame.Create* die Zuweisung ausführt, und erniedrigt, wenn das *TFrame*-Objekt freigegeben wird.

Diese Freigabe findet wie folgt statt:

```
procedure TForm1.ClearFrameList;
var
  i: integer;
  frameWrapperObj: TFrame;
begin
  for i := FAllFrames.Count-1 downto 0 do begin
    frameWrapperObj := TFrame(FAllFrames[i]);
    frameWrapperObj.Free;
    FAllFrames.Delete(i);
  end;
end;
```

Die Referenzzählung hindert den Internet Explorer übrigens nicht daran, alle mit einer Seite zusammenhängenden Objekte freizugeben, wenn eine neue Seite geladen wird.

Bemerkung: In diesem Beispielprogramm ist es natürlich nicht unbedingt notwendig, eine *IHTMLDocument2*-Liste zu speichern, sondern alle Methoden, die den Inhalt der Seite untersuchen, könnten von den erwähnten Schleifen auf Rekursion umgestellt werden – nach dem Beispiel der Methode *CollectFrames*.

Markierte Dokumentbereiche extrahieren *R69*

Als Letztes soll dieser Abschnitt die Programmfunktion vorstellen, die die Fähigkeit zur dynamischen *Veränderung* der HTML-Seite demonstriert. Grundsätzlich können Sie jedes HTML-Element über Object-Pascal-Code anpassen, indem Sie seine inhaltsbezogenen Properties nicht auslesen, sondern beschreiben. Dabei haben Sie die Wahl, nur ein einzelnes Attribut, wie etwa *Body.Background* oder *AnchorElement.HRef*, oder das gesamte Element auf einmal zu verändern (*InnerText* für Text zwischen den begrenzenden Tags, *InnerHtml* für Text mit zusätzlichen Tags, *OuterHtml* für den gesamten HTML-Quelltext des Elements mit äußeren Tags). Eine andere Möglichkeit der Veränderung einer Seite sind die *insertAdjacent*...-Methoden der HTML-Elemente.

22 Nicht zu verwechseln mit Delphis Frame-Klasse für den Formular-Entwurf.

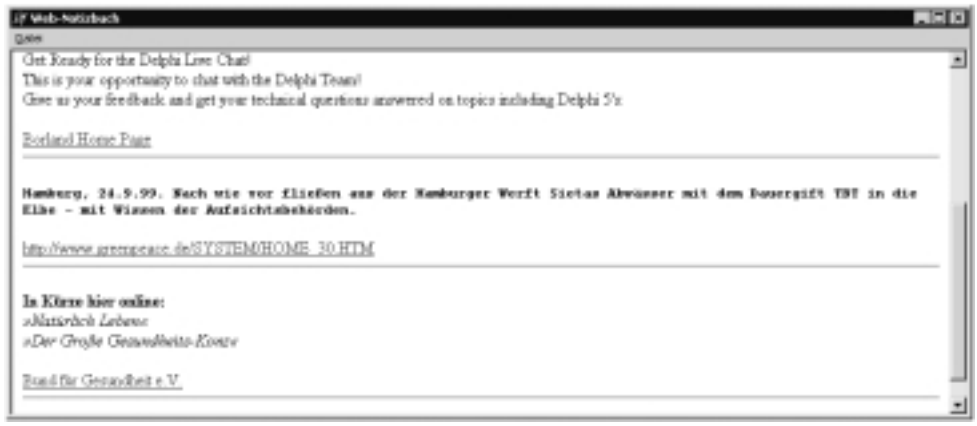


Abbildung 8.13: Ausschnitte aus verschiedenen Web-Seiten mit insertAdjacentText in ein leeres Dokument eingefügt

Das Beispielprogramm beschränkt sich darauf, mit der Methode *insertAdjacentText* Text am Anfang eines leeren Dokuments einzufügen, und zwar einen Text, der aus zusammengefügten *Notizen* besteht. Eine »Notiz« ist dabei ein Auszug aus einer HTML-Seite, die vom Benutzer markiert und per Menübefehl als »Notiz« gespeichert wurde (Menüpunkt MARKIERTEN BEREICH NOTIEREN).

Abbildung 8.13 zeigt das Fenster, das bei Anwahl des Menüpunkts NOTIZEN ANZEIGEN erscheint. Es basiert auf einem zweiten Formular des Beispielprogramms, das ein weiteres *TWebBrowser*-Steuerelement enthält. In dieses lädt es, wenn es durch die Methode *ShowHTML* angezeigt wird, eine leere HTML-Seite `emptytemplate.htm`, die sich im Verzeichnis des Programms befinden muss:

```
procedure TCollectionForm.ShowHTML(htmltext: String);
var
  Flags, TargetName, PostData, Headers: OleVariant;
begin
  WebBrowser.Navigate(BackSlash(ExtractFilePath(Application.ExeName))+
       'emptytemplate.htm', Flags, TargetName, PostData, Headers);
  TextToInsert := htmlText;
  Show;
end;
```

Den anzuzeigenden Text (*htmltext*) kann es noch nicht direkt in das WebBrowser-Control laden, da dieses ja erst die leere Seite laden muss. Also gilt es, den Text zwischenzuspeichern (*TextToInsert*) und das Ereignis *OnDocumentComplete* abzuwarten. Wenn dies eintritt, wird der Text mit *insertAdjacentHTML* am Beginn des *Body*-Tags der leeren Seite eingefügt (`emptytemplate.htm` ist also nicht wirklich leer, sondern enthält als Platzhalter ein leeres *Body*-Tag):

```
procedure TCollectionForm.WebBrowserDocumentComplete(Sender: TObject;
  const pDisp: IDispatch; var URL: OleVariant);
var
  Doc: IHTMLDocument2;
begin
  if (pDisp = WebBrowser.ControlInterface) then begin
    Doc := WebBrowser.Document as IHTMLDocument2;
    Doc.body.insertAdjacentHTML('afterBegin', TextToInsert);
  end;
end;
```

Wie erwähnt werden die Notizen im HTML-Format gespeichert. *TextToInsert* kann also beliebige HTML-Tags enthalten, die sich irgendwie durch den Benutzer im WebBrowser-Control markieren lassen. Interessant ist in diesem Zusammenhang noch, wie der markierte Bereich aus diesem Control ausgelesen werden kann.

Die Methode *TakeNoteClick* ist mit dem Menüpunkt MARKIERTEN BEREICH NOTIEREN verknüpft und gelangt auf etwas komplizierten Wegen zu einem *IHTMLTxtRange*-Objekt, von dem es den markierten Text abfragt. Wenn gar kein Text markiert ist, zeigt sich das daran, dass *Selection.type* nicht den Wert *'Text'* zurückliefert. Allerdings muss auch hier wieder bedacht werden, dass ein Dokument aus Frames bestehen kann. *TakeNoteClick* muss also eventuell alle Frames nach einer Markierung durchsuchen:

```
procedure TForm1.TakeNoteClick(Sender: TObject);
var
  Doc: IHTMLDocument2;
  Range: IHTMLTxtRange;
  frameindex: Integer;
  DocV: OLEVariant;
  FoundSelection: Boolean;
begin
  Doc := WebBrowser.Document as IHTMLDocument2;
  SO := Doc.Selection;
  if Doc.Selection.Type_ = 'Text' then begin
    Range := Doc.Selection.createRange as IHTMLTxtRange;
    Notizen := Notizen + Range.htmlText;
    FoundSelection := True;
  end else begin
    for frameindex:=0 to FrameCount-1 do begin
      DocV := Frame[frameindex];
      if DocV.Selection.type = 'Text' then begin
        Notizen := Notizen + DocV.Selection.createRange.htmlText;
        FoundSelection := True;
      end;
    end;
  end;
  if FoundSelection then begin
    Notizen := Notizen + '<P><A HREF="' + WebBrowser.LocationURL + '">'
               + WebBrowser.LocationName + '</A>' +
               '<HR WIDTH="100%"><P>';
```

```
    end else
      ShowMessage('Markieren Sie bitte zuerst einen Bereich!');
  end;
```

Im letzten *if*-Block wird jede neue Notiz zusätzlich noch mit einem Hyperlink auf die Quellseite und einer Abschlusslinie versehen (wie in Abbildung 8.13 gezeigt). Dazu wird der String *Notizen* einfach um den entsprechenden HTML-Code erweitert.

8.7 COM-Automations-Server

Während Automations-Clients – zumindest solche der in Kapitel 8.6.1 beschriebenen Art – in allen Ausgaben von Delphi 6 entwickelt werden können, sind die Automations-Server seit Delphi 3 eigentlich nicht mehr in der Personal / Standard-Ausgabe vorgesehen. Da erst die Professional-Version einen Typenbibliotheks-Editor enthält, können Sie die Beispiele dieses Kapitels zwar mit der Personal-Version kompilieren, aber nicht sinnvoll erweitern oder nachbauen.

Hinweis zu Delphi Personal/Standard (3 – 6) und Delphi 2: Auf der CD zu diesem Buch finden Sie in der Delphi-2-Version des TreeDesigners jedoch auch ein Beispiel für einen Automations-Server, der noch im Delphi-2-Stil ohne Typenbibliotheken auskommt und auf diese Weise zeigt, wie Automations-Server auch mit der Personal / Standard-Version entwickelt werden können. Erläuterungen dazu finden Sie ebenfalls auf der CD.

8.7.1 Implementieren eines Automations-Objekts

Verglichen mit dem *Steuern* eines Automations-Servers müssen Sie erheblich mehr Aufwand betreiben, um einen eigenen Server zu schreiben. Für sich betrachtet ist jedoch auch das mit Delphi eine ziemlich einfache Aufgabe, denn Delphi stellt hierfür ab der Professional-Version einen Automatisierungsobjekt-Experten zur Verfügung. Unter anderem mit der Hilfe dieses Experten wird der TreeDesigner in diesem Kapitel zu einem Automations-Server ausgebaut.

Hinweis: Aufgrund der starken Abhängigkeit der Automation vom Component Object Model (COM) werden in diesem Kapitel häufig Begriffe aus dem COM fallen, deren Verständnis für das Schreiben eigener Automations-Server nicht unbedingt notwendig, aber doch von Vorteil ist. Ausführliche Erläuterungen von Grundlagen zum COM finden Sie in den Kapiteln 2.7 und 8.5.

Automatisieren von TreeDesigner-Funktionen

Das in Abbildung 8.14 gezeigte Beispielprogramm *TDCtrl* ist ein Automatisierungs-Controller; es steuert den TreeDesigner aus Kapitel 5 und stellt dem Benutzer zwei Funktionen zur Verfügung, die der TreeDesigner alleine noch nicht bietet:

- Es gibt eine Liste aller Objekte und deren Koordinaten aus.
- Es erlaubt die Erzeugung eines neuen Grafikobjekts durch Angabe von dessen Koordinaten (Schalter NEUES OBJEKT). Auf diese Weise können Sie neue Objekte genau aneinander ausrichten, was mit der Maus nur schwer möglich ist, da der TreeDesigner kein Ausrichtungsgitter besitzt.

Gegenüber der Steuerung von Word in Kapitel 8.6.1 gibt es hier den folgenden qualitativen Unterschied: Nicht nur die gesamte Anwendung (der TreeDesigner), sondern auch einzelne Objekte innerhalb dieser Anwendung (die einzelnen Dokumente) werden als Objekte angesprochen.

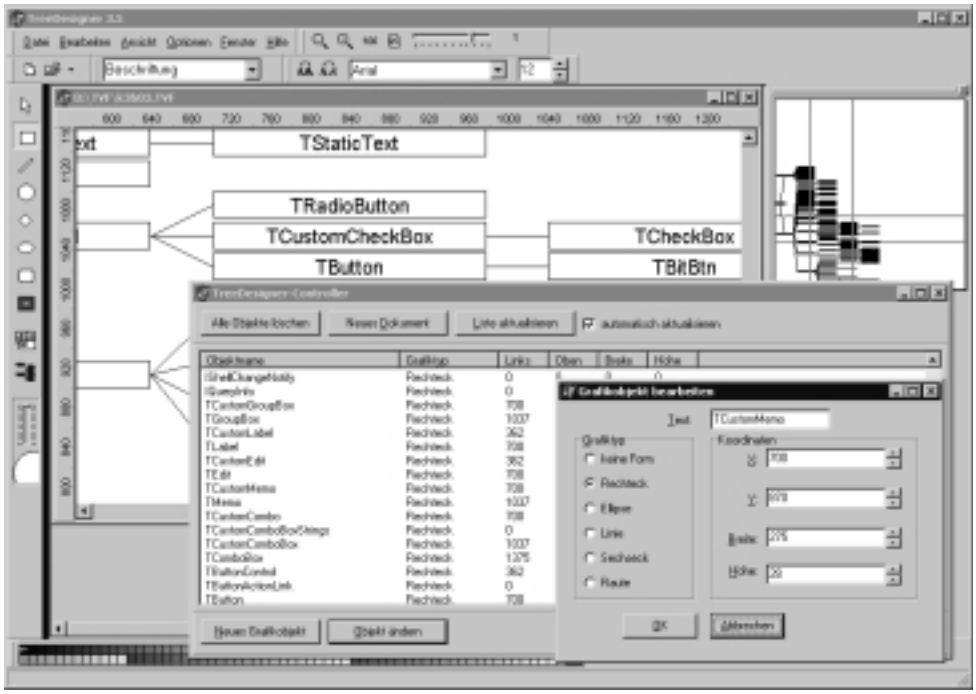


Abbildung 8.14: Dieses Beispielprogramm automatisiert den TreeDesigner und ergänzt diesen mit demonstrativen Präzisionsfunktionen.

Die Programmierung des TreeDesigners soll beispielsweise wie in der folgenden Methode aussehen, die mit dem Schalter NEUES OBJEKT verknüpft ist. Sie erfragt mit

der Automationsfunktion *TreeDesigner.GetCurrentDoc* ein Automations-Objekt für das aktuelle Dokument des TreeDesigners, ruft das in der Abbildung gezeigte Dialogformular von *TDCtrl* auf und erzeugt über die Dokument-Methode *CreateObject* ein Objekt mit den in der Dialogbox eingegebenen Parametern. Dabei ist *TreeDesigner* eine bei Programmstart erzeugte Variante, die der Variante *MsWord* des letzten Kapitels entspricht und für den gesamten TreeDesigner steht:

```
// in der OnCreate-Methode:
  TreeDesigner := CreateOleObject('TreeDsgn.TreeDesigner2');

procedure TForm1.NewObjectClick(Sender: TObject);
var
  CurrentDoc: Variant;
begin
  with NewObjectDlg do
    if ShowModal = mrOK then begin
      CurrentDoc := TreeDesigner.GetCurrentDoc;
      CurrentDoc.CreateObject(GrType.ItemIndex + 1,
           X.Position, Y.Position, W.Position, H.Position,
           Text.Text);
    end;
end;
```

Im Folgenden wird es darum gehen, dem TreeDesigner unter anderem die im Beispiel aufgerufenen Automatisierungs-Methoden *GetCurrentDoc* und *CreateObject* zu verleihen.

Hinweis: Falls Ihre Anwendung irgendwann einmal von Delphi 1 portiert worden ist, müssen Sie das Hauptprogramm noch um einen Aufruf von *Application.Initialize* erweitern, um die Anwendung für die COM-Automation tauglich zu machen. Alle 32-Bit-Delphi-Versionen erzeugen diesen Aufruf beim Anlegen einer neuen Projektdatei bereits automatisch.

Erzeugen eines Automationsservers *R170*

Ein *Automatisierungsobjekt* im Sinne von Delphis Experten (genauer: eine Automatisierungs*klasse*) ist eine von *TAutoObject* abgeleitete Klasse, die ein von *IDispatch* abgeleitetes COM-Interface unterstützt. Die Methoden eines *IDispatch*-Interfaces können in einem Automatisierungs-Client aufgerufen werden, ohne dass diese Aufrufe bereits fest im kompilierten Programmcode eingebunden sind. Im einfachsten Fall können Sie diese Klasse tatsächlich als *das* Automatisierungsobjekt betrachten, denn wenn der Client die Prozedur *CreateOleObject* aufruft, brauchen Sie sich im Server nicht um die Erzeugung eines Objekts dieser Klasse zu kümmern – dies erledigt die VCL des Servers automatisch.

Um eine Automatisierungsklasse zu erzeugen, rufen Sie mit DATEI | NEU | WEITERE Delphis Objektablage auf und wählen *Automatisierungsobjekt* von der Seite *ActiveX*. In der Dialogbox geben Sie die folgenden Daten ein:

- Einen Klassennamen (im Beispiel des TreeDesigners: *TreeDesigner2*, Delphi fügt das obligatorische *T* für den Klassennamen und das *I* für das COM-Interface selbst hinzu)
- Eine Instanziierungs-Art, die Sie bei *Mehrere Instanzen* belassen können, sofern Ihr Server mehrere Clients gleichzeitig bedienen kann; im Falle des TreeDesigners wurde hier *Eine Instanz* gewählt, damit ein TreeDesigner-Dokument nicht gleichzeitig von mehreren Clients aus editiert werden kann. (Die dritte mögliche Einstellung *Intern* werden wir noch für das Dokument-Automationsobjekt benötigen.)
- Unter *Threading-Model* wurde für den TreeDesigner *Einfach* gewählt, da nicht vorgesehen ist, dass der TreeDesigner aus mehreren Threads gleichzeitig angesprochen wird. Wie aber schon bei der Registrierung von einfachen COM-Objekten (siehe Kapitel 8.5.4) wirkt sich diese Angabe nur für die automatische Registrierung aus und impliziert keine weiteren Garantien, etwa, dass wirklich immer nur einem Thread erlaubt wird, auf den TreeDesigner zuzugreifen.

Wenn Sie die Eingaben im Dialog bestätigen, erzeugt Delphi zweierlei Dateien:

- Eine Typenbibliothek, in der die Schnittstelle der Automationsklasse in Form eines COM-Interfaces beschrieben wird. Die Bibliothek erhält den Namen des Projekts mit der Endung `.tlb`, im Falle des TreeDesigners als `treedsgn.tlb`. Diese Bibliothek ergänzen Sie später mit der Deklaration der Automationsmethoden und -Properties.
- Eine noch unbenannte Unit, die ein Gerüst für die Automationsklasse enthält. Der *initialization*-Teil der Unit enthält bereits den zur Initialisierung eines normalen Automationsobjektes notwendigen Code.

Hinweis: Den Code im Initialisierungsteil der Unit können Sie später dazu verwenden, die im Automatisierungsobjekt-Experten vorgenommenen Einstellungen für *Instanziierung* und *Threading-Modell* zu überprüfen. Diese schlagen sich im Aufruf von *TAutoObjectFactory.Create* beispielsweise in den Werten *ciInternal* für eine interne Instanziierung und *tmSingle* für das einfache Threading-Model nieder.

Typenbibliotheken und ihr Editor

Wir beginnen damit, die Typenbibliothek mit den Automationsmethoden zu ergänzen. Hierzu öffnet Delphi nach dem Anlegen eines Automationsobjekts automatisch den Typenbibliotheks-Editor (TLB-Editor), den Sie mit DATEI | ÖFFNEN jederzeit erneut öff-

nen können, wenn Sie in der Dateiauswahlbox die entsprechende `.tlb`-Datei angeben (diese steht erst nach dem ersten Speichern des Projekts zur Verfügung).

Bei jeder Speicherung der `.tlb`-Datei wird auch eine Object-Pascal-Unit erzeugt, die die in der Bibliothek definierten Schnittstellen in Pascal-Syntax beschreibt. Der Name dieser Unit wird von Delphi vorgegeben, er setzt sich aus dem Namen der Bibliothek und einem Zusatz zusammen, im Fall des TreeDesigners heißt sie `TreeDsgn_TLB.pas`. Delphi bindet diese Unit automatisch in das Projekt ein. Da sie automatisch neu erzeugt wird, sobald Sie im Typenbibliotheks-Editor eine kleine Änderung vornehmen und speichern, lohnt es sich nicht, sie von Hand zu ändern.

Der Typenbibliotheks-Editor

Abbildung 8.15 zeigt die im Editor geöffnete Typenbibliothek des TreeDesigners, noch ohne die in Kapitel 8.7.5 hinzukommenden Schnittstellen. Der Editor stellt die Struktur einer solchen Bibliothek in einem TreeView hierarchisch dar. Auf der ersten Ebene unter der Wurzelebene können sich Objekte der vier ersten in der Symbolleiste gezeigten Kategorien befinden: *Interface*, *DispInterface*, *CoClass* und *Enumeration*. Die Bibliothek des TreeDesigners enthält ein Interface zur Definition der Automationsmethoden (*ITreeDesigner2*) und die Angabe einer Co-Klasse, die diese Methoden implementiert (die Co-Klasse selbst gehört *nicht* zur Typenbibliothek). Die vierte Objektkategorie, *Enumeration*, ist dafür gedacht, die Wertebezeichnungen von Aufzählungstypen anderen Anwendungen zugänglich zu machen (siehe z.B. die Einträge *TxMouseButton* und *TxDragMode* in der Typenbibliothek aus Kapitel 6.8.2).

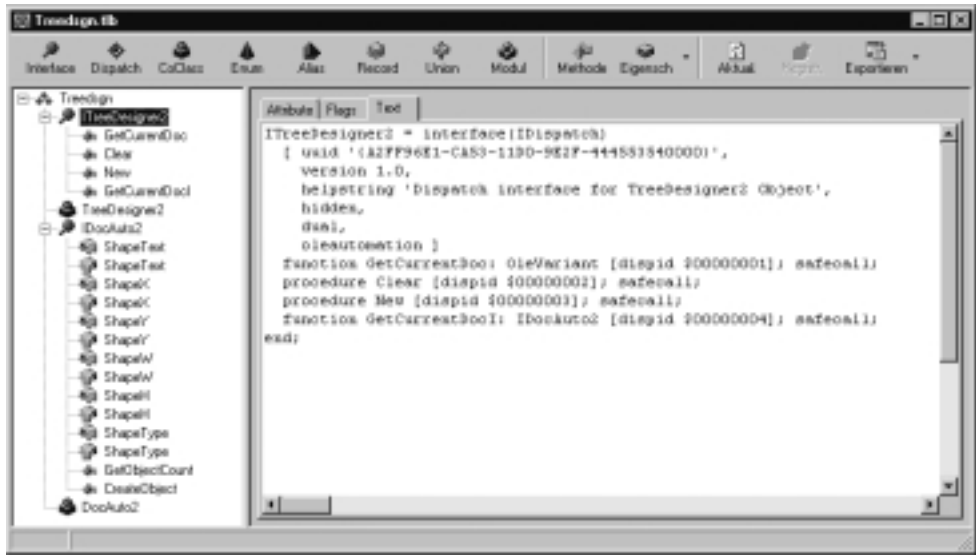


Abbildung 8.15: Die Automatisierungs-Schnittstellen des TreeDesigners im Typenbibliotheks-Editor

Die TLB des TreeDesigners

Die für die Automation notwendigen Knoten der Typenbibliothek (in Abbildung 8.15 sind das die Knoten *ITreeDesigner2* und *TreeDesigner2*) werden bereits durch den Automations-Experten erzeugt; Ihre Aufgabe ist es, die Methoden zu definieren, die vom Automations-Client aufgerufen werden sollen. Das Haupt-Automationsobjekt des TreeDesigners soll über die folgenden drei Methoden verfügen:

```
function GetCurrentDoc: OleVariant;
procedure Clear;
procedure New;
```

Statt nun die Baumdarstellung auf der linken Seite des TLB-Editors schrittweise um einzelne neue Methoden zu erweitern und die Methodendeklarationen auf der Seite *Attribute* einzutragen, können Sie einfach den *ITreeDesigner2*-Knoten wählen, die Seite *Text* aufschlagen und die Deklarationen dort vor dem *end* der dort angezeigten Interface-Deklaration eingeben. Dies funktioniert jedoch nur, wenn der TLB-Editor auf die richtige Sprache eingestellt ist. Eventuell müssen Sie den Editor zunächst schließen, in der IDE unter TOOLS | UMGEBUNGSOPTIONEN | TYPBIBLIOTHEK die Sprache PASCAL wählen und dann den TLB-Editor neu öffnen.

Nachdem Sie die Methoden in der üblichen Pascal-Syntax eingegeben haben, drücken Sie den Schalter mit dem Hinweistext *Implementierung aktualisieren*, damit Delphi ein Gerüst für die Unit der CoKlasse erzeugt (bzw. dieses aktualisiert, falls es bereits früher erzeugt wurde). Wenn Sie unter TOOLS | UMGEBUNGSOPTIONEN | TYPBIBLIOTHEK die Option ÄNDERUNGEN VOR AKTUALISIERUNG ANZEIGEN eingeschaltet haben, erhalten Sie vorher noch ein Dialogfenster, in dem Sie die einzelnen Änderungen bestätigen können.

Implementieren der CoKlasse

Wenn Sie die Schnittstelle Ihres Automationsobjekts wie beschrieben im TLB-Editor festgelegt haben, können Sie wieder im bekannten Code-Editor weiterarbeiten, denn dort müssen die Methoden noch implementiert werden. Dazu müssen Sie lediglich das schon erwähnte Unit-Gerüst »ausfüllen« .

Im Falle des TreeDesigners wurde die Implementationsunit unter dem Namen *TDAuto* gespeichert. Der *uses*-Zeile im Implementationsteil der Unit wurden die Unit *MdiForm* und *DocForm* hinzugefügt, da sowohl das Hauptformular als auch die Dokumentformulare im Automationscode angesprochen werden. Die Methoden *Clear* und *New*, deren Rümpfe automatisch durch den Schalter *Implementation aktualisieren* erzeugt wurden, sehen wie folgt aus:

```
procedure TTreeDesigner2.New;
begin
  MainForm.MdiNewClick(nil);
```

```
end;

procedure TTreeDesigner2.Clear;
begin
  with MainForm.ActiveMDIChild as TDocumentForm do
    CmDeleteAllClick(nil);
end;
```

Da beide Methoden nur mit dem Hauptfenster (*MdiForm*) arbeiten und es davon nur eines gibt, sind sie völlig unproblematisch. Im Beispielcode zur Verwendung der TreeDesigner-Automation wurde jedoch auch schon eine Methode *CurrentDoc.CreateObject* aufgerufen, und diese bezieht sich auf ein einzelnes Dokumentfenster.

Der TreeDesigner definiert für diese Dokumentfenster eine zweite Automationsklasse und stellt für jedes dieser Fenster ein eigenes Automationsobjekt zur Verfügung. Ein solches Objekt soll dem Automatisierungs-Client ermöglichen, ein einzelnes TreeDesigner-Dokument wie ein Objekt zu manipulieren. Ein solches Dokument-Objekt kann nicht mit *CreateOleObject* erzeugt werden (der Grund dafür ist, vereinfacht gesagt, dass *CreateOleObject* ja nicht wissen kann, für welches Dokument ein Objekt erzeugt werden soll); hier kommt die bereits im Beispiel aufgerufene Methode *GetCurrentDoc* ins Spiel.

Um die Automatisierungs-Schnittstelle möglichst einfach zu halten, erlaubt es der TreeDesigner seinem Client nur, jeweils das Automatisierungsobjekt des aktiven Fensters abzufragen, indem er die Methode *GetCurrentDoc* des »Haupt-Automatisierungsobjekts« aufruft. *GetCurrentDoc* ist wie folgt implementiert:

```
function TTreeDesigner2.GetCurrentDoc: OleVariant;
begin
  with MainForm.ActiveMDIChild as TDocumentForm do
    Result := GetAutoObjectVariant;
end;
```

GetCurrentDoc gibt also wirklich nur das Automatisierungsobjekt von *ActiveMDIChild* zurück. Wie *AutoObjectVariant* erzeugt und von *TDocumentForm* verwaltet wird und was es eigentlich ist, bleibt noch zu besprechen, wird aber bis zum Abschnitt *Verwaltung der Dokument-Automationsobjekte* im nächsten Kapitel hinausgeschoben.

Hinweis: In gewisser Weise sind auch Delphi-Editor und TLB-Editor Two-Way-Tools, denn auch wenn Sie eine Automationsklasse wie im Beispiel die Klasse *TTreeDesigner2* implementiert haben, können Sie das Automations-Interface noch im TLB-Editor ausbauen. Delphi erweitert dann – wenn Sie wieder den Schalter *Implementierung aktualisieren* drücken – die Implementations-Unit (im Beispiel *TDAuto*) um neue, leere Methodenrümpfe und fügt die Deklaration der Methode zur CoKlasse hinzu.

Dies entspricht damit weitgehend dem Hinzufügen einer neuen Ereignisbearbeitungsmethode zu einem Formular. Wenn Sie eine Methode aus dem Interface löschen, wird die zugehörige Implementationsmethode natürlich nicht gelöscht. Eine weitere Ähnlichkeit mit den Two-Way-Tools für Formulare finden Sie im Projektquelltext: In *uses*-Klauseln vermerkt Delphi hinter Units, die eine Automationsklasse implementieren, den Namen dieser Klasse, ähnlich wie auch der Name der Formulare hinter den Formular-Units notiert wird.

8.7.2 Interne Automations-Objekte

Das folgende Listing zeigt die Schnittstelle des Automationsobjekts für ein TreeDesigner-Dokument:

```
property ShapeText[i: Integer]: WideString;
property ShapeX[i: Integer]: Integer;
property ShapeY[i: Integer]: Integer;
property ShapeW[i: Integer]: Integer;
property ShapeH[i: Integer]: Integer;
property ShapeType[i: Integer]: Integer;
function GetObjectCount: Integer;
procedure CreateObject(SType, x1, y1, x2, y2: Integer;
                       const text: WideString);
```

Neben der schon erwähnten Methode *CreateObject* gehören zu dieser Schnittstelle vor allem Properties, die verschiedene Daten der Grafikobjekte zugänglich machen. Der Typ des Beschriftungstextes ist als *WideString* angegeben, denn nur dieser String-Typ ist für die voreingestellte Art der COM-Automation zugelassen.

Erzeugen der zweiten Automationsklasse

Um im TreeDesigner-Projekt eine zweite Automationsklasse mit dieser Schnittstelle zu erzeugen, ist zunächst ein weiterer Durchgang mit dem Automatisierungsobjekt-Experten erforderlich: Der Klassenname ist diesmal *DocAuto2*, die Art der Instanziierung heißt *Intern*, als Threading-Model wird wieder *Einfach* gewählt.

Es ist diesmal allerdings nicht möglich, die oben gezeigten Deklarationen direkt in der *Text*-Seite des TLB-Editors einzugeben, denn der Editor verlangt (in Anlehnung an die COM-Spezifikation) für Properties die separate Angabe von Lese- und Schreibmethoden. Für die Erzeugung neuer Properties ist es wahrscheinlich am einfachsten, wenn Sie die Deklaration daher nicht komplett auf der Seite *Text* angeben, sondern im TreeView auf der linken Seite des Editors das Interface markieren (im Beispiel *IDocAuto2*), aus dem lokalen Menü NEU | EIGENSCHAFT wählen und den Property-Namen an Ort und Stelle angeben. Der TLB-Editor erzeugt dann automatisch eine neue Lese- und

Schreibmethode, deren weitere Parameter Sie auf den Seiten im rechten Teil des Editors modifizieren können (den Property-Typ auf der Seite *Attribute* und die eventuellen Array-Indizes auf der Seite *Parameter*).

Die endgültigen Deklarationen für die Properties *ShapeText* und *ShapeX* für das TreeDesigner-Interface *IDocAuto2* sehen beispielsweise wie folgt aus:

```
function Get_ShapeText(i: Integer): WideString;
procedure Set_ShapeText(i: Integer; const Value: WideString);
function Get_ShapeX(i: Integer): Integer;
procedure Set_ShapeX(i, Value: Integer);
```

Entsprechend werden auch die anderen Properties deklariert. Zum Schluss bewirkt der bekannte Schalter *Implementierung aktualisieren*, dass Delphi Methodenrümpfe zur bisher vollautomatisch erzeugten Co-Klasse hinzufügt.

Hinweis: Der Grund für die etwas umständliche Angabe von Lese- und Schreibmethode liegt darin, dass COM-Interfaces nur Methoden, aber keine Properties unterstützen. Im Automations-Client können trotzdem die oben gezeigten Properties verwendet werden, denn sie sind Teil der vollständigen, von Delphi erzeugten *IDocAuto2*-Deklaration (siehe Unit *TreeDsgn_TLB*).

Formular und Automationsklasse in einer Unit

Damit wäre die Arbeit im Typenbibliotheks-Editor schon wieder erledigt und wir können uns der Implementation dieser zweiten Co-Klasse zuwenden. Für diese Automationsklasse eine eigene Unit zu verwenden, erweist sich jedoch aus mehreren Gründen als ungünstig:

- Dokument-Automationsklasse und Formular müssen sehr eng zusammenarbeiten. Damit die Automationsklasse auch auf private Variablen des Formulars zugreifen kann, muss sie sich in derselben Unit wie das Formular befinden.
- Da sich Automations- und Formularklasse gegenseitig verwenden, müsste eine der beiden Units im Implementationsteil der anderen Unit eingebunden werden. Dies verträgt sich schlecht mit der Tatsache, dass Automationsobjekt und Formular sich gegenseitig in Variablen speichern, die bereits in der Klassendeklaration im Interface der Unit deklariert werden müssen (hier gibt es nur umständliche Lösungen wie etwa, das Formular in der Automationsklasse als *TForm* zu deklarieren und diese Variable dann in die Implementation in jeder Methode in *TDocumentForm* umzuwandeln).

Aus diesen Gründen wurde der gesamte automatisch erzeugte Code inklusive des *initialization*-Teils aus der automatisch erzeugten Unit in das Dokumentformular übernommen und die unbenannte Unit aus dem Projekt gelöscht.

Bei diesem manuellen Eingriff wird jedoch die Verbindung zwischen TLB-Editor und automatisch erzeugter Unit zerstört, die bisher bewirkt hat, dass Änderungen an der Typenbibliothek auch auf die Implementierungs-Unit angewendet werden. Diese »unsichtbare Verbindung« lässt sich aber wieder herstellen, da sie gar nicht so unsichtbar ist. Anfangs wurde sie durch folgende Zeile in der *uses*-Anweisung der Projektdatei hergestellt:

```
uses
  ...,
  UNIT2 in 'UNIT2.PAS' {DocAuto2: CoClass},
```

Dieser Kommentar am Ende der Zeile muss lediglich zu der Unit übertragen werden, in die die Implementierung des Automationsobjekts verschoben wurde, also hier in die Unit *DocForm*, die in der Projektdatei noch wie folgt eingebunden wird:

```
DOCFORM in 'DOCFORM.PAS' {DocumentForm}
```

Nach der folgenden kleinen Änderung erkennt Delphi diese Unit als CoKlassen-Implementierungs-Unit und bietet bei Drücken des *Aktualisieren*-Schalters im TLB-Editor die Aktualisierung der Unit an:

```
DOCFORM in 'DOCFORM.PAS' {DocumentForm, DocAuto2: CoClass},
```

Hinweis: Eine unerfreuliche Nebenwirkung dieser Maßnahme ist, dass das Formular *DocumentForm* nicht mehr in der Formularliste der IDE aufgeführt wird. Dies hatte jedoch bisher keine weiteren erkennbaren negativen Folgen.

Implementierung des internen Automationsobjekts *R171*

Bevor wir zum interessantesten Aspekt unseres zweiten Automationsobjektes kommen, nämlich der Erzeugung und Verwaltung, sollen die folgenden Codeauszüge belegen, dass die Implementation der Klasse völlig unproblematisch ist. Als Erstes benötigen wir eine neue Variable für die Klasse; in dieser wird das Dokumentformular gespeichert, mit dem das Automationsobjekt zusammenarbeitet:

```
TDocAuto2 = class(TAutoObject, IDocAuto2)
protected
  DocForm: TDocumentForm;
```

Die Initialisierung dieser Variablen gehört wieder zu den »Verwaltungsaufgaben«, die wir ja hier noch nicht untersuchen wollen.

Hinweis: Sie können einer Automationsklasse beliebige neue Variablen und Methoden hinzufügen. Nur Methoden, die vom Client aufgerufen werden sollen, müssen Sie über den Typenbibliotheks-Editor erzeugen. Dies ist eine weitere Ähnlichkeit zu den Formularklassen, für die Sie Ereignisbearbeitungsmethoden über den Objektinspektor erzeugen können, die Sie ansonsten aber fast völlig frei erweitern dürfen.

Als Beispiel dafür, wie ein Dokumentformular durch das Automationsobjekt gesteuert wird, soll die schon erwähnte Methode *CreateObject* dienen; sie greift auf die Liste der Grafikelemente des verbundenen Dokumentfensters zu, um ein neues *TGraphicElement*-Objekt zu erzeugen:

```
procedure TDocAuto.CreateObject(SType: Integer;
  x1, y1, x2, y2: Integer; text: String);
var
  NewNode: TGraphicElement;
begin
  NewNode := DocForm.Document.
    CreateObjectByDupName(Doc.PaintBox, text, TShapeType(SType));
  NewNode.SetNewRect(Rect(x1, y1, x2, y2));
end;
```

Die Property-Zugriffsmethoden sind sehr einfach zu implementieren, z.B.:

```
procedure TDocAuto2.Set_ShapeText(i: Integer; const Value: WideString);
begin
  DocForm.Document[i].Text := Value;
end;
```

Dabei wird der *WideString*-Parameter automatisch in einen normalen String für das *Text*-Property des Grafikelements umgewandelt (zu den Properties der Grafikelemente und zum Aufbau des TreeDesigner-Dokuments siehe Kapitel 5.3.3).

Verwaltung der Dokument-Automationsobjekte

Der Lebenszyklus eines Dokument-Automationsobjekts im TreeDesigner beginnt damit, dass der Automations-Controller die Methode *GetCurrentDoc* des TreeDesigner-Automationsobjekts aufruft. Diese Methode wurde bereits gezeigt, sie holt sich das Automations-Objekt einfach aus dem Property *AutoObjectVariant* des Dokumentformulars:

```
public
  function GetAutoObjectVariant: Variant;
```

Die Property-Lesemethode *GetAutoObjectVariant* ist sehr einfach aufgebaut:

```
function TDocumentForm.GetAutoObjectVariant: Variant;
var
  AutoObject: TDocAuto2;
```

```
begin
  AutoObject := TDocAuto2.Create;
  AutoObject.Doc := self;
  Result := AutoObject as IDispatch;
end;
```

Sie erzeugt eine Instanz der Automations-Co-Klasse *TDocAuto2*, mit anderen Worten: das gewünschte Automationsobjekt. Dieses Objekt speichert sie nur deshalb in der Variablen *AutoObject* zwischen, da sie noch seine *Doc*-Variable setzen muss. Dann gibt sie das *IDispatch*-COM-Interface des Automationsobjekts zurück. Über dieses Interface können alle allgemeinen Automations-Clients sich zur Laufzeit an die Methoden des Automationsobjekts ankoppeln (d.h., dass die Methoden dieses Objekts bei der Übersetzung des Automations-Clients noch nicht bekannt sein müssen; wie auch im Beispiel der Steuerung von MS-Word in Kapitel 8.6.1).

Außer der Methode *GetAutoObjectVariant* ist kein weiterer Code erforderlich, denn in Object Pascal geschehen alle weiteren Verwaltungsschritte des Automationsobjekts automatisch: Bei der Zuweisung *Result := AutoObject as IDispatch* wird der Referenzzähler des Automationsobjekts erhöht, so dass das Objekt nicht automatisch wieder freigegeben wird, wenn die Variable *AutoObject* am *end* der Funktion ungültig wird. Die Freigabe des Automationsobjekts wird alleine vom Automations-Controller bestimmt. Als Beispiel soll noch einmal die am Anfang des Kapitels abgedruckte Methode von *TdCtrl* dienen, denn so wie dort werden die Automationsobjekte auch in allen anderen Methoden von *TdCtrl* verwendet:

```
procedure TTDControllerForm.NewObjectButtonClick(Sender: TObject);
var
  CurrentDoc: OleVariant;
begin
  CurrentDoc := TreeDesigner.GetCurrentDoc; // Referenzzähler = 1
  ... Verwendung von CurrentDoc...
end; // CurrentDoc wird ungültig, Referenzzähler = 0
```

Wenn *CurrentDoc* beim *end* dieser Prozedur ungültig wird, ruft der vom Compiler erzeugte Code automatisch die *_Release*-Methode auf, die den Referenzzähler des Objekts um eins senkt. Nach dem Verlassen der Methoden *GetAutoObjectVariant* und *GetCurrentDoc* hatte dieser Zähler noch einen Stand von 1, so dass er nach *NewObjectButtonClick* bei 0 angekommen ist. Das bedeutet, dass sich das Automationsobjekt automatisch freigibt.

Hinweis: Solange das Automationsobjekt noch nicht auf diese Weise freigegeben wurde, können Sie die Automations-Server nicht schließen, ohne dass die Automationsunterstützung der VCL automatisch eine Warnung ausgibt.

Sie können Ihre Automationsobjekte natürlich auch so verwalten, dass der Automations-Controller die Objekte für längere Zeit behalten darf. Ein Beispiel dafür liefert

Delphis Beispielprogramm `demos\ActiveX\oleauto\autoserv\memoedit.dpr`: Ein Automations-Server, der wie der TreeDesigner eine MDI-Anwendung ist, bei der jedes Dokument ein eigenes Automationsobjekt erzeugen darf. Diese Automationsobjekte werden allerdings in Formularvariablen gespeichert, so dass jedes Dokument nur einmal ein Automationsobjekt erzeugen muss. Damit der Controller dieses solange behalten darf, wie er will, muss das Dokumentformular beim seinem *OnDestroy*-Ereignis die Verknüpfung zwischen Automationsobjekt und Formular löschen (auf *nil* setzen), so dass das Automationsobjekt nicht mehr in Versuchung kommt, Methoden eines nicht mehr existenten Formulars aufzurufen: Zu Beginn prüft jede Methode des Automationsobjekts, ob die Formularvariable noch nicht auf *nil* gesetzt wurde (siehe die Methoden von *TMemoDoc* in `editfrm.pas` im eben genannten Verzeichnis).

Im Beispielprogramm *TdCtrl* kann es noch zu einem Fehler kommen, wenn der Benutzer beispielsweise den Schalter *Objekt ändern* drückt und, bevor er den dann erscheinenden Dialog mit *Ok* beendet, das aktuelle Dokumentformular im TreeDesigner schließt.

Registrierungs-Formalitäten

Eine Voraussetzung für die Funktion der COM-Automatisierung ist, dass der Automatisierungs-Server in der Windows-Registrierung eingetragen ist. Die VCL registriert die nicht-internen Automatisierungsobjekte eines Servers bereits automatisch beim dessen Start. Es kann jedoch sein, dass der Server niemals als eigenständige Anwendung gestartet werden soll – spätestens dann ist es erforderlich, die Registrierung auf eine andere Weise vorzunehmen. Automations-Server, die als EXE-Datei vorliegen, unterstützen zu diesem Zweck den Startparameter */REGSERVER*. Die VCL erkennt diesen Parameter bereits automatisch und bricht, wenn er angegeben ist, die Ausführung des Programms nach dem Eintragen der Registrierungsinformation in die Registry ab (in der Methode *Application.Initialize*, es wird also kein Formular angezeigt). Es bietet sich daher an, einen zu registrierenden Server mit dem Parameter */REGSERVER* aus einem Installationsprogramm heraus aufzurufen.

Hinweis: Ein Automatisierungs-Server sollte noch zwei weitere Parameter erkennen: »/UNREGSERVER« (zum Löschen der Registrierungsinformation) und »/EMBEDDING« (wenn der Server nicht vom Benutzer, sondern von einem Automatisierungs-Controller aufgerufen wurde; per Voreinstellung sollte der Server dann nicht am Bildschirm sichtbar werden, sondern seine Arbeit im Hintergrund verrichten). Beide Parameter werden von der VCL in das Property *StartMode* des globalen Objekts *Automation* umgewandelt (siehe Online-Referenz zu *TComServer.StartMode*).

Automations-Server, die als DLL vorliegen, stellen für die (De-)Registrierung die Funktionen *DllRegisterServer* und *DllUnregisterServer* zur Verfügung. Zum Aufruf dieser Funktionen aus Installationsprogrammen und zu den Umständen, unter denen die VCL auch diese Funktionen automatisch bereitstellt, siehe Kapitel 8.5.4.

Nicht registriert werden interne Automationsklassen wie *DocAuto2* im Beispiel des TreeDesigners. Es wäre auch gar nicht sinnvoll, ein *DocAuto2*-Objekt per *CreateOleObject('TreeDsgn.DocAuto2')* zu erzeugen (wie das *TreeDesigner2*-Objekt), da es nur durch *TTreeDesigner.GetCurrentDoc* richtig mit einem Dokumentformular verknüpft werden kann.

8.7.3 Alternative Dispatch-Möglichkeiten

Wie in Kapitel 8.6.2 schon erläutert wurde, gibt es noch einen effizienteren Weg, die Methoden eines Servers aufzurufen, und zwar unter Verwendung einer Typenbibliothek. Dieser Abschnitt soll nun diese Technik auch auf den TreeDesigner übertragen, für den wir ja in den vorangegangenen Abschnitten bereits eine Typenbibliothek angelegt haben. Diese gilt es jetzt auch im Automations-Controller zu nutzen. Hierzu finden Sie auf der CD unter dem Namen *TdCtrl2* eine zweite Version des TreeDesigner-Clients, die Typenbibliotheken verwendet. In diesem speziellen Fall wird dadurch die Geschwindigkeit der Automation verdoppelt.

Automations-Clients unter Nutzung einer Typenbibliothek *R133*

Das folgende Listing zeigt die ersten beiden Stellen, an denen sich *TdCtrl2* vom bisher besprochenen Beispielprogramm unterscheidet:

```
uses // Unterschied 1: Einbinden der Type Library:
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComObj, ComCtrls, ExtCtrls, TreeDsgn_TLB;

type
  TTDControllerForm = class(TForm)
    ...
    TreeDesigner: ITreeDesigner2; // Unterschied 2: Verwendung der TLB

procedure TTDControllerForm.FormCreate(Sender: TObject);
begin
  InitTreeDesigner; // virtuelle Methode,
                    // wird in Kapitel 8.7.4 überschrieben
  ...
procedure TTDControllerForm.InitTreeDesigner;
begin
  try  // Unterschied 3: Objekterzeugung
    TreeDesigner := CoTreeDesigner2.Create;
```

```
  except
    MessageDlg('Konnte Automations-Server nicht finden.', ...
  ...
```

TreeDesigner wird hier nicht als *OleVariant*, sondern als *ITreeDesigner2* deklariert, also als Schnittstelle eines COM-Objekts, genauer gesagt: des TreeDesigner-Automationsobjekts. In Kapitel 8.7.1 haben wir die Deklaration dieser Schnittstelle im Typenbibliotheks-Editor erzeugt, der Pascal-Quelltext dazu befindet sich in der Unit *TreeDsgn_TLB*, die oben in das Projekt *TdCtrl2* eingebunden wird.

CoTreeDesigner2 ist eine von Delphi automatisch erzeugte Klasse, die eine schnelle Möglichkeit bietet, ein TreeDesigner-Automationsobjekt direkt und ohne den Aufruf von *CreateOleObject* zu erzeugen. Delphi erzeugt eine solche *Co...*-Klasse für jedes mit seinem Experten erstellte Automatisierungsobjekt, die Details dazu finden Sie für Ihr Projekt jeweils in der Unit *Projekt_TLB.pas* bzw. für den TreeDesigner in *TreeDsgn_TLB.pas*.

Hinweis: Der Aufruf von *CoTreeDesigner2.Create* erspart zwar nicht viel Code, denn wie Sie aus dem von Delphi erzeugten Quelltext dieser *Create*-Methode ersehen können, besteht sie nur aus einer einzigen Zeile. Der Vorteil darin, diese Zeile in *CoTreeDesigner2* »zu verstecken«, liegt darin, dass wir *CoTreeDesigner2.Create* so aufrufen können, als handle es sich dabei um die Co-Klasse des TreeDesigner-Automationsinterfaces. Die »echte« Co-Klasse ist ja in der Unit *TDAuto* deklariert und heißt *TTreeDesigner2*, da diese Klasse jedoch in den Bereich des Automations-Servers gehört und vom Automations-Client versteckt bleibt, ist es sinnvoll, dass Delphi mit *CoTreeDesigner2* eine Klasse erzeugt, die auch vom Client leicht aufgerufen werden kann. Die Situation ist hier ähnlich wie in Kapitel 6.8.5, in dem Delphi die Klasse *TColorPaletteX* einmal für die ActiveX-DLL und einmal für den ActiveX-Container deklariert hat.

Umstellung des internen Automationsobjekts *R172*

Erheblich wichtiger als die Verwendung von *ITreeDesigner2* ist es, für die Geschwindigkeitssteigerung des Beispielprogramms auch die interne Automationsklasse *DocAuto2* auf die Verwendung der Typenbibliothek umzustellen, denn ihre Methoden werden in *TdCtrl* sogar in Schleifen wiederholt aufgerufen.

So wie oben der Aufruf von *CreateOleObject* ersetzt wurde, muss jetzt also noch der Aufruf von *TreeDesigner.GetCurrentDoc* durch den Aufruf einer Methode ersetzt werden, die statt der *OleVariant* eine *IDocAuto2*-Schnittstelle zurückliefert. Hierzu wird die Klasse des TreeDesigner-Dokumentformulars wie folgt erweitert:

```
public
  // verwendet in TTreeDesigner2.GetCurrentDoc
```

```
  function GetAutoObjectVariant: Variant;
  // alternative Funktion für effektiveren Automations-Aufruf:
  function GetAutoObject: IDocAuto2;
```

Das folgende Listing zeigt, worin sich die Methode *GetAutoObject* von der bisher verwendeten Methode *GetAutoObjectVariant* unterscheidet:

```
function TDocumentForm.GetAutoObject: IDocAuto2; // statt Variant
var
  AutoObject: TDocAuto2;
begin
  AutoObject := TDocAuto2.Create;
  AutoObject.Doc := self;
  Result := AutoObject as IDocAuto2; // statt IDispatch
end;
```

Mit dieser Methode stellt das Dokumentformular seine Automationsobjekte nicht mehr als »undurchsichtige« *OleVariant*-Strukturen dar, sondern direkt als *IDocAuto2*-Schnittstelle, die in unserer neuen Version des Automations-Clients direkt angesprochen werden soll, wie oben schon bei der *ITreeDesigner2*-Schnittstelle geschehen.

Was noch fehlt, ist lediglich eine kleine Erweiterung der *ITreeDesigner2*-Schnittstelle, um die *IAutoDoc2*-Schnittstelle an den Client weiterzugeben. Zur bereits gezeigten Methode gesellt sich nun die Methode *GetCurrentDocI*:

```
function TTreeDesigner2.GetCurrentDocI: IDocAuto2;
begin
  Result := (MainForm.ActiveMDIChild as TDocumentForm).GetAutoObject;
end;
```

Mit dieser kann die aus dem *TdCtrl*-Projekt bekannte Methode *UpdateList* wie folgt geändert werden:

```
procedure TTDControllerForm.UpdateList;
var
  CurrentDoc: IDocAuto2; // Unterschied 4: Noch einmal TLB
  ...
begin  // Unterschied 4: neue Automations-Methode:
  CurrentDoc := TreeDesigner.GetCurrentDocI;
  ...
  // bei der Arbeit mit CurrentDoc kein Unterschied im Quelltext:
  for i := 0 to CurrentDoc.GetObjectCount-1 do begin
```

Duale Schnittstellen

Das Versprechen, *TdCtrl2* sei doppelt so schnell wie *TdCtrl*, verlangt natürlich noch nach einer Erklärung. Der Geschwindigkeitsunterschied beruht auf den verschiedenen Arten der Bindung von Methodenaufrufen: Bei *TdCtrl* handelt es sich um späte Bindung zur Laufzeit, während die Methodenaufrufe von *TdCtrl2* schon vom Compiler erzeugt werden:

- Im ersten Fall wurden die Methoden des Automations-Servers über Varianten aufgerufen. Für den Aufruf einer Methode über eine Variante kann der Compiler keinen direkten Funktionsaufruf erzeugen, da er noch gar nicht weiß, ob eine Methode des angegebenen Namens überhaupt existiert, geschweige denn, wo sie sich befindet. Die Verbindung vom Namen der Methode zum Methodencode, die normalerweise der Compiler herstellt, wird beim Varianten-Aufruf erst zur Laufzeit gesucht. Das heißt, dass das Programm zur Laufzeit Stringvergleiche durchführen muss, bevor es zum Code der gewünschten Methode verzweigen kann.

  Grundlage für die beschriebene Form des Aufrufs von Automationsmethoden ist die *IDispatch*-Schnittstelle, die die Methoden *GetIdsOfNames*, *GetTypeInfo*, *GetTypeInfoCount* und *Invoke* bereitstellt, über die die gesamte Kommunikation zwischen Client und Server und die späte Bindung der Methodenaufrufe zum Methodencode abgewickelt werden kann.

- In *TdCtrl2* werden die Automationsmethoden über die *ITreeDesigner2*- bzw. *IAutoDoc2*-Schnittstelle aufgerufen. Wie Sie in *TreeDsgn_TLB.pas* überprüfen können, sind diese Schnittstellen von *IDispatch* abgeleitet, beinhalten also die eben beschriebene Funktionalität zur späten Bindung. Schon die erste Version des Clients *TdCtrl* hat also mit den Schnittstellen *ITreeDesigner2* und *IDocAuto2* gearbeitet, ohne es zu wissen. Erst durch die Einbindung der Typenbibliothek konnte der Compiler die Namen und Adressen der Methoden erfahren und so eine frühe Bindung durchführen, so dass der Aufruf der Methode erheblich schneller abläuft, als dies der Fall wäre, wenn er einen Stringvergleich voraussetzen würde.

Da Schnittstellen wie *ITreeDesigner2* und *IDocAuto2* auf zwei verschiedene Weisen aufgerufen werden können, heißen sie auch *duale Schnittstellen*.

Hinweis: Die »frühe Bindung« in der COM-Automation entspricht der späten Bindung beim Aufruf virtueller Methoden, denn der Aufruf im Automations-Controller findet noch immer über die Zwischenstation einer virtuellen Methodentabelle (VMT) statt.

8.7.4 Distributed COM (DCOM)

Einer der wichtigsten Trends im Bereich der Informationstechnologie ist wohl das verteilte Computing – die transparente Zusammenarbeit von Softwarekomponenten auf verschiedenen Rechnern im Netz. Transparent soll diese Zusammenarbeit insbesondere auch für den Softwareentwickler sein, das heißt, dass dieser nicht zwischen der Verwendung einer Komponente auf demselben Rechner und der Verwendung einer entfernten Komponente unterscheiden muss. Diese Transparenz herzustellen, ist die Aufgabe spezieller Softwareschichten, die zwischen dem Aufrufer eines Objekts (dem Client) und der Anwendung, die das Objekt implementiert (dem Server) vermittelt.

Hierfür gibt es konkurrierende Technologien, von denen Delphi unter anderem das von Microsoft entwickelte DCOM und das von der Object Management Group erfundene CORBA (Common Object Request Broker Architecture, siehe *www.omg.org*) unterstützt, Letzteres allerdings erst ab der Enterprise-Ausgabe.

Da DCOM in allen aktuellen Windows-Betriebssystemen enthalten ist, hat es bereits eine weite Verbreitung gefunden. Dies, die gute Unterstützung auch in den kleinen Ausgaben von Delphi und die Tatsache, dass DCOM auf COM aufbaut, welches auch als Grundlage für zahlreiche andere Windows-Technologien dient, sind Gründe dafür, dass sich dieses Kapitel nur mit DCOM, nicht aber mit der sicherlich nicht »weniger guten« CORBA befasst.

Windows 98/Me vs. Windows NT/2000/XP

Das verwendete Betriebssystem spielt in der Praxis eine wichtige Rolle. Auch wenn DCOM in Windows 98 implementiert und als kostenlose Erweiterung sogar für Windows 95 erhältlich ist, steht der volle Funktionsumfang nur in der Windows NT-Linie ab der Version 4.0 zur Verfügung. Während Clients unter Windows 9x noch wenig Einschränkungen unterworfen sind, müssen Sie bei einem DCOM-Server, der unter einem dieser Betriebssysteme läuft, verschiedene Dinge manuell durchführen, die unter Windows NT automatisch ablaufen: Sowohl das Programm `rpcss.exe` im System-Verzeichnis als auch die Anwendung, die den COM-Server enthält, müssen gestartet werden, *bevor* der COM-Client Zugriff auf diese Objekte verlangt. Hinzu kommen laut Microsoft-Dokumentation (z.B. Knowledge Base-Artikel *Q165101*) Einschränkungen bezüglich Sicherheit und Performance. Für den Test der Beispielprogramme wurden jedenfalls zwei Installationen von Windows NT 4.0 Workstation verwendet.

COM und DCOM

Dass DCOM auf COM aufbaut, bedeutet, dass Sie Ihr gesamtes COM-Wissen auf DCOM übertragen können. Alle zurückliegenden und noch folgenden Kapitel über die COM-Programmierung können auch auf DCOM angewendet werden, ohne dass dies dort explizit erwähnt wird.

Dies liegt daran, dass es ja schon die Aufgabe des COM war, getrennten Anwendungen eine einfache Zusammenarbeit über COM-Interfaces zu ermöglichen. Die Anwendungen müssen dabei im Wesentlichen nur zwei wichtige Voraussetzungen erfüllen: Sie müssen ihre COM-Klassen und Interfaces in der Windows-Registry eintragen, und sie müssen sich an die von COM unterstützten Datentypen halten (die Bereitstellung von Class Factories wird hier, weil von Delphi automatisch erledigt, nicht weiter beachtet). COM stellt dann im Gegenzug API-Funktionen bereit, mit denen Objekte von beliebigen in der Registry verzeichneten Klassen hergestellt werden können. Statt

Referenzen auf die eigentlichen Objekte werden nur Interfaces zurückgeliefert, deren Methoden Sie aber genauso aufrufen können wie Methoden Ihrer eigenen Anwendung. Tatsächlich führt der Aufruf einer solchen Methode dazu, dass die COM-Systembibliotheken von Windows den Aufruf erst an die andere Anwendung vermitteln und dabei auch die Daten in den Adressraum dieser anderen Anwendung übertragen.

Damit eine Anwendung nun über DCOM ein COM-Objekt auf einem anderen Rechner verwenden kann, übergibt sie der Objekt-Initialisierungsfunktion lediglich einen zusätzlichen Parameter, der diesen Rechner angibt (bei entsprechender Konfiguration entfällt jedoch auch dieser Unterschied). Die COM-Systembibliotheken kontaktieren dann über eines von mehreren Protokollen die COM-Systembibliotheken des anderen Rechners, und diese sehen in der Registry »ihres« Rechners nach der angeforderten COM-Klasse, worauf sie das gewünschte Objekt erzeugen. Alles, was danach anders abläuft als bei »Rechner-internem« COM, betrifft nur die Vermittlung von der Client- zur Server-Anwendung, die aber bei COM sowieso transparent für die Anwendungen abläuft.

Neben der erwähnten Besonderheit bei der Initialisierung eines entfernten Objekts gibt es noch einige andere beachtenswerte Aspekte, auf die die folgenden Abschnitte kurz eingehen werden.

Konfiguration der Systeme

Vor dem Genuss der Vorteile des DCOM ist eine kleine Einstiegshürde zu nehmen: die Konfiguration der PCs, auf die die Software verteilt werden soll. Aus bekannten Sicherheitsgründen ist es nämlich oft nicht wünschenswert, dass auf einen Rechner beliebige Zugriffe aus dem Netz möglich sind, und so müssen Sie den Zugriff auf COM-Objekte von anderen Rechnern erst erlauben. Bei unzureichender Konfiguration der Sicherheitseinstellungen kann es auf der Seite der Client-Anwendungen zu den verschiedensten Fehlermeldungen kommen (»Interface not supported«, »Access denied«, »Exception EIntfCastError«).

Zentrale Stelle für diese Konfiguration ist das Utility `dcomcnfg.exe` (Abbildung 8.16), das Sie, falls Sie keine schnellere Verknüpfung darauf haben, direkt aus dem Start-Menü über AUSFÜHREN erhalten können. In diesem Utility können Sie systemweit geltende Voreinstellungen eintragen (die in diesem Dialog aufgeführten Einstellungen gelten zunächst einmal für alle DCOM-Anwendungen), aber auch jeden registrierten COM-Server einzeln konfigurieren (auf der Seite *Anwendungen* des Hauptdialogs können Sie für jeden Server einen weiteren Dialog mit individuellen Einstellungen öffnen).

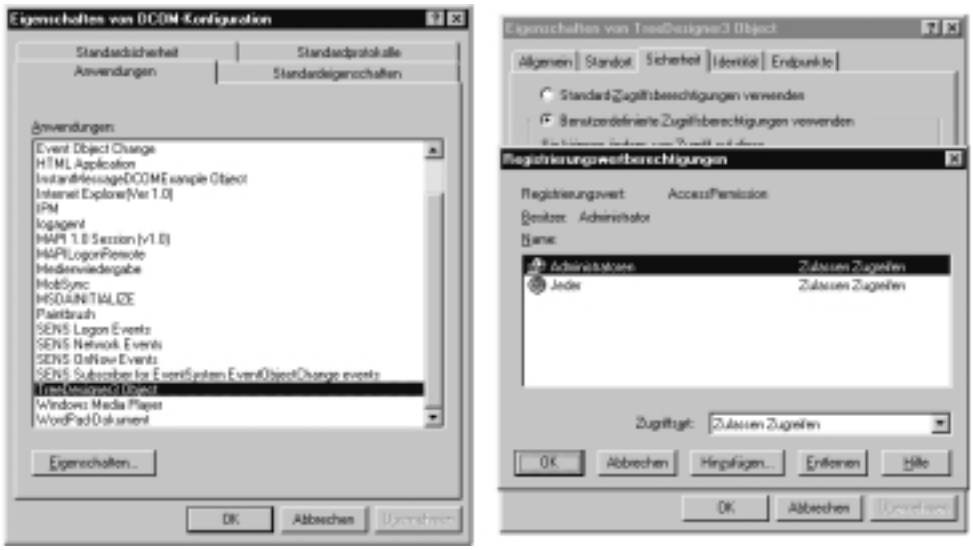


Abbildung 8.16: Konfiguration der DCOM-Optionen mit DComCnfg.exe von Windows NT/2000

Die Seite *Standardsicherheit* befasst sich unter anderem mit zwei wichtigen Arten von Benutzungsrechten:

- die Startberechtigung, die man braucht, falls eine DCOM-Server-Anwendung nicht bereits gestartet ist,
- die Zugriffsberechtigung, die bei laufender Server-Anwendung für alle weiteren Methodenaufrufe erforderlich ist.

Die Berechtigung zu einer bestimmten Aktion kann erst ermittelt werden, wenn die Identität des Aufrufers festgestellt wurde. Und hier kommt ein weiterer Aspekt ins Spiel: die Authentifizierung, durch die sichergestellt werden kann, dass der Aufrufer wirklich der ist, der er zu sein vorgibt. Windows NT/2000 bieten in DCOMCNFG beispielsweise sieben verschiedene Authentifizierungsstufen an (für die systemweite Einstellung sind diese auf der Seite *Standardeigenschaften* von *DComCnfg* zu finden).

Wenn Sie zwei PCs beispielsweise mit Windows 2000 über ein einfaches Netzprotokoll wie NetBEUI verbunden haben, die beiden Computer Teil derselben Arbeitsgruppe sind und Sie sich auf beiden unter demselben Namen und – besonders wichtig – mit demselben Passwort angemeldet haben, sollte es eigentlich problemlos möglich sein, von einem PC Zugriff auf die COM-Objekte des jeweils anderen PCs zu erhalten. Dazu genügt es, wenn Ihr Benutzerkonto auf dem PC, der als COM-Server fungieren sollen, sowohl unter Start- als auch unter Zugriffsberechtigung eingetragen ist.

Ziel dieses Buches ist jedoch nicht die Einweisung in die administrativen Funktionen diverser Windows-Versionen, daher soll hier nur noch eine Notlösung beschrieben

werden, mit der Sie die DCOM-Sicherheitsfunktionen für einzelne Beispielprogramme abschalten können, damit Sie in jedem Fall Zugriff erhalten.

- Server-PC *und* Client-PC: In den für alle COM-Anwendungen geltenden Einstellungen (STANDARDEIGENSCHAFTEN in DCOMCNFG) wird die Authentifizierungsebene auf *(kein)* herabgesetzt. Die ansonsten festgelegten Rechtebeschränkungen können aufrechterhalten werden. Fehler in dieser Einstellung können unter anderem zur Exception *EIntfCastError* in der Client-Anwendung führen.
- Server-PC: Die zu startende Anwendung wird aus der Liste der Seite *Anwendungen* ausgewählt (z.B. unter dem Namen »TreeDesigner3 Object«) und der Eigenschaftsdialog wird aufgerufen. Hier können Sie unter *Sicherheit* für *Zugriffsberechtigung* und *Startberechtigung* die Benutzergruppe *Jeder* eintragen.
- Außerdem sollten Sie sicherstellen, dass DCOM sowohl auf dem Client als auch auf dem Server überhaupt aktiviert ist (entsprechendes Markierungsfeld auf der Seite *Standardeigenschaften*).

Darüber hinaus gelten die schon von COM bekannten Voraussetzungen: Der COM-Server muss auf dem Rechner, auf dem er ausgeführt werden soll, registriert sein.

Hinweis: Die TreeDesigner-Automationsfunktionen, die auch über DCOM verfügbar sind, enthalten keine Möglichkeit zum Speichern oder Laden des aktuellen Dokuments, womit der ferngesteuerte Zugriff auf die Festplatte des Server-Rechners eigentlich ausgeschlossen sein und es kein Sicherheitsrisiko bedeuten sollte, den TreeDesigner in der jetzigen Version als frei verfügbaren COM-Server anzubieten (sieht man einmal von böswilligen Clients ab, die versuchen, den Speicher zu füllen oder ganz viele Meldungsfenster zu öffnen).

Die Identität des Benutzers

Für das Beispiel des TreeDesigners ist es in jedem Fall wichtig, in DCOMCNFG eine kleine Einstellungsänderung vorzunehmen. Rufen Sie dazu auf der Seite *Anwendungen* den Einstellungsdialog für *TreeDesigner3 Object* auf und markieren Sie auf der Seite *Identität* das Feld *Benutzer, der die Anwendung startet*. Ansonsten steuern Sie den TreeDesigner zwar vom Client-PC aus, können ihn aber nicht auf dem Bildschirm des Server-PCs sehen (höchstens als Eintrag in der Prozessliste des Task-Managers[23]).

23 Diese Beschreibung setzt Windows NT 4.0 oder Windows 2000 als Server-Plattform voraus, andere Betriebssysteme können abweichende Eigenschaften aufweisen.

Aufruf entfernter Objekte

Im bisherigen Verlauf von Kapitel 8.7 wurden mehrere Arten vorgestellt, einen COM-Server zu starten: über den Aufruf von *CreateOleObject* im Programmcode, über den Konstruktor der von Delphi erzeugten Co-Klassen-Hüllklasse (z.B. *CoTreeDesigner.Create*) und über die Verwendung einer von *TOleServer* abgeleiteten Komponente (wie z.B. *TWordApplication* oder *TWebBrowser*). Zu all diesen Aufrufarten gibt es Erweiterungen, die die Erzeugung des Objekts auf einem entfernten Rechner erlauben. In jedem Fall wird der Name des entfernten Rechners benötigt (*MachineName*) – entweder so, wie er im Windows-Explorer unter dem Knoten NETZWERK genannt wird, oder als IP-Adresse (je nach Systemkonfiguration). Ansonsten gibt es nur wenige Unterschiede zum Erzeugen eines lokalen Objekts:

- Statt Co[KlassenName].Create verwenden Sie Co[KlassenName].CreateRemote-MachineName), also z.B. CoTreeDesigner.CreateRemote('PC9').
- Anstelle von *CreateOleObject* verwenden Sie die Funktion *CreateRemoteComObject*, welche ebenfalls in der VCL-Unit *ComObj* enthalten ist. Aufrufformat und Ergebnistyp dieser Funktion weichen allerdings von *CreateOleObject* ab. Außer *MachineName* geben Sie als Parameter die *ClassID* des zu erzeugenden Objekts an (statt eines Strings wie bei *CreateOleObject*), und der Rückgabetyp ist nicht *IDispatch*, sondern ein ganz allgemeiner *IUnknown*-Zeiger, der erst manuell in das gewünschte Interface (z.B. *IDispatch*) umgewandelt werden muss. Ein Beispiel für den Aufruf von *CreateRemoteComObject* finden Sie in jeder von Delphi erzeugten `xxx_TLB.pas`-Datei in der Implementierung der bereits erwähnten Methode *CreateRemote*.
- Bei Verwendung einer *TOleServer*-Komponente verändern Sie zwei ihrer Properties. Setzen Sie *ConnectKind* auf *ckRemote* und *RemoteMachineName* auf den Namen bzw. die Adresse des Rechners, auf dem das Objekt erzeugt werden soll.

Alle diese drei Alternativen verschonen Sie noch von der Verwendung der API-Funktionen, denn sie sind entweder in Delphis Unit *ComObj* oder in der von Delphi erzeugten Datei `xxx_TLB.pas` implementiert. Für viele Anwendungsfälle dürften diese drei Alternativen bereits ausreichen.

In manchen Fällen erreichen Sie allerdings durch andere Vorgehensweisen eine erhöhte Effizienz, so z.B. durch die Verwendung von Class Factories zur Instanziierung *mehrerer Objekte* der gleichen Klasse oder durch die Verwendung der Windows-API-Funktion *CoCreateInstanceEx* zur gleichzeitigen Lieferung *mehrerer Schnittstellen* von einem entfernten Server. Die Verwendung von Windows-API-Funktionen verträgt sich übrigens ohne weiteres mit den oben genannten VCL-Funktionen. Beispiele für ihren Aufruf finden Sie im Quelltext der VCL selbst, die ja intern die API-Funktionen von Windows verwenden muss.

Die Dokumentation dieser API-Funktionen ist Teil der Win32-Hilfe von Delphi (HILFE | WINDOWS-SDK unter Delphi 6, falls installiert). Für eine noch detailliertere Kontrolle (oder zur Fehlersuche) ist es außerdem möglich, die von *CoCreateInstanceEx* intern verwendeten API-Funktionen *CoCreateClassObject* und *IClassFactory.CreateInstance* selbst aufzurufen.

Hinweis: Wie schon erwähnt können Sie Ihr System auch so konfigurieren, dass der COM-Client nicht wissen muss, dass das Server-Objekt auf einem anderen Rechner läuft. Die Delphi-Anwendung kann also beispielsweise statt *CoTreeDesigner.CreateRemote* weiter *CoTreeDesigner.Create* aufrufen, und trotzdem wird das Objekt auf einem entfernten Server gestartet. Dafür muss dieser COM-Server auf dem Client-Rechner registriert sein (auf dem Server selbstverständlich ebenfalls). Stellen Sie dann unter `dcomcnfg.exe` auf diesem Client-Rechner für diesen COM-Server die Option *Anwendung auf dem folgenden Computer ausführen* ein und geben Sie den Namen des Server-Computers an. Sie brauchen dann die im weiteren Verlauf beschriebenen Maßnahmen, um das Objekt explizit auf einem anderen Computer zu erstellen, nicht zu treffen.

Der TreeDesigner als Komponente

Um das Beispiel der TreeDesigner-Automation aus Kapitel 8.7.3 fortzusetzen, befassen sich die folgenden Abschnitte mit einer dritten Version des Automations-Clients *TdCtrl3* und mit einer erweiterten Automationsschnittstelle des TreeDesigners (Kapitel 8.7.5 und 8.7.6).

In Kapitel 8.2.5 wurde gezeigt, wie *TdCtrl2* das TreeDesigner-Automationsobjekt mit *CoTreeDesigner2.Create* initialisiert hat. *TdCtrl3* geht einen anderen Weg und spricht die TreeDesigner-Automationsschnittstelle über eine COM-Server-Hüllkomponente an, ähnlich der in Kapitel 8.6.2 verwendeten Hüllkomponenten der Word-Schnittstellen (gemeinsame Basisklasse all dieser Hüllkomponenten ist *TOleServer*). Um diese Hüllkomponente von Delphi erzeugen zu lassen, wird der Menüpunkt PROJEKT | TYPBIBLIOTHEK IMPORTIEREN der Delphi-IDE ausgeführt und das TreeDesigner-Automationsobjekt zum Import ausgewählt. Delphi installiert daraufhin eine neue Hüllkomponente für dieses Objekt in der Palette. Das Formular von *TdCtrl3* enthält ein Exemplar dieser Komponente, welches in den späteren Listings unter dem Namen *TreeDesigner3* auftaucht. Die Vorteile der Verwendung einer Hüllkomponente werden allerdings erst in Kapitel 8.7.6, wenn es um die Events gehen wird, richtig klar werden.

Des Weiteren ist es für das Verständnis von *TdCtrl3* wichtig zu wissen, dass sein Hauptformular (Abbildung 8.17) per Formularvererbung auf dem Hauptformular von *TdCtrl2* basiert. Die Funktion der Schalter *Alle Objekte löschen*, *Neues Dokument*, *Neues Grafikobjekt* und *Objekt ändern* wird von *TdCtrl2* geerbt und die zugehörigen Ereignis-

bearbeitungsmethoden können unverändert auch in *TdCtrl3* verwendet werden, wenn eine Voraussetzung erfüllt ist: Die *TreeDesigner*-Variable vom Interface-Typ *ITreeDesigner2* (z.B. verwendet in *TTDControllerForm.UpdateList*, Kapitel 8.7.3) muss gültig sein. Damit alle von *TdCtrl2* geerbten Methoden ohne Probleme auf diese Variable zugreifen können, weist das abgeleitete *TdCtrl3*-Formular ihr das neue *TreeDesigner3*-Objekt zu (siehe Listing im nächsten Abschnitt).

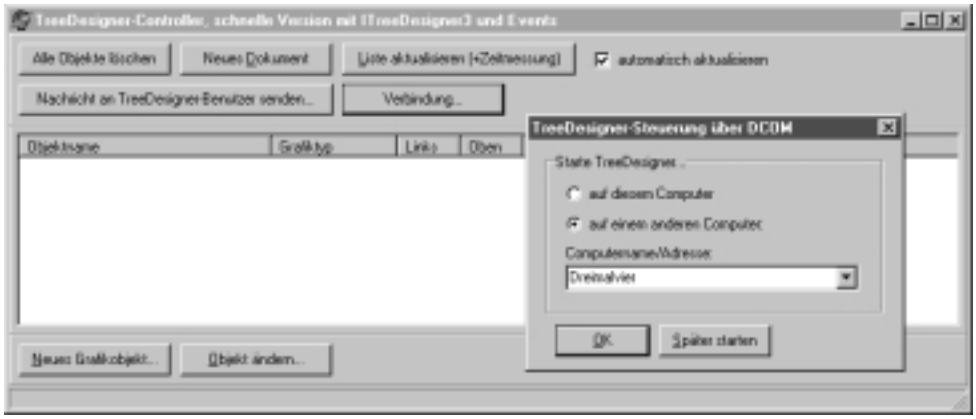


Abbildung 8.17: Das Hauptformular von TdCtrl3 und der Dialog zum Erzeugen eines COM-Automations-Objekts

Ein Dialog für die Verbindungsaufnahme

TdCtrl3 unterscheidet sich von *TdCtrl2* außerdem dadurch, dass es den TreeDesigner auch auf einem anderen Rechner starten kann. Es gestattet dem Benutzer, in einem Dialog anzugeben, ob er eine lokale oder eine entfernte Instanz des TreeDesigners wünscht. Der Dialog verwendet die in Kapitel 6.5.3 beschriebene Komponente *THistoryCombo*, um die bisher eingegebenen Computernamen zwischen den Sitzungen in der Registry zu speichern und dem Benutzer eine wiederholte Eingabe desselben Namens zu ersparen.

Der Dialog wird gleich beim Programmstart aufgerufen, die Auswahl kann aber per Schalter SPÄTER STARTEN auf einen späteren Zeitpunkt verschoben werden. In diesem konkreten Beispiel ist dies sicher nur zur Demonstration des Verbindungsdialogs sinnvoll, da das Programm ohne den TreeDesigner gar nichts machen kann.

Da die Verbindung zu einem anderen Computer einige Sekunden Zeit in Anspruch nehmen kann, öffnet *TdCtrl3* bei der Verbindungsaufnahme ein kleines Meldungsfenster, das dem Benutzer den Grund dieser Wartezeit mitteilt. Das Meldungsfenster darf allerdings nicht mit *ShowMessage* oder *ShowModal* geöffnet werden, da diese Aufrufe das Programm anhalten würden und die Verbindung zum COM-Server erst nach dem Schließen dieses Fensters aufgebaut werden könnte. *TdCtrl3* besitzt daher ein weiteres

Formular namens *PauseForm*, das mit *Show* angezeigt wird, das aber trotzdem nicht wie ein modales Formular in den Hintergrund gerückt werden kann, da seine Freigabe mit *Free* schon erfolgt, bevor eventuelle Eingaben des Benutzers verarbeitet werden können (Eingaben werden zwar sofort nach dem Öffnen des Fensters in *ProcessMessages* verarbeitet, dies dürfte aber für den Benutzer zu schnell sein):

```
ModalResult := LogonForm.ShowModal;
if ModalResult = mrOK then begin
  if LogonForm.RadioRemoteComputer.Checked then begin
    TreeDesigner3.ConnectKind := ckRemote;
    TreeDesigner3.RemoteMachineName := LogonForm.ComputerName.Text;
  end;
  PauseForm := TPauseForm.Create(nil);
  PauseForm.Show;
  Application.ProcessMessages; // für das Zeichnen des Meldungs-Labels
  try
    TreeDesigner := // Diese Interface-Variable wird nur in den von
                    // TdCtrl2 geerbten Methoden verwendet.
      // Der folgende Zugriff führt implizit
      // zum Aufruf von TreeDesigner3.Connect:
      TreeDesigner3.DefaultInterface as ITreeDesigner2;
  finally
    PauseForm.Free;
  end;
end else if ModalResult = ...
```

Wichtig ist, dass das Programm an anderen Stellen nicht einfach so auf das *TreeDesigner3*-Objekt zugreift, denn dabei würde selbst im Modus *AutoConnect = False* der von Delphi automatisch erzeugte Code dazu führen, dass das TreeDesigner-Objekt initialisiert wird, ohne dass der Verbindungsdialog vorher angezeigt wird.

Vor jedem Zugriff auf *TreeDesigner3* muss daher überprüft werden, ob die Verbindung bereits besteht. Interessanterweise verfügt aber eine *TOleServer*-Komponente wie *TreeDesigner3* über keinerlei Möglichkeit, dies festzustellen, denn spätestens beim ersten Zugriff wird automatisch eine Verbindung hergestellt. Das Programm muss sich also selbst merken, ob es bereits Verbindung aufgenommen hat. Hierzu braucht es nur nachzusehen, ob der vom Basisformular geerbten Interface-Variablen *TreeDesigner* bereits ein Wert zugewiesen wurde:

```
if not Assigned(TreeDesigner) then begin
  ShowMessage(TreeDesignerStartHinweis);
  exit;
end;
```

Die virtuelle Initialisierungsmethode *InitTreeDesigner* von *TdCtrl2*, in der die Automationsschnittstelle mit *CoTreeDesigner2.Create* initialisiert wurde (Kapitel 8.7.3), muss natürlich in *TdCtrl3* überschrieben werden, da hier statt *CoTreeDesigner2* die *TOleServer*-Komponente aus der Komponentenpalette verwendet werden soll. In *TdCtrl3* soll

diese Initialisierung allerdings nur nach Abschluss des Benutzerdialogs gestartet werden. Da hierfür bereits der oben gezeigte Code zuständig ist, hat *InitTreeDesigner* in *TdCtrl3* nichts mehr zu tun, sie wird daher durch einen leeren Methodenrumpf ersetzt:

```
procedure TTDControllerForm3.InitTreeDesigner; { override }
begin
end;
```

8.7.5 Übertragung beliebiger Datenstrukturen

Dieser Abschnitt stellt die Technik der Variant-Arrays vor, mit der sich praktisch beliebige Datenstrukturen auch über Anwendungs- und Rechnergrenzen hinweg transportieren lassen und mit der sich der Datenaustausch zwischen Server und Client erheblich beschleunigen lässt. Auch diese Technik soll am Beispiel des TreeDesigners demonstriert werden. Ziel ist es, dass der COM-Client nicht mehr jedes einzelne Datenelement eines jeden Grafikelements einzeln abfragen muss, sondern alle Daten aller Grafikelemente auf einen Schlag in seinen Speicherbereich kopiert werden. Die Daten sollen in folgender Struktur abgelegt werden:

```
TElementData = record
  Points: TRect; // Koordinaten des Elements
  ElementType: TShapeType; // Form des Elements
  Text: String[GrObjectTextLen]; // Beschriftung
end;
```

Für mehrere Elemente sollen die Daten in einem Array aus *TElementData*-Records abgelegt werden. Wenn Sie die Klasse *TGraphicElement* aus Kapitel 5.3.3 kennen, werden Sie feststellen, dass nur ein kleiner Teil aller *TGraphicElement*-Daten im Record gespeichert wird, aber hier soll es auch nur um das Prinzip gehen.

Hinweis: Wollten wir den TreeDesigner zu einem »voll ausgewachsenen« COM-Server nach Art des Internet Explorers ausbauen, würden wir wahrscheinlich nicht nur das Dokument, sondern auch die Grafikelemente über COM-Interfaces zur Verfügung stellen. Der Client würde dann beispielsweise über eine *IGraphicElement*-Schnittstelle die einzelnen Daten abfragen. Dies wäre allerdings nicht wesentlich effizienter als die bisher verwendete Abfragetechnik, und die Typenbibliothek und die COM-Implementierung des TreeDesigners würden so stark wachsen, dass der Rahmen eines Beispielprogramms bei weitem gesprengt werden würde.

Erweiterung bestehender Interfaces *R136*

Bevor wir zur anvisierten Übertragung der Record-Daten kommen, beobachten wir eine interessante Situation: In Kapitel 8.7.1 wurde der TreeDesigner mit der Schnittstelle *ITreeDesigner2* versehen (siehe Abbildung 8.15), und nun stehen wir vor dem Ziel, dem TreeDesigner eine neue Methode zu verpassen, die alle Grafikdaten in einem

Array zurückliefert. Die Frage ist nun, ob das bestehende Interface *ITreeDesigner2* erweitert werden kann, soll oder darf.

Da *ITreeDesigner2* bereits aus einer älteren Programmversion (dem TreeDesigner 2.0) stammt, sollten wir, auch wenn es nur ein Beispielprogramm ist, annehmen, dass diese alte Schnittstelle bereits verwendet wird. Und ein wichtiges Grundprinzip vom COM besteht darin, dass bereits definierte Schnittstellen nicht mehr verändert werden sollten. Im Internet Explorer sehen Sie dieses Prinzip beispielsweise darin verwirklicht, dass dieser die Schnittstellen *IHTMLDocument*, *IHTMLDocument2* und *IHTMLDocument3* unterstützt. Dabei sind die Schnittstellen mit der höheren Zahl jeweils in späteren Programmversionen hinzugekommen.

Der TreeDesigner soll diese Vorgehensweise nun »nachmachen«. In der Version 3.0 wird daher folgendes Zusatz-Interface definiert, das außerdem von *ITreeDesigner2* abgeleitet wird, so dass alle alte Methoden auch über das neue Interface aufgerufen werden können:

```
ITreeDesigner3 = interface(ITreeDesigner2)
  ['{586633BF-5BCF-11D3-A0B1-0080AD80B083}']
  function  GetCurrentDocData: OleVariant; safecall;
  procedure UserMessage(const text: WideString); safecall;
  procedure Quit; safecall;
end;
```

Die erste neue Methode wird in diesem Abschnitt erläutert, sie soll die angesprochenen Record-Daten liefern. Zur zweiten Methode wird Kapitel 8.7.6 genauer Stellung nehmen. *Quit* bewirkt lediglich, dass *Application.Terminate* den TreeDesigner schließt.

Das Interface wird wie gewohnt in der TLB des TreeDesigners definiert, in der bereits *ITreeDesigner2* enthalten ist. Sie können eine bestehende TLB immer mit DATEI | ÖFFNEN in den TLB-Editor laden oder mit F12 (UMSCHALTEN FORMULAR/UNIT), wenn Sie sich im Quelltexteditor bei der zugehörigen Pascal-Unit befinden. In diesem Fall erhalten Sie den Editor jedoch auch, wenn Sie mit DATEI | NEU | WEITERE, ACTIVEX, AUTOMATISIERUNGSOBJEKT ein neues Objekt namens *TreeDesigner3* und gleichzeitig eine neue Implementierungs-Unit erzeugen.

Im TreeDesigner heißt diese Unit *TDAuto3*. Die Deklaration der Co-Klasse wurde von Hand so angepasst, dass sie von der bereits in Kapitel 8.7.1 erstellten Klasse *TTreeDesigner2* (Unit *TDAuto*) abgeleitet wird und daher die Implementierung der Methoden für die Schnittstelle *ITreeDesigner2* erbt:

```
TTreeDesigner3 = class(TDAuto.TTreeDesigner2, ITreeDesigner3)
protected
  function GetCurrentDocData: OleVariant; safecall;
  procedure UserMessage(const Text: WideString); safecall;
// Gekürzt. Weitere Details kommen in Kapitel 8.7.6 hinzu.
end;
```

Hinweis: Wenn Sie die TreeDesigner-Typenbibliothek mit PROJEKT | TYPBIBLIOTHEK IMPORTIEREN »re-importieren«, um eine COM-Server-Komponente für den TreeDesigner in der Komponentenpalette zu erhalten, erzeugt Delphi für jede Schnittstelle eine Komponente. Sie erhalten also sowohl eine *TTreeDesigner2*- als auch eine *TTreeDesigner3*-Komponente. Der bereits im letzten Abschnitt vorgestellte Beispiel-Client *TdCtrl3* verwendet davon natürlich die Komponente mit dem aktuellen Interface, *TTreeDesigner3*.

Varianten-Arrays *R126*

Nun geht es um die anfangs beschriebene Übertragung der *TElementData*-Records. Da COM den Typ *TElementData* nicht kennt, müssen die Daten in einer COM-kompatiblen Form verpackt werden. Hierzu bietet sich wieder der Typ *OleVariant* an. Grob zusammengefasst läuft die Übertragung der Records wie folgt ab:

- Die Record-Daten werden als Folge von Bytes aufgefasst, und die Folge von Bytes wird als Array interpretiert.
- Die *OleVariant*-Variable wird so geformt, dass sie ein Array von Bytes aufnehmen kann, genauer gesagt ein Array von *OleVariant*-Variablen, die jeweils einen *Byte*-Wert speichern.
- Das Array von *Byte*-Varianten wird im Speicher so festgehalten, dass ein Array von echten Bytes entsteht.
- Die Bytes der Records werden schnell in diesen Speicherbereich hineinkopiert, z.B. über Zuweisung oder wie im Beispiel über die Kopierprozedur *Move*.
- Nun wird die im vorletzten Schritt vorgenommene Fixierung aufgehoben, und die OleVariant wird wieder zu einem Variant-Array mit Byte-»Untervarianten«.
- Dieses Variant-Array wird (als äußerlich unauffällige *OleVariant*) an den COM-Client übermittelt.
- Der COM-Client liest nun in einer umgekehrten Reihenfolge die Record-Daten wieder aus.

Während der erste Schritt ein rein mentaler Schritt im Kopf des Entwicklers ist, entsprechen die weiteren Schritte jeweils einem Prozeduraufruf des folgenden Programmcodes: *VarArrayCreate*, *VarArrayLock*, *Move*, *VarArrayUnlock* und schließlich die Zuweisung des Ergebnisses an *Result*.

Vor dem Aufruf von *Move* muss allerdings noch ein *TElementData*-Record vorbereitet werden, der als Quelle der Kopieraktion dienen kann. Da der TreeDesigner bei seiner normalen Arbeit keine *TElementData*-Records verwaltet, wird für jedes Grafikelement jeweils ein solcher Record »on the fly« zusammengestellt und dann in den Speicher-

bereich der Variante kopiert. Dieser Speicherbereich wird vorher (nur für die Zufriedenstellung des Compilers) in ein *TByteArray* umdeklariert. Die Records sollen hintereinander in das Byte-Arrays kopiert werden, weshalb der Startindex im Byte-Array jedes Mal neu berechnet werden muss.

```
// uses Variants; <- ab Delphi 6 muss diese Unit eingebunden werden.

type
  TByteArray = array[0..0] of Byte;

function TTreeDesigner3.GetCurrentDocData: OleVariant;
var
  ArrayVariant: OleVariant;
  Doc: TGraphicDoc;
  ElementIndex: Integer;
  VariantMemory: ^TByteArray;
  OneRec: TElementData;
  Element: TGraphicElement;
begin
  Doc := (MainForm.ActiveMDIChild as TDocumentForm).Document;
  ArrayVariant := VarArrayCreate([0, Doc.Count*sizeof(TElementData)],
                                 varByte);
  VariantMemory := VarArrayLock(ArrayVariant);
  for ElementIndex := 0 to Doc.Count-1 do begin
    with Doc[ElementIndex] do begin
      GetOriginalRect(OneRec.Points);
      OneRec.ElementType := ShapeType;
      OneRec.Text := Text;
    end;
    Move(OneRec, VariantMemory^[ElementIndex*sizeof(TElementData)],
         sizeof(TElementData));
  end;
  VarArrayUnlock(ArrayVariant);
  Result := ArrayVariant;
end;
```

Die genauere Funktionsweise der Varianten-Arrays ist wie folgt (zu Varianten allgemein siehe Kapitel 8.6.1):

- *VarArrayCreate* erzeugt eine Variante mit dem Typencode *varArray*. Der erste Parameter gibt an, wie groß das Array dimensioniert werden soll, und zwar schreiben Sie wie bei der Deklaration eines Arrays einfach den Start- und den Endindex in eckige Klammern. Der zweite Parameter gibt den Typ der Array-Elemente an. Im vorliegenden Beispiel benötigen wir Bytes, was dem Typencode *varByte* entspricht. Alle Typencodes finden Sie in der Online-Hilfe unter *TVarData*. So ist es beispielsweise durch Verwendung des Typencodes *varArray* möglich, geschachtelte bzw. mehrdimensionale Arrays zu erzeugen. Ein einmal eingerichtetes Array kann übrigens später mit *VarArrayRedim* in seiner Größe verändert werden.

- *VarArrayLock* erlaubt es, direkt auf den Speicher eines Varianten-Arrays zuzugreifen, ohne die normalerweise bei Zuweisung an Varianten stattfindenden zeitraubenden Typüberprüfungen und -konvertierungen durch die Laufzeitbibliothek. *VarArrayLock* liefert einen Zeiger auf einen Speicherbereich, indem sich die Daten der Variante wie in einem normalen Array wiederfinden. Der Zeiger ist untypisiert (Typ *Pointer*) und muss eventuell erst wie im obigen Beispiel in einen typisierten Zeiger umgewandelt werden.
- *VarArrayUnlock* hebt die Wirkung von *VarArrayLock* auf, der von letzterer Funktion zurückgelieferte Zeiger wird ungültig und die Variante kann wieder über *VarArrayRedim* umdimensioniert werden.

Empfang des Arrays im Client

Der Automations-Client *TdCtrl3* empfängt nun also von der Methode *GetCurrentDocData* das oben konstruierte Varianten-Array. Um es effizient auszulesen, lässt auch er sich von *VarArrayLock* einen Speicherbereich zur Verfügung stellen, aus dem er nun mit *Move* in umgekehrter Richtung die Speicherbereiche der einzelnen *TElementData*-Records in eine *TElementData*-Variable kopieren kann. Die einzelnen Daten werden dann wie in der Progammversion *TdCtrl2* in einem ListView angezeigt, der hierfür zuständige Code wurde im Folgenden abgekürzt, da er nichts Neues mehr bringt:

```
procedure TTDControllerForm3.UpdateList; { override }
var
  CurrentDocData: OleVariant;
  CurrentDoc: IDocAuto2;
  OneRec: TElementData;
  i: integer;
  Item: TListItem;
  VariantMemory: ^TByteArray;
begin
  if not Assigned(TreeDesigner) then exit;
  Screen.Cursor := crHourGlass;
  CurrentDocData := (TreeDesigner as ITreeDesigner3).GetCurrentDocData;
  CurrentDoc := TreeDesigner.GetCurrentDocI;
  VariantMemory := VarArrayLock(CurrentDocData);
  with ListView1 do begin
    Items.Clear;
    for i := 0 to CurrentDoc.GetObjectCount-1 do begin
      Move(VariantMemory^[i*sizeof(TElementData)],
           OneRec, sizeof(TElementData));
      Item := Items.Add; // neuer ListView-Item
      Item.Caption := OneRec.Text;
    ...
 VarArrayUnlock(CurrentDocData);
 ...
```

Geschwindigkeitsmessung

Bereits in *TdCtrl2* ist eine kleine Zeitmessfunktion in den *Aktualisieren*-Schalter implementiert, die mit Hilfe der API-Funktion *GetTickCount* die Zahl der Millisekunden für das Aktualisieren des ListViews – genauer: für die Ablaufdauer von *UpdateList* - misst und in der Statuszeile ausgibt. Da *UpdateList* virtuell ist, funktioniert diese Zeitmessung auch im abgeleiteten Formular von *TdCtrl3*.

Während bereits in Kapitel 8.7.3 der Umstieg auf die Interfaces für die Methodenaufrufe eine Geschwindigkeitsverdopplung brachte, haben wir es nun bei der Übertragung aller Daten in einem Varianten-Array mit einer noch größeren Beschleunigung zu tun. Bei einer Beispielgrafik mit mehreren hundert Elementen benötigte *TdCtrl3* nur ein Viertel der Zeit von *TdCtrl2* für die Neuausgabe der Liste. Und dieser Test fand auf einem einzelnen Rechner statt. Wenn der COM-Server auf einem anderen Computer läuft, dürfte der Geschwindigkeitsvorteil noch größer sein, da hier die Methodenaufrufe zwischen den Rechnern noch mehr Zeit in Anspruch nehmen und die Reduktion auf einen einzigen Methodenaufruf (*GetCurrentDocData*) folglich mehr Zeit einsparen kann.

Hinweis: Die DataSnap-Technologie der Enterprise-Version von Delphi für mehrschichtige Datenbankanwendungen verwendet intern ebenfalls Varianten-Arrays, um Datenmengen über DCOM zwischen der Server-Anwendung (der Mittelschicht) und dem Client hin- und herzubewegen. Umgekehrt gedacht können Sie sich bei einfachen Problemstellungen den Einsatz der Enterprise-Version von Delphi ersparen, indem Sie die Daten selbst zwischen Client und Server übertragen (in Delphi 5 fällt diese Funktionalität noch unter den Begriff »MIDAS«).

8.7.6 COM-Objekte mit Events

Nachdem Kapitel 8.6.3 bereits Beispiele für die Bearbeitung von Events des Internet Explorers gegeben hat, geht es jetzt darum, selbst ein Automationsobjekt zu entwickeln, das seinem Client Events zur Verknüpfung anbietet. Delphi vereinfacht die Programmierung von COM-Events erheblich. Während diese Events auf der Ebene des COM ziemlich kompliziert unter Zuhilfenahme verschiedener Schnittstellen wie *IConnectionPoint*, *IConnectionPointContainer* und *IEventSink* realisiert werden, brauchen Sie in Delphis Automatisierungsobjekt-Experten (DATEI | NEU | WEITERE – ACTIVE X, AUTOMATISIERUNGS-OBJEKT) lediglich das Feld EREIGNISUNTERSTÜTZUNG GENERIEREN zu markieren, um Delphi die gesamte Verwaltungsarbeit erledigen zu lassen.

Ihre Arbeit beginnt dann bei einem neuen Interface im TLB-Editor, das den Namen der Automatisierungsschnittstelle mit angehängtem *Events* trägt. In dieser Schnittstelle tragen Sie alle Events, die Ihr COM-Objekt bei seinem Client auslösen kann, als Methoden ein.

Ein neues Interface für die TreeDesigner-Events

Auch hier soll wieder der TreeDesigner als Beispiel dienen. In Kapitel 8.7.5 wurde bereits ein neues Automationsobjekt namens *TreeDesigner3* erzeugt, welches nun auch die Events unterstützen soll. Dazu wurde bei der Erzeugung des COM-Objekts durch den Experten *Ereignisunterstützung generieren* gewählt. Das zunächst leere Interface für die Events wurde wie folgt erweitert:

```
ITreeDesigner3Events = dispinterface
  ['{586633C1-5BCF-11D3-A0B1-0080AD80B083}']
  procedure OnUserMessage(const text: WideString); dispid 3;
  procedure OnElementChanged(ElementIndex: Integer); dispid 1;
  procedure OnOrderChanged(Element1: Integer; Element2: Integer);dispid 5;
  procedure OnNewElement(ElementIndex: Integer); dispid 6;
  procedure OnElementDeleted(ElementIndex: Integer); dispid 7;
  procedure OnDocumentInvalidated; dispid 4;
  procedure OnQuit; dispid 2;
end;
```

Die meisten der Events sollten dazu dienen, den Client über Änderungen im Dokument zu informieren. *TdCtrl3* erhält dann beispielsweise, wenn der Benutzer des TreeDesigners ein Grafikelement verschiebt, ein *OnElementChanged*-Ereignis, das den Index des verschobenen Elements angibt. Mit diesem können die neuen Elementdaten ermittelt werden, und im ListView von *TdCtrl* muss nur eine einzige Zeile aktualisiert werden (in *TdCtrl2* gab es als einzige Möglichkeit der Aktualisierung, das gesamte ListView neu auszugeben).

Das Ereignis *OnUserMessage* ist dafür gedacht, wenn TreeDesigner und COM-Client auf verschiedenen Rechnern laufen. Der Benutzer des TreeDesigners kann dann den Menüpunkt HILFE | KONTAKT MIT FERNSTEUERUNG... aufrufen und eine Nachricht an den Client senden. Diese Funktion wird am Ende des Kapitels erläutert.

Das letzte Event, *OnQuit*, tritt erwartungsgemäß ein, wenn der Benutzer den TreeDesigner schließt.

Hinweis: Die im Programmcode erzeugten Änderungsereignisse sollen hauptsächlich das Prinzip veranschaulichen, decken aber nicht alle möglichen Änderungsaktionen im TreeDesigner ab. Das Event *OnDocumentInvalidated* dient dazu, eine komplette Aktualisierung des ListViews zu erzwingen, beispielsweise wenn der Benutzer des TreeDesigners zu einem anderen Dokumentfenster wechselt. In diesem Fall muss *TdCtrl3* unbedingt sein ListView neu ausgeben, weil sich die folgenden Änderungsereignisse alle auf das neu aktivierte Dokumentfenster beziehen. Diese Einschränkung auf das aktuelle Dokument ist für einen COM-Server natürlich keine gute Lösung, wurde hier aber eingegangen, um den Beispielcode möglichst einfach zu halten.

Generierung der Events

Während Sie bei der Definition der Automationsschnittstelle zu jeder neuen Interface-Methode eine Implementierungsmethode der Co-Klasse schreiben mussten (für die Delphi automatisch einen leeren Rumpf erzeugte), werden die Event-Bearbeitungsmethoden erst im Client implementiert. Was allerdings im COM-Server geschehen sollte, ist, diese Methoden aufzurufen, und hierzu müssen wir uns die von Delphi automatisch erzeugte Co-Klassen-Deklaration genauer ansehen. In Kapitel 8.7.5 wurden bereits die selbst definierten neuen Methoden *GetCurrentDocData* und *UserMessage* gezeigt, im Folgenden sehen Sie die komplette Klassendeklaration mit allen von Delphi erzeugten Elementen:

```
TTreeDesigner3 = class(TDAuto.TTreeDesigner2,
                       IConnectionPointContainer, ITreeDesigner3)
private
  { Private declarations }
  FConnectionPoints: TConnectionPoints;
  FEvents: ITreeDesigner3Events;
public
  procedure Initialize; override;
  property Events: ITreeDesigner3Events read FEvents; // Manuell erzeugt
protected
  property ConnectionPoints: TConnectionPoints read FConnectionPoints
    implements IConnectionPointContainer;
  procedure EventSinkChanged(const EventSink: IUnknown); override;
  function GetCurrentDocData: OleVariant; safecall;
  procedure UserMessage(const Text: WideString); safecall;
end;
```

Hieran lässt sich erkennen, dass Delphi zur Implementierung der Events tatsächlich die erwähnten *ConnectionPoints* verwendet, worum wir uns jedoch nicht weiter zu kümmern brauchen. Das Einzige, was wir zum Generieren der Events zur Laufzeit des Programms benötigen, ist die Interface-Variable *FEvents*. Im TreeDesigner ist es außerdem erforderlich, dieses Events-Interface von außerhalb der Klasse *TTreeDesigner3* anzusprechen, daher wurde zusätzlich zu den von Delphi erzeugten Elementen noch das Property *Events* manuell deklariert.

Die Erzeugung eines Events bzw. der Aufruf eines Event-Handlers sieht wie folgt aus:

```
procedure TGraphicElement.ElementChangeNotify;
var
  AutoObject: TTreeDesigner3;
begin
  AutoObject := TDAuto3.GetTD3AutoObject;
  if Assigned(AutoObject) and Assigned(AutoObject.Events) then
    AutoObject.Events.OnElementChanged(Document.IndexOf(self));
end;
```

Zuerst wird das Automationsobjekt aus der Unit *TDAuto3* ermittelt. Dann müssen zwei Bedingungen eintreffen, unter denen das Event nur erzeugt werden kann: Das Automationsobjekt muss existieren (es existiert nicht, wenn der TreeDesigner nicht durch einen COM-Client, sondern ganz normal vom Benutzer gestartet wurde), und sein *Events*-Property muss gesetzt sein. Dies ist nur dann der Fall, wenn sich der COM-Client überhaupt für die Events interessiert – ein charakteristisches Merkmal von Events ist ja, dass der potenzielle Empfänger sie gar nicht zu beachten braucht.

Hinweis: Selbst wenn der Client nur eines von vielen Events wirklich bearbeitet, muss er eine *IDispatch*-Schnittstelle zur Verfügung stellen, über die der Server dann auch alle anderen Events erzeugen kann – selbst wenn der Client auf diese nicht reagieren sollte. Wenn der Client eine Delphi-Anwendung mit *TOleServer*-Komponenten ist, wird dies von Delphi automatisch gehandhabt, so dass Sie im Objektinspektor wieder die Möglichkeit haben, einige Events unbearbeitet zu lassen.

TGraphicElement ruft *ElementChangeNotify* immer dann auf, wenn sich die Daten des Elements geändert haben, zum Beispiel in der Schreibmethode für das Property *Text*:

```
procedure TGraphicElement.SetText(s : TGrObjectText);
begin
  FText := s;
  InvalidateAllViews;
  ElementChangeNotify;
end;
```

Der Instanziierungsmodus

Die anderen TreeDesigner-Events werden analog zur obigen Methode *ElementChangeNotify* erzeugt. Was diesen Event-Aufruf allerdings vereinfacht hat, ist der Instanziierungsmodus des TreeDesigners, der im Automatisierungsobjekt-Experten auf EINFACHE INSTANZ (bzw. in der Implementierungs-Unit auf *ciSingleInstance*) eingestellt wurde. Dieser hat zur Folge, dass, falls sich mehrere COM-Clients für die Dienste des TreeDesigners interessieren sollten, automatisch mehrere Instanzen des TreeDesigners erzeugt werden, so dass jede einzelne Instanz es mit höchstens einem COM-Client zu tun hat und folglich auch nicht mehr als ein Automatisierungsobjekt der Klasse *TTreeDesigner3* braucht.

Im Instanziierungsmodus *ciMultiInstance* können aber in ein und derselben TreeDesigner-Instanz mehrere Objekte von *TTreeDesigner3* erzeugt werden, die möglicherweise alle ihren eigenen Client über die Änderung des Dokuments informieren sollen. Hierzu würde der TreeDesigner eine Liste all dieser Objekte benötigen und müsste dann diese Liste durchlaufen, um das Änderungsereignis auch an alle interessierten Clients weiterzugeben.

Bearbeiten der Events

Auf der Seite des Clients erscheinen die soeben geschaffenen Events auf die gleiche Weise wie bereits in Kapitel 8.6 die Events von Word und Internet Explorer. Sie brauchen nur die Komponente für den COM-Server im Formular auszuwählen und die Events über den Objektinspektor zu verknüpfen. Im vorliegenden Beispielprogramm werden bei diesen Events zumeist Aktualisierungen am ListView vorgenommen. Bei *OnElementChanged* wird eine Zeile des ListViews neu ausgegeben, bei *OnNewElement* wird der Einfachheit halber die gesamte Liste erneuert. All dies ist aber für dieses Kapitel nicht relevant.

Benutzer-Kommunikation zwischen zwei Computern

Das Ereignis *OnUserMessage* fällt ein wenig aus dem Rahmen, denn es hat eine »interaktive Handlung« zur Folge: Auf dem Bildschirm des Clients soll ein Meldungsfenster geöffnet werden, welches der Benutzer des Clients mit OK schließen muss. Die Erzeugung dieses Ereignisses ist so einfach wie die der bereits besprochenen Änderungsereignisse und findet statt, wenn der Benutzer im TreeDesigner den Menüpunkt HILFE | KONTAKT MIT FERNSTEUERUNG... auswählt:

```
procedure TMainForm.KontaktmitFernsteuerung1Click(Sender: TObject);
var
  Nachricht: String;
  AutoObject: TTreeDesigner3;
begin
  Nachricht := '';
  AutoObject := TDAuto3.GetTD3AutoObject;
  if not Assigned(AutoObject) then
    ShowMessage('Diese TreeDesigner-Instanz wird nicht ferngesteuert!')
  else begin
    if AutoObject.Events = nil then
      ShowMessage('Fernsteuerung hört nicht zu.') else
    if InputQuery('Nachricht an Fernsteuerung:',
         'Bitte geben Sie den Text ein:', Nachricht) = True then
      AutoObject.Events.OnUserMessage(Nachricht);
  end;
end;
```

Die Besonderheit liegt in der Bearbeitung des Ereignisses, denn diese soll den Programmfluss des TreeDesigners nicht unnötig blockieren. Die Bearbeitungsmethode für *OnUserMessage* in der Client-Anwendung *TdCtrl3* soll also nicht warten, bis der Benutzer des Client-PCs das Meldungsfenster geschlossen hat, sondern so schnell wie möglich zum Aufrufer zurückkehren. Sie öffnet daher nicht selbst ein Meldungsfenster, sondern legt lediglich eine dynamische Kopie des Meldungstextes an, damit dieser später angezeigt werden kann. Dann bedient sie sich der in Kapitel 3.1.4 unter *Asyn-*

chrone Nachrichtensendungen beschriebenen Technik, sich selbst mit *PostMessage* eine Nachricht zu senden, die dann erst bearbeitet wird, wenn die aktuelle Ereignisbearbeitung (*OnUserMessage*) abgeschlossen ist:

```
const
  CM_REMOTE_MESSAGE = WM_APP + 101;

procedure TTDControllerForm3.TreeDesigner3UserMessage(Sender: TObject;
  var text: OleVariant);
var
  MehrText: String;
  DynText: PChar;
begin
  MehrText := 'Nachricht vom TreeDesigner: ' + Text;
  DynText := StrNew(PChar(MehrText));
  PostMessage(Handle, CM_REMOTE_MESSAGE, 0, Integer(DynText));
end;
```

Die Methode zur Bearbeitung der selbst definierten Nachricht kann dann das Meldungsfenster öffnen:

```
type
  TTDControllerForm3 = class(TTDControllerForm)
    // Deklaration der Methode im Formular des Automations-Controllers:
    procedure CMRemoteMessage(var Msg: TMessage);
                message CM_REMOTE_MESSAGE;
  ...

procedure TTDControllerForm3.CMRemoteMessage(var Msg: TMessage);
var
  DynText: PChar;
begin
  DynText := PChar(Msg.lParam); // Nachricht erkennen
  ShowMessage(DynText); // Nachricht anzeigen
  StrDispose(DynText); // dynamischen Speicher freigeben
end;
```

Natürlich sollte der Benutzer am Client-Rechner auf eine solche Nachricht auch antworten können. Hierzu wird allerdings kein Event benötigt, sondern die Schnittstelle *ITreeDesigner3* enthält die Methode *UserMessage*, die der Client wie jede andere Methode aufrufen kann. Die Implementierung dieser Methode im TreeDesigner entspricht der Bearbeitung des *OnUserMessage*-Events im Client-Programm: Der TreeDesigner schickt sich mit *PostMessage* selbst eine Nachricht und gibt die Kontrolle unmittelbar wieder an den Client zurück.

8.8 Web-Server-Anwendungen

Anwendungen, die auf einem Web-Server laufen, erzeugen meist nur HTML-Dokumente, die im Browser des Endbenutzers angezeigt werden, und besitzen keine eingebaute GUI-Schnittstelle, die man aus Delphis visuellen Komponenten zusammensetzen könnte. Trotzdem kommen bei der Entwicklung solcher Anwendungen mit Delphi viele Einrichtungen der IDE zum Einsatz, die schon vom Formularentwurf bekannt sind: die Komponentenpalette, der Objektinspektor, Property-Editoren, das Objekt-Hierarchie-Fenster, Verknüpfung von Ereignissen mit Programmcode per Mausklick. Anstelle des Formular-Designers tritt ein Entwurfsfenster für Webmodule, das von seiner Entwurfszeit-Funktion her dem Datenmodulfenster einer Datenbankanwendung ähnelt.

Dieses Kapitel beginnt mit den bisher vorherrschenden Anwendungen, die HTML-Dokumente an einen Browser zurückschicken, und endet mit einem Abschnitt zu einer neuen Form von Web-Anwendungen, den SOAP-Web-Services, die auf Endbenutzerseite nicht durch einen Browser, sondern durch eine beliebige Anwendung angesprochen werden können.

- In Abschnitt 8.8.1 geht es zunächst um die Grundlagen aller HTML (bzw. XML) produzierenden Server-Anwendungen. Zur Demonstration wird dabei ein einfaches CGI-Programm geschrieben, das schon mit der Personal Edition von Delphi läuft.
- Abschnitt 8.8.2 greift dann auf die weiter gehende Unterstützung der Professional-Version von Delphi zurück: Webmodule helfen bei der Verteilung unterschiedlicher Anfragen an verschiedene Ereignisbearbeitsungsmethoden und Seitenproduzenten wirken beim Aufbau des HTML-Dokuments mit.
- Abschnitt 8.8.3 befasst sich mit den nur in Delphi Enterprise angebotenen Web-Snap-Komponenten, die nur auf die richtige Weise zusammengeschaltet werden müssen, um eine vollautomatische Web-Datenbank-Anwendung mit Tabellen, Formularen und Editiermöglichkeiten zu bilden.
- Der letzte Abschnitt widmet sich den Web-Diensten, wobei eine in Abschnitt 8.8.2 implementierte Web-Server-Funktion nun als Web Service angeboten wird.

8.8.1 Interaktion zwischen Web-Server und Anwendung

Die offensichtlichtste Funktion eines Web-Servers besteht darin, einem irgendwo auf der Welt befindlichen Browser-Client HTML-Dokumente zu liefern. Technisch ist dies durch den Austausch von HTTP-Botschaften realisiert, wobei es sich im Normalfall um einfache Textdokumente handelt, die aus einem Header und einem Datenteil bestehen. Der Client sendet zunächst eine solche HTTP-Botschaft als Anfrage zum Ser-

ver, und der Server antwortet ebenfalls mit einer HTTP-Botschaft, die beispielsweise ein HTML-Dokument enthalten kann. Dieses HTML-Dokument könnte nun auch noch dynamische Elemente wie etwa JavaScript enthalten, die auf der Client-Seite vom Browser gehandhabt werden.

Web-Server-Anwendungen, wie Sie sie mit Delphi entwickelt können, kommen ins Spiel, wenn nicht lediglich statische HTML-Dateien von der Festplatte des Servers abgerufen werden, sondern der Server einen dynamischen Inhalt bereitstellen soll, der womöglich für jeden Client individuell zusammengestellt wird. Oder wenn der Server auf Verlangen des Clients besondere Aufgaben durchführen soll, wie z.B. eine EMail zu verschicken, einen Account zur Registrierung von Software zu eröffnen oder eine Datenbank zu aktualisieren.

Dazu wird im einfachsten Fall die Server-Anwendung auf der Festplatte des Servers gespeichert und der Server zur Ausführung dieser Anwendung konfiguriert. Zu dieser Konfiguration gehört auch, dass dem Server (hier ist die Server-Software wie Apache, Microsoft Information Server etc. gemeint) mitgeteilt wird, auf welche Anfragen hin die Anwendung gestartet werden soll. Im laufenden Betrieb des Servers wird dann die Anwendung nur auf bestimmte Anfragen hin gestartet. Mit der Rücksendung einer Antwort endet dann normalerweise die Laufzeit der Anwendung.

CGI-Schnittstelle

Um diese grundlegende Funktionsweise anschaulich zu demonstrieren, eignet sich hervorragend eine einfache Art von Server-Anwendungen: die CGI-Anwendungen, bei denen es sich im Prinzip um Konsolenanwendungen handeln, die auch direkt von der Befehlszeile des Servers gestartet werden könnten (also beispielsweise von der Eingabeaufforderung, wenn der Server unter Windows NT betrieben wird, was in diesem Buch angenommen werden soll). CGI steht für »Common Gateway Interface« und bestimmt die Art, wie der Informationsaustausch zwischen Server und Anwendung realisiert ist, insbesondere wie die Anwendung vom Server aufgerufen wird:

- Erhält der Server eine Anfrage, die für die in Frage stehende Anwendung bestimmt ist, so startet er diese Anwendung und übergibt ihr die gesamte HTTP-Anfrage über die Standardeingabe, wie sie jeder Konsolenanwendung zugeordnet ist.
- Die Anwendung liest nun die Informationen von der Standardeingabe (in Delphi gibt es hierfür die schon in Turbo Pascal für DOS verwendete Funktion *readln*). Als Alternative dazu können einige Informationen auch aus Umgebungsvariablen ausgelesen werden (hier finden sich auch manche Informationen, die nicht Teil der HTTP-Anfrage sind, wie etwa Informationen über den Server selbst in den Variablen SERVER_NAME, SERVER_SOFTWARE u.a.). Welche Umgebungsvariablen

der Server vor dem Aufruf der Anwendung setzen sollte, ist ebenfalls Bestandteil der CGI-Spezifikation, wobei kleinere Abweichungen davon in der Dokumentation des jeweiligen Servers beschrieben sind.

- Als Ergebnis produziert die Anwendung nun beispielsweise ein HTML-Dokument (jedoch sind auch beliebige andere Ausgabeformate wie XML oder binäre Formate wie GIF denkbar). Der Weg von der Server-Anwendung zum Web-Server führt dabei über die Standardausgabe, die Standard-Pascal-Funktion zum Beschreiben dieser Ausgabe ist *writeln*. Der Web-Server fängt diese Antwort ab und leitet sie via HTTP an den Client weiter.

CGI-Programme mit Delphi Personal

Konsolenanwendungen können Sie in allen Delphi-Ausgaben erzeugen, indem Sie eine neue Projektdatei ohne Einbindung von Formulardateien erzeugen und die Compileranweisung *$APPTYPE CONSOLE* angeben. Eine solche Projektdatei kann theoretisch mit jedem Texteditor erstellt werden, am schnellsten erhalten Sie ein solches Anwendungsgerüst aber, indem Sie in der IDE DATEI | NEU | WEITERE | NEU | KONSOLENANWENDUNG auswählen.

Das im Folgenden erläuterte Beispielprogramm *CGITestMitDelphi* soll dem Zweck dienen, den oben beschriebenen Datenaustausch mit dem Web-Server zu veranschaulichen. Es listet sowohl einige der Umgebungsvariablen als auch die kompletten über die Standardeingabe empfangenen Daten auf:

```
program CGITestWithDelphi;

{$APPTYPE CONSOLE}

uses
  SysUtils;

procedure WriteEnvVar(VarName: String);
begin
  writeln(VarName, ' = ', GetEnvironmentVariable(VarName)+'<BR>');
end;

const
  FileEnd = ^Z;
var
  s: String;
  LineCount: integer;
begin
  // Teil 1: Fester Header und Überschriften
  writeln('Content-type: text/html');
  writeln;
  writeln('<HTML>');
  writeln('<H1>Ausgabe der CGI-Beispiel-Anwendung</H1>');
```

```
  writeln('Listing einiger Environment-Variablen:<BR>');
  // Teil 2: Ausgabe der Umgebungsvariablen
  WriteEnvVar('REMOTE_ADDR');
  WriteEnvVar('REMOTE_HOST');
  WriteEnvVar('REMOTE_IDENT');
  WriteEnvVar('HTTP_REFERER');
  WriteEnvVar('PATH_INFO');
  WriteEnvVar('REQUEST_METHOD');
  WriteEnvVar('QUERY_STRING');
  writeln('<H1>Listing der Konsoleneingabe:</H1>');
  // Teil 3: "Verarbeitung" der HTTP-Anfragebotschaft
  LineCount := 0;
  repeat
    inc(LineCount);
    readln(s);
    if s <> FileEnd then
      writeln(s+'<BR>');
  until (s = '') or (LineCount > 100) or (s = FileEnd);
  // Teil 4: Abschluss des HTML-Dokuments
  writeln('</HTML>')
end.
```

Das CGI-Programm hat die Aufgabe, eine komplette HTTP-Antwort zu generieren, die an den Client gesendet werden kann. Zu einer solchen Antwort gehört auch ein HTTP-Header, der in diesem Beispiel lediglich aus der Zeile `Content-type: text/html` besteht. Um den Header von den Daten zu trennen, wird danach mit *writeln* eine leere Zeile eingefügt.

Die restlichen Ausgabebefehle des Listings erzeugen das HTML-Dokument mit Hilfe von Tags, die in jedem HTML-Grundkurs vorkommen dürften (<*H1*> für eine Überschrift und <*BR*> für einen Zeilenumbruch). Zunächst sollen die Inhalte einiger Umgebungsvariablen angezeigt werden. Einige davon geben Auskunft über die Herkunft der Anfrage:

- REMOTE_ADDR, REMOTE_HOST und REMOTE_IDENT enthalten Informationen über den Client, von dem die HTTP-Anfrage stammt: IP-Adresse des Clients, Name des Host-Rechners und Bezeichnung des Benutzers (falls verfügbar).
- HTTP_REFERER gibt an, von welcher Web-Adresse die Referenz stammt, über die der Client die aktuelle Seite angesteuert hat (dieser Wert ist ein leerer String, wenn der Benutzer die Adresse nicht über einen Link, sondern beispielsweise über ein Lesezeichen oder eine direkte Adressangabe im Browser angesteuert hat).

Die anderen vom Beispielprogramm ausgegebenen Variablen können dazu verwendet werden, der Web-Server-Anwendung genau mitzuteilen, was zu tun ist:

- PATH_INFO gibt eine eventuelle zusätzliche Pfadangabe an, die in der URL noch hinter dem Namen der Anwendung folgt. So hat beispielsweise bei der Adresse

`http://localhost/CGITestWithDelphi.exe/virtualdir1` *PATH_INFO* den Wert »virtualdir1« . Diese Pfadangabe muss nichts mit einem existierenden Verzeichnis auf dem Dateisystem irgendeines Datenträgers zusammenhängen, sondern kann auch lediglich dazu dienen, einen bestimmten Funktionsbereich der Server-Anwendung auszuwählen. In Kapitel 8.8.2 finden Sie ein Beispiel zur Verwendung dieser Pfadinformation.

- QUERY_STRING enthält eventuelle Parameter für das CGI-Programm, die über die URL mitgeliefert wurden. Dies ist, wie sich noch herausstellen wird, für *GET*-Anfragen wichtig.
- REQUEST_METHOD ermöglicht beispielsweise die Unterscheidung zwischen *GET*- und *POST*-Anfragen (hierauf kommen wir am Schluss dieses Abschnitts zu sprechen).

Installation unter Apache

Die Installation der Beispielprogramme wird in diesem Buch immer am Beispiel des Web-Servers Apache beschrieben. Delphi ist jedoch auch in der Lage, Anwendungen und DLLs für den Netscape-Server oder Microsofts Information Server zu erstellen, CGI-Anwendung sollten sogar auf beliebigen Web-Servern laufen, die Unterstützung für CGI aufweisen. Wenn Sie einen anderen Server als Apache verwenden, müssen Sie die Konfiguration entsprechend der jeweiligen Dokumentation anpassen.

Apache hat für unsere Zwecke den Vorteil, dass er sowohl für Linux als auch für Windows frei erhältlich ist (*www.apache.org*) und sich einfach installieren lässt. Bei der für den Test verwendeten Version 1.3.20 genügten bereits die folgenden Schritte, um die Anwendung auf dem lokalen PC zum Laufen zu bringen (der lokale PC beherbergt also gleichzeitig den Browser-Client als auch den Web-Server):

- den Server installieren (dies geschieht über ein komfortables Installationsskript, das den Microsoft Installer verwendet)
- Apache.exe starten (dies kann unter Windows über das START-Menü erfolgen)
- die Delphi-Anwendung in das Verzeichnis `Apache\cgi-bin` schreiben (dies geht am einfachsten, indem Sie dieses Verzeichnis in den Projektoptionen als *Ausgabeverzeichnis* angeben und die Anwendung in der Delphi-IDE neu kompilieren / erzeugen)
- eventuell benötigte HTML-Dateien in das Verzeichnis `Apache\htdoc` kopieren (in diesem Beispiel wird die Datei `test.html` kopiert, die zum Aufruf des CGI-Programms dient, siehe Abbildung 8.19)
- einen Internet-Browser starten und diesen zur Adresse `http://localhost/test.html` leiten (die Angabe von *localhost* führt zunächst dazu, dass die Anfrage des Brow-

sers an den Apache-Server des lokalen PCs geleitet wird, dieser sucht die gewünschten Dokumente standardmäßig in seinem *htdoc*-Verzeichnis, weshalb `test.html` im vorangegangenen Schritt in dieses Verzeichnis kopiert wurde).

Wenn Sie andere Verzeichnisse zur Ablage der CGI-Programme oder HTML-Dateien verwenden wollen, müssen Sie die Konfigurationsdatei von Apache ergänzen. In der getesteten Apache-Version lässt sich übrigens sowohl diese Konfigurationsdatei als auch die dabei natürlich erforderliche Dokumentation über das Windows-START-Menü aufrufen.

Auch für die noch folgenden Beispielanwendungen wird dieses Kapitel jeweils eine möglichst einfache Installationsart zur Nutzung auf dem lokalen Rechner beschreiben. Wenn die Anwendungen nicht nur auf dem lokalen PC, sondern über das Internet aufgerufen werden sollen, sind eventuell zusätzliche Konfigurations- und Sicherheitsmaßnahmen erforderlich, für die auf spezielle Literatur zur Einrichtung von Web-Servern verwiesen sei. Auf die Programmierung der Delphi-Anwendung hat dies jedoch keinen Einfluss.

Test der CGI-Anwendung

Oben wurde bereits beschrieben, wie Sie die Anwendung über die Adresse `http://localhost/test.html` im Web-Browser testen können. Hierzu ist allerdings die zusätzliche Datei `test.html` erforderlich. Ein Test der reinen Anwendung ist möglich, indem Sie statt dessen die Adresse `http://localhost/cgi-bin/CGITestMitDelphi.exe` angeben.

Abbildung 8.18: Ausgabe des Beispielprogramms bei direkter Adressangabe mit manuell eingegebenen zusätzlichen Parametern

Auch die Verarbeitung von Parametern können Sie schon testen, indem Sie sie hinter der Adresse angeben, beispielsweise `http://localhost/cgi-bin/CGITestMitDelphi.exe?Param1=5&Param2=10` (das Beispielprogramm akzeptiert ja beliebige Parameter und zeigt diese lediglich wie eine Art Echo auf der Ergebnisseite an). Abbildung 8.18 zeigt die Ausgabe der Anwendung für diesen Fall.

Normalerweise werden die Parameter jedoch nicht vom Endandwender in die URL geschrieben, sondern über eine Automatik in den im Browser verarbeiteten HTML-Dokumenten. Ein Beispiel dafür ist das schon erwähnte `test.html` (Abbildung 8.19).

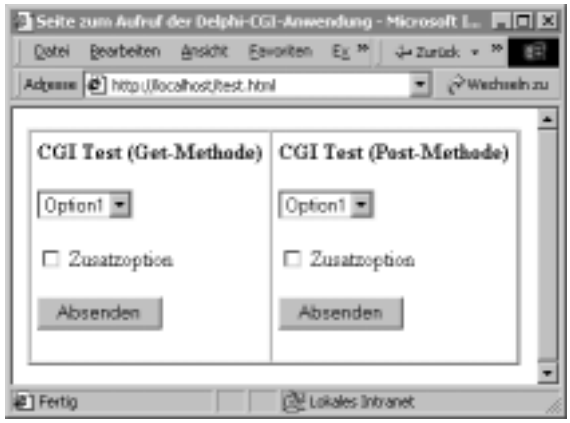


Abbildung 8.19: Eine HTML-Dokument, das über Formulare den Aufruf des Beispielprogramms mit Parametern gestattet

Zusammenstellen der Parameter über ein HTML-Formular

Das folgende Listing zeigt den Abschnitt aus `test.html`, der für die linke Tabellenzelle des in Abbildung 8.19 gezeigten Dokuments verantwortlich ist:

```
<FORM NAME="HtmlFormular1" ACTION="/cgi-bin/CgiTestWithDelphi.exe"
 METHOD=GET>
<H1>CGI Test (Get-Methode)</H1>
<SELECT id="FormularKombinationsfeld1"
    NAME="FormularKombinationsfeld1" >
    <OPTION VALUE="1" SELECTED>Option1</OPTION>
    <OPTION VALUE="2">Option2</OPTION>
    <OPTION VALUE="3">Option3</OPTION>
</SELECT><BR><BR>
<INPUT id="FormularKontrollkästchen1" TYPE="checkbox"
 NAME="Checkbox" VALUE="1" > Zusatzoption<BR><BR>
<INPUT TYPE="submit" NAME="SendButton" VALUE="Absenden"
 id="FormularSchaltfläche1" >
</FORM>
```

Während für Details auf die umfangreiche Literatur zu HTML verwiesen werden muss, sollen die entscheidenden Punkte kurz zusammengefasst werden:

- Den äußeren Rahmen des Abschnitts bildet das *FORM*-Tag, welches alle darin enthaltenen Kontrollelemente zu einem Formular zusammenfasst. Die Werte dieser Kontrollelemente werden vom Web-Browser als Parameter einer Anfrage an den Web-Server gesendet, wenn der Benutzer einen speziellen Schalter betätigt.
- Dieser Schalter wird im Listing durch die Angabe `TYPE="SUBMIT"` gekennzeichnet.
- Als weitere Kontrollelemente kommen im Formular ein *SELECT*-Feld (entspricht einer Kombobox und enthält im Beispiel drei Optionen) und ein *INPUT*-Feld, das über die Option `TYPE="CHECKBOX"` als Markierungsschalter gekennzeichnet wird.
- Nun fehlt noch eine Art »Link« zu der CGI-Anwendung, die bei Betätigung des Schalters aufgerufen werden soll. Dieser Link wird im *FORM*-Tag als *ACTION*-Parameter angegeben (`ACTION="/cgi-bin/CgiTestWithDelphi.exe«`). Um zwischen den beiden möglichen Anfragearten *GET* und *POST* zu unterscheiden, enthält das *FORM*-Tag außerdem noch ein Angabe `METHOD=...`, im obigen Beispiel wurde die *GET*-Methode gewählt.

Get und Post

Für die Web-Server-Anwendung sieht der Unterschied zwischen *Get* und *Post* wie folgt aus:

- Bei *Get* sind die Parameter in der Umgebungsvariable *QUERY_STRING* zu finden, die Variable *REQUEST_METHOD* hat den Wert »GET« und der Datenteil der HTTP-Anfrage ist leer.
- Bei *Post* enthält dagegen typischerweise die Variable *QUERY_STRING* einen leeren String. Um die Parameter zu kennen, muss die Anwendung die HTTP-Anfrage untersuchen (also die Daten, die in unserem Beispiel mit *readln* gelesen werden; dabei geht den eigentlichen Daten noch der HTTP-Header voran, dessen Ende durch eine Leerzeile zu erkennen ist).

Normalerweise bestimmt es der Entwickler der Server-Anwendung, ob die Kommunikation über *GET* oder über *POST* abläuft. Das Beispielprogramm erlaubt zur Demonstration beide Anfragetypen.

Ein Vorteil der *GET*-Methode liegt darin, dass der Endbenutzer solche Abfragen in den Lesezeichen des Browsers speichern kann, denn die Abfrage ist Teil der URL (z.B. der oben schon als Beispiel genannte Teil »`?Param1=5&Param2=10«` hinter der Adresse des CGI-Programms). Daten, die bei einer *POST*-Methode an den Server gesendet werden, werden dagegen nicht in den Browser-Lesezeichen gespeichert und müssen vom

Benutzer nach dem Aufruf des Lesezeichens neu zusammengestellt (z.B. durch Ausfüllen eines Formulars) und erneut an den Server gesendet werden (z.B. durch Anklicken eines Buttons).

Um das obige HTML-Formular von der *GET*- auf die *POST*-Methode umzustellen, muss nur das entsprechende Wort im *FORM*-Tag ersetzt werden. Der rechte Teil des in Abbildung 8.19 gezeigten Browser-Inhalts wird durch den gleichen HTML-Quelltext erzeugt wie der linke Teil, lediglich die ersten beiden Zeilen weisen geringe Unterschiede auf:

```
<FORM NAME="HtmlFormular2" ACTION="/cgi-bin/CgiTestWithDelphi.exe"  METHOD=POST>
<H1>CGI Test (Post-Methode)</H1>
```

Wenn Sie also den unteren *Absenden*-Schalter drücken, werden die Einstellungen in den Kontrollelementen (ebenfalls nur die aus der unteren Hälfte) zu einem Parameterstring zusammengefasst und vom Browser per *POST*-Anfrage an den Web-Server gesendet. Dieser startet die Delphi-Anwendung, welche daraufhin das in Abbildung 8.20 gezeigte Dokument generiert.

Abbildung 8.20: Ausgabe des Beispielprogramms mit Parametern bei Verwendung der Post-Methode

8.8.2 Web-Module und Seitenproduzenten

Das bisherige Beispielprogramm diente dem Zweck, die recht einfachen Anfrage-Antwort-Abläufe zu veranschaulichen, die auch komplexen Web-Anwendungen zu Grunde liegen, doch wenn wir die begonnene Beispielanwendung ohne zusätzliche

Hilfsmittel weiter entwickeln wollten, wären mit großer Wahrscheinlichkeit einige der folgenden Probleme zu lösen:

- Zur Abfrage der Parameter müssten Strings der Art `Param1=5&Param2=10` in ihre Bestandteile zerlegt werden.
- Wenn anspruchsvollere Ergebnis-Dokumente erzeugt werden sollen, wäre dies mit einer Folge von *writeln*-Anweisungen zwar theoretisch möglich, aber doch sehr umständlich. Es wäre sicher angenehmer, wenn bereits eine Schablone der HTML-Datei bereit liegen würde, die nur noch ausgefüllt zu werden bräuchte.
- Wenn eine Web-Anwendung viele verschiedene Funktionen anbietet, sollten diese auch in jeweils eigenen Methoden der Anwendung gekapselt werden. Die Anwendung müsste also zuerst die Anfrage-Parameter oder die zusätzliche Pfadangabe (siehe *PATH_INFO*-Variable in Kapitel 8.8.1) untersuchen, um dann die Methode aufrufen zu können, welche die eigentliche Arbeit erledigen soll.

All dies lässt sich natürlich mit den sprachlichen Mitteln der Personal-Edition von Delphi nicht allzu schwer bewerkstelligen, doch da die genannten Aufgaben in nahezu jeder Web-Anwendung anfallen, hat Borland dafür Bibliotheken entwickelt, die ab der Professional-Version von Delphi mitgeliefert werden. Ab jetzt werden wir uns also, anstatt das Rad neu zu erfinden, mit diesen Bibliotheken befassen und damit die Personal-Edition von Delphi verlassen.

Ziel dieses Abschnitts ist die Entwicklung einer Anwendung, die eine Seite mit festem Grundgerüst mit diversen dynamisch generierten Inhalten füllt, und zwar (siehe auch Abbildung 8.25 auf Seite 1206) mit der aktuellen Uhrzeit, mit einer einzeiligen Information über den nächsten Termin aus einer Datenbank, mit einer kompletten Datenbanktabelle und schließlich mit Vorgabewerten für die Eingabeelemente der Seite.

Erstellung einer Web-Broker-Anwendung

Die Bibliotheks-Funktionalität für Web-Server-Anwendungen firmiert in Delphi unter dem Namen »Web Broker« und wurde in Delphi 3 eingeführt. Für die oben genannten drei Punkte bietet Web Broker die folgenden Lösungen an:

- Die Parameterstrings aller Anfragen werden automatisch geparst und in einem Property (*TRequest.QueryFields*) gespeichert.
- Zum Erzeugen von HTML-Dokumenten aus Schablonen gibt es *TPageProducer*-Komponenten.
- Verschiedene Funktionen einer Web-Anwendung können automatisch über Ereignis-Handler aufgerufen werden, ein Web-Modul dient dabei als Ereignis-Verteiler (Dispatcher).

Für das Anlegen einer Web-Broker-Anwendung empfiehlt sich die Verwendung der IDE-Funktion DATEI | NEU | WEITERE... | NEU | WEB-SERVER-ANWENDUNG. Sie gelangen dadurch in den in Abbildung 8.21 gezeigten Dialog, in dem Sie aus fünf verschiedenen Arten von Anwendungen wählen können. Danach erhalten Sie ein neues Projekt, in das automatisch die erforderlichen Bibliotheken zur Unterstützung des gewählten Web-Server-Interfaces (IIS, Apache etc.) eingebunden wurden und das auch schon über das Herzstück einer Web-Broker-Anwendung verfügt: ein Web-Modul.

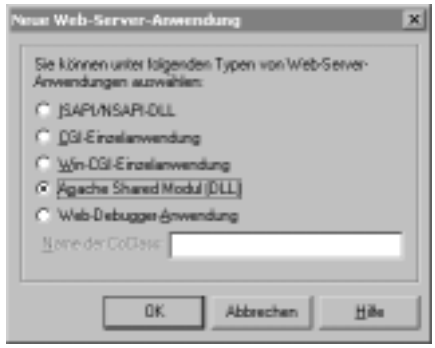


Abbildung 8.21: Die bei der Erzeugung des Web-Moduls aus dem Beispielprogramm gewählte Dialogeinstellung

Das Web-Modul tritt zur Entwurfszeit ähnlich wie ein Datenmodul durch ein Fenster in Erscheinung, welches zur Sammlung nicht-visueller Komponenten dient (Abbildung 8.22. Hier werden wir in Kürze *PageProducer*- und andere Komponenten unterbringen. Zunächst sei jedoch noch auf die Konsequenzen des Dialogs aus Abbildung 8.21 eingegangen.

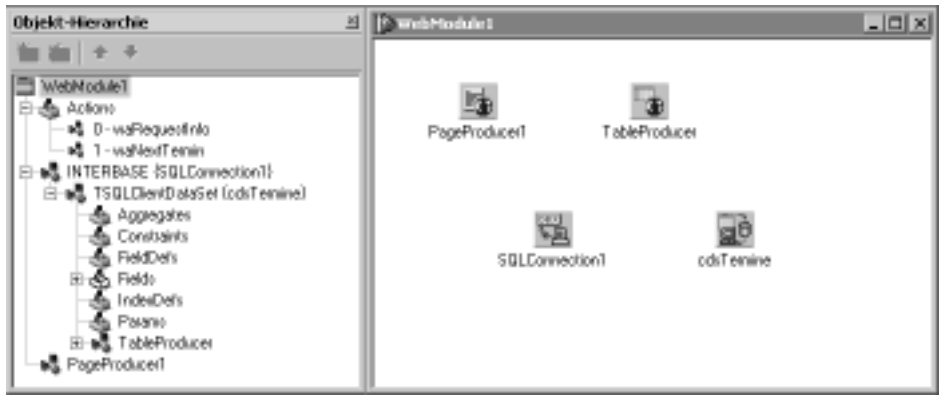


Abbildung 8.22: Das Web-Modul der fertigen Beispielanwendung

DLLs für Apache

Für das Beispielprogramm, das Sie auf der CD finden, wurde die Option *Apache Shared Modul (DLL)* gewählt. Beim Kompilieren erzeugt Delphi eine `.dll`-Datei, die von Apache allerdings nicht so ohne weiteres geladen wird, wie wir es von der CGI-Anwendung im `cgi-bin`-Verzeichnis kennen. Zur »Aktivierung« der DLL bedarf es einer Erweiterung der Apache-Konfigurationsdatei `Apache\conf\httpd.conf`, bei der auch ein Bezeichner angegeben werden muss, der in der Projektdatei der Web-Broker-Anwendung genannt wird. In der Projektdatei des Beispielprogramms finden Sie die folgende Angabe (der automatisch von Delphi erzeugte Name *Project1_Module* wurde manuell in *delphiwebxp_module* geändert):

```
library DelphiWebXP;

...

exports
  apache_module name 'delphiwebxp_module';
```

Wenn Sie die auf diese Weise kompilierte DLL in das Verzeichnis `Apache\Modules` kopieren, können Sie die DLL mit den folgenden Zeilen in `httpd.conf` bei Apache registrieren:

```
LoadModule delphiwebxp_module Modules/DelphiWebXP.dll
<Location /delphiapp>
  SetHandler delphiwebxp-handler
</Location>
```

Dabei ist *delphiwebxp-handler* ein automatisch bereitgestellter Name, er wird von der Web-Broker-Bibliothek grundsätzlich vom Namen des Projekts abgeleitet. Wichtig ist die Kleinschreibung der Bezeichner in der Konfigurationsdatei, auch wenn im Projektnamen unter Delphi Großbuchstaben vorkommen sollten. Der hier verwendete Projektname steht übrigens für *DelphiWeb eXPeriment*.

Doch nun zurück zur Apache-Konfigurationsdatei: Der unter *Location* angegebene Name */delphiapp* gibt quasi einen Alias-Namen für die DLL an, unter dem man ihre Funktionalität in einer URL ansprechen kann. In diesem Beispiel wird die DLL also unter der URL

```
http://localhost/delphiapp/*
```

erreichbar sein, wobei anstelle von »*« beliebige weitere Pfadangaben oder Parameter folgen können, die dann an die DLL weitergeleitet werden.

Wie schon in Kapitel 8.8.1 angekündigt, kann dieses Buch nicht auf alle Arten von Web-Servern eingehen und beschränkt sich daher auf den frei zugänglichen Apache. Mit Hilfe der Dokumentation des Microsoft Information Servers und den mit Delphi mitgelieferten Beispielen (z.B. unter `Demos\Internet\WebServ\IIS`) sollte sich das Beispiel

jedoch leicht auf den IIS übertragen lassen. Hierfür muss zunächst mit DATEI | NEU | WEITERE... | NEU | WEB-SERVER-ANWENDUNG ein neues Projekt erstellt werden, wobei diesmal die Option *ISAPI/NSAPI* gewählt werden muss. Dann können Sie das automatisch erzeugte leere Webmodul durch das bestehende Webmodul ersetzen (PROJEKT | AUS DEM PROJEKT ENTFERNEN und PROJEKT | DEM PROJEKT HINZUFÜGEN).

Hinweis: Auf die letzte im Dialog von Abbildung 8.21 genannte Option, der *Web-Debugger-Anwendung*, wird am Ende des Kapitels eingegangen.

CGIs vs. dynamisch geladene Module

Auch wenn die Web-Server-Anwendung in Form einer DLL vorliegt, wird sie vom Server nur geladen, wenn eine Anfrage für sie vorliegt, d.h. wenn der oben als *Location* definierte Pfad in der URL genannt wurde. Der Datenaustausch kann jedoch nicht mehr über Standardein- und Ausgabe erfolgen, sondern findet nun über Funktionsparameter statt. Bezüglich des Namens dieser Funktionen und ihrer Aufrufformate verfügen verschiedene Web-Server über verschiedene Schnittstellen, doch werden diese Unterschiede von Delphis Web Broker ausgeglichen. Je nachdem, welchen Server-Typ Sie bei der Erstellung der Anwendung wählen, sorgt Delphi automatisch dafür, dass die Anwendung die entsprechende Schnittstelle bereitstellt. Die Programmierung der Anwendung im Web-Modul ändert sich dadurch nicht.

Hinweis: Wie aus Abbildung 8.21 hervorgeht, können Sie mit Web Broker auch einfache CGI-Anwendungen erstellen. Der Zugriff auf Standardein- und -ausgabe, der in Kapitel 8.8.1 noch »von Hand« erledigt wurde, wird in diesem Fall komplett von Web Broker übernommen. Das im Folgenden entwickelte Webmodul könnte mit all seinen Aktionen, Ereignisbehandlungsmethoden und Seitenproduzenten also ohne Änderung auch in einer CGI-Anwendungen verwendet werden.

Bearbeiten von Anfragen über Aktionen und Ereignisse

Ein Webmodul bietet nicht nur einen Container für andere visuelle Komponenten, sondern stellt auch noch eine zentrale Funktion einer Web-Broker-Anwendung bereit: Das Verteilen (Dispatchen) von verschiedenen Anfragen auf verschiedene Methoden. Als Unterscheidungsmerkmal der Anfragen dient die Pfadangabe in der URL. Das Beispielprogramm soll zwei Dienste zur Verfügung stellen:

- unter `localhost/delphiapp/request_info` soll wieder eine Seite mit Informationen über die empfangene Anfrage ausgegeben werden, ähnlich wie beim CGI-Programm aus dem vorigen Kapitel (Abbildung 8.24)

- unter `localhost/delphiapp/termin_info` sollen Datenbankinformationen angezeigt und dabei die Termindatenbank aus Kapitel 7.5 wieder verwendet werden (Abbildung 8.25).

Bei den angegebenen URLs ist natürlich *localhost* in der Praxis durch die Adresse des WWW-Servers zu ersetzen, auf dem die Anwendung läuft; *delphiapp* ist die Pfadangabe, unter der wir die Anwendung in der Apache-Konfigurationsdatei registriert haben; *requst_info* bzw. *termin_info* sind die zusätzlichen Pfadangaben, die an die Anwendung weitergegeben und vom Webmodul zum Verteilen der Anfrage verwendet werden.

Bevor ein Webmodul die Anfragen verteilen kann, müssen Sie die von Ihrer Anwendung erwarteten Anfragen als *Aktionen* definieren. Über das *Actions*-Property des Webmoduls (oder einfach über einen Doppelklick in das Entwurfszeit-Fenster des Moduls) erhalten Sie den in Abbildung 8.23 gezeigten Aktionseditor, in dem Sie auf die übliche Weise über Toolbar oder lokales Menü neue Aktionen hinzufügen können. Die einzelnen Merkmale der Aktionen stellen Sie im Objektinspektor ein, der Aktionseditor zeigt die wichtigsten noch einmal tabellarisch an:

- *VerzInfo* (Property *PathInfo*) enthält die schon beschriebenen Pfadangaben, über die zwischen den Aktionen unterschieden wird.
- *Aktiviert* (Property *Enabled*) gestattet es, eine Aktionsbehandlung zeitweise aus der Verteilung auszuklammern.
- *Standard* (Property *Standard*) kann immer nur bei einer Aktion gesetzt werden und führt dazu, dass die Aktion für alle Anfragen ausgeführt wird, für die kein anderer Handler gefunden wurde (die Standard-Aktion wird übrigens auch dann ausgeführt, wenn *Enabled=False*).
- *Produzent* (Property *Producer*) weist auf einen der weiter unten erläuterten Seitenproduzenten (es wurde in Delphi 5 für den automatischen Aufruf der schon seit Delphi 3 verfügbaren Seitenproduzenten eingeführt).

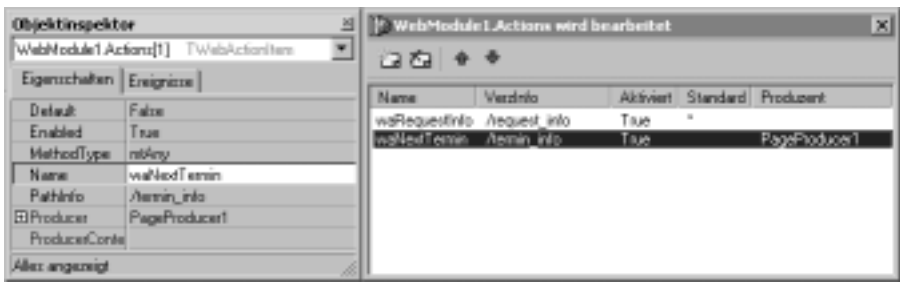


Abbildung 8.23: Die Web-Aktionen des Beispielprogramms in Aktionseditor und Objektinspektor

Bei der Verteilung der Anfragen an Aktionen gibt es zusätzliche Möglichkeiten, was die Behandlung einer Anfrage durch mehrere Aktions-Handler betrifft, für die hier auf die Online-Hilfe verwiesen sei. Außerdem können Sie jede Aktion über das Property *MethodType* auf bestimmte Arten von HTTP-Anfragen beschränken (neben den schon erläuterten *GET*- und *POST*-Anfragen gibt es auch noch *HEAD* und *PUT*). Für das Beispiel genügt es, wenn zwei Aktionen erstellt und die Properties eingestellt werden, wie in der Abbildung gezeigt.

Hinweis: Sie können sich ein Webmodul auch als Datenmodul vorstellen, das um die beschriebene Dispatch-Funktion erweitert wurde. Tatsächlich können Sie in einer Web-Broker-Anwendung anstelle des Webmoduls auch ein Datenmodul verwenden, in dem Sie die Dispatch-Funktion durch eine *TWebDispatcher*-Komponente nachrüsten (zu finden in der Komponentenpalette auf der Seite *Internet*).

Neben den Properties verfügt jede Web-Aktion auch noch über das Ereignis *OnAction*, mit dem Sie nun die Methode verknüpfen können, die die Anfrage bearbeitet. Wir beginnen mit der ersten Aktion des Beispielprogramms, in der die IP-Adresse des Clients, die eventuell in der URL (oder im Inhaltsteil einer POST-Anfrage) angegebenen Parameter sowie die für die Verteilung der Anfragen auf Web-Aktionen entscheidende Pfadangabe angegeben werden.

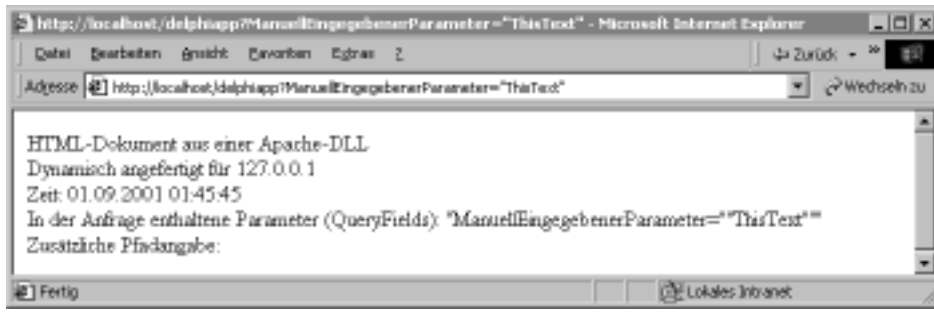


Abbildung 8.24: Die Standardseite der Beispielanwendung wird in einem Ereignishandler manuell aufgebaut.

Wir wenden uns nun der ersten, in Abbildung 8.24 gezeigten Seite des Beispielprogramms zu. In Kapitel 8.8.1 wurde eine ähnliche Web-Seite erzeugt, indem Umgebungsvariablen abgefragt, die Standardeingabe gelesen und schließlich der HTML-Antwort-Code mit *writeln* auf die Standardausgabe geschrieben wurde. Dies ist in einer Web-Broker-Anwendung nicht nötig, da hier die komplette HTTP-Anfrage in einem *TWebRequest*-Objekt gekapselt wird (ein wichtiger Vorteil ist dabei, dass *TWe-*

bRequest völlig unabhängig von der Schnittstelle des Web-Servers ist, ebenso wie von der Tatsache, ob es sich um eine CGI-Anwendung oder ein dynamisch geladenes Server-Modul handelt).

Für die Antwort steht entsprechend ein Objekt der Klasse *TWebResponse* zur Verfügung. Die Ereignismethode einer Web-Aktion erhält jeweils ein Anfrage- und ein Antwort-Objekt in den Parametern *Request* bzw. *Response*. Ihre Aufgabe bei der Entwicklung einer Web-Anwendung besteht normalerweise darin, die Antwort im *Response*-Objekt abzulegen.

Die folgende Beispielmethode macht intensiven Gebrauch von *Request* und *Response*; sie ist mit dem *OnAction*-Ereignis der ersten in Abbildung 8.23 gezeigten Aktion des Beispielprogramms verknüpft, wird also über die Adresse `localhost/delphiapp/request_info` aufgerufen. Die Verwendung der einzelnen Properties dieser Objekte im Beispielprogramm ist selbsterklärend:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := 'HTML-Dokument aus einer Apache-DLL<BR>'+
    'Dynamisch angefertigt für '+Request.RemoteAddr+'<BR>'+
    'Zeit: '+DateTimeToStr(Now) + '<BR>';
  if Request.QueryFields.Count > 0 then
     Response.Content := Response.Content +
       'In der Anfrage enthaltene Parameter (QueryFields): '
       + Request.QueryFields.CommaText + '<BR>'
  else Response.Content := Response.Content
         + 'Keine Anfrageparameter vorhanden.<BR>';
  if Request.ContentFields.Count > 0 then
       Response.Content := Response.Content +
          'Felder im "Content"-Teil der Anfrage (ContentFields): '
          + Request.ContentFields.CommaText + '<BR>';
  Response.Content := Response.Content +
     'Zusätzliche Pfadangabe: '+Request.PathInfo + '<BR>';
  Handled := True; // (nicht unbedingt nötig, da True bereits Vorgabewert)
end;
```

Wenn Sie sich für Details zu den verwendeten Properties interessieren, finden Sie diese in der Online-Hilfe. Hier sei nur noch einmal hervorgehoben, dass die einzelnen Parameter über *Request.QueryFields* viel einfacher abfragbar sind als in einem langen String. (Dieser Vorteil kommt im Beispielprogramm nicht so deutlich zum Ausdruck, da hier alle einzeln gespeicherten Parameter über *QueryFields.CommaText* wieder zu einem String aneinanger gereiht werden. An den Kommata in diesem String können Sie jedoch erkennen, an welchen Stellen Web Broker den Original-Parameter-String aus der URL getrennt hat.)

Hinweise: Neben den *OnAction*-Ereignissen der Web-Aktionen bieten sich zur Anfragebearbeitung auch noch die Ereignisse *BeforeDispatch* und *AfterDispatch* des Webmoduls an. Mit ihnen können Sie Methoden verknüpfen, die bei *allen* Anfragen durchgeführt werden sollen (nicht zu verwechseln mit der Standardaktion, die nur dann ausgeführt wird, wenn keine andere passende Aktion gefunden wurde).

Nicht immer ist eine Anfragebearbeitung so reibungslos möglich wie in diesem Beispiel. Falls Sie ein Fehlermeldung zurückgeben wollen, können Sie dies über die *TWebResponse*-Properties *StatusCode* und *ReasonString* tun. Der HTTP-Standard definiert bereits einige häufig verwendete Fehlercodes wie etwa den Code 404, der den meisten Internet-Nutzern von der Eingabe einer falschen Adresse bekannt sein dürfte, oder den Statuscode 200, der für »OK« steht und von der Delphi-Anwendung standardmäßig zurückgeliefert wird, falls weder Sie *StatusCode* auf einen anderen Wert setzen noch ein anderer Fehler in den Web-Broker-Komponenten auftritt.

Verwenden von Seitengeneratoren und transparenten Tags

Die zuletzt gezeigte Methode verwendet zwar kein *writeln*, baut aber die einzelnen Zeilen der HTML-Antwort kaum weniger mühselig mit einzelnen String-Operationen auf. Für die zweite Aktion des Programms fällt auch dies weg. Sie verwendet als Schablone den im Folgenden abgedruckten HTML-Quelltext:

```
<HTML>
<HEAD>
<TITLE>Delphi Web eXPeriment (Apache DLL)</TITLE>
</HEAD>

<BODY>
<H1>Testseite für die Webserver-DLL</H1>
<H2>Abschnitt 1</H2>
<P><FONT SIZE="-1">(Generiert mit Hilfe von transparenten Tags)</FONT></P>
<P>Aktuelle Zeit: <#AktuelleZeit></P>
<P>Der nächste Termin in der Termindatenbank ist: <B><#NaechsterTermin></B></P>

<H2>Abschnitt 2</H2>
<P>Die folgende Ansicht der Termintabelle enthält alle Datensätze, ist aber beschr
   änkt auf eine Teilmenge der Spalten. Ohne Lookup-
   Felder für Adressnummer und Aktionsnummer und ohne Editiermöglichkeit.</P>
<P><#TerminTabelle></P>
</BODY>
</HTML>
```

Diese Datei enthält drei *transparente Tags*, die durch das »#« als erstes Zeichen zu erkennen sind: *<#AktuelleZeit>*, *<#NaechsterTermin>* und *<#TerminTabelle>*. Für einen normalen HTML-Browser sind diese Tags unsichtbar, jedoch ist es auch nicht Zweck dieser

Tags, bis an den Browser des Clients weitergeleitet zu werden, sondern sie stellen die Schablonenfelder dar, die noch in der Delphi-Web-Anwendung durch konkrete Informationen ersetzt werden sollen. Das Ergebnis dieser Ersetzung ist in Abbildung 8.25 zu sehen.

Abbildung 8.25: Die Datenbankseite wird durch einen PageProducer unter Zuhilfenahme eines DataSetTableProducers erzeugt.

Um eine solche Schablone in Funktion zu setzen, sind folgende Schritte auszuführen:

- Definieren Sie die Schablone, beispielsweise in einem HTML-Editor, der die Eingabe von transparenten Tags unterstützt.
- Fügen Sie einen Seitenproduzenten in das Web-Modul der Delphi-Anwendung ein; im Beispielprogramm wird die Klasse *TPageProducer* verwendet (Abbildung 8.22, Komponente *PageProducer1*; wie alle Web-Broker-Komponenten zu finden auf der Palettenseite *Internet*).
- Laden Sie die Schablone in den Seitenproduzenten, indem Sie den gesamten HTML-Text in dessen Property *HTMLDoc* einfügen. Auf diese Weise wird der HTML-Code fest in der Delphi-Anwendung verankert. Alternativ dazu können Sie auch auf eine externe HTML-Datei verweisen, indem Sie das Property *HTMLFile* setzen.

- Verknüpfen Sie den Seitenproduzenten mit einer Aktion, indem Sie das *Producer*-Property der Aktion auf den Seitenproduzenten setzen (siehe die Aktion *waNextTermin* in Abbildung 8.23). Wenn eine solche Verknüpfung besteht, ist es nicht mehr erforderlich, das *OnAction*-Ereignis der Aktion zu bearbeiten, sondern der Seitenproduzent wird automatisch aufgerufen und sein Ergebnis wird als Ergebnis der Anfrage (*Response.Content*) verwendet.
- Beim Ersetzen der transparenten Tags durch dynamisch bereitgestellte Informationen benötigt der Seitenproduzent Ihre Hilfe. Er stellt dafür das Ereignis *OnHTMLTag* bereit, mit dem Sie eine Methode verknüpfen, in dem die transparenten Tags durch die passende Information ersetzt werden.

Diese Methode sieht im Beispielprogramm wie folgt aus:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'AktuelleZeit' then
    ReplaceText := DateTimeToStr(Now)
  else if TagString = 'NaechsterTermin' then
    ReplaceText := SharedDBCode.GetNextDate(
       cdsTermine,
       Request.QueryFields.Values['BaseDate'],
       Request.QueryFields.Values['NoSort'] = '1')
  else if TagString = 'TerminTabelle' then
    ReplaceText := TableProducer.Content;
end;
```

Das *OnHTMLTag*-Ereignis tritt einmal für jedes transparente Tag auf, somit wird die obige Methode für die oben gezeigte HTML-Schablone pro Anfrage insgesamt dreimal aufgerufen, wobei jedes Mal ein anderes Tag vorliegt. Die Bereitstellung des Ersatztextes könnte kaum einfacher sein: Sie setzen lediglich die Variable *ReplaceText*, die *PageProducer*-Komponente kümmert sich darum, dass dieser richtig in die Schablone eingesetzt wird.

Während im Beispiel das Tag für die aktuelle Uhrzeit direkt durch die Standardfunktionen *DateTimeToStr(Now)* ersetzt werden kann, dient zur Abfrage des nächsten Termins in der Termindatenbank die Funktion *GetNextDate*, die in einer eigenen Unit definiert wurde (der pragmatischste Grund dafür ist, dass die Funktion in Kapitel 8.8.4 noch in einem ganz anderen Projekt verwendet werden soll). Die Funktion sucht in der übergebenen *ClientDataSet*-Datenmenge nach dem nächsten Termin und gibt diesen als String zurück, damit er leicht in das transparente Tag eingesetzt werden kann. Zur Implementierung dieser Funktion finden Sie weiter unten einen eigenen Abschnitt.

Zuerst soll es um das dritte transparente Tag gehen, das gleich durch eine ganze Tabelle ersetzt wird, für die wir nun zu einer weiteren Web-Broker-Komponente kommen werden.

Der Web-Broker-Generator für eine Datenbanktabelle

Die Komponente *TDataSetTableProducer* (siehe Komponente *TableProducer* in Abbildung 8.22) erzeugt HTML-Code für eine Tabelle, in der der Inhalt einer beliebigen *TDataSet*-Komponente angezeigt wird (es werden *alle* Datensätze der Datenmenge angezeigt, in der Praxis wird es sich bei dieser Datenmenge meistens um das Ergebnis einer Abfrage oder um eine gefilterte Datenmenge handeln; im Beispielprogramm werden dagegen zunächst ungefiltert alle Datensätze angezeigt, die in der Datenbank enthalten sind).

Im Beispielprogramm wurde das Property *TableProducer.DataSet* auf die Komponente *cdsTermine* gesetzt, die schon in Kapitel 7.5.2 als Menge aller Termine der Termindatenbank verwendet wurde. Dazu wurde *cdsTermine* aus dem Terminverwaltungs-Projekt in das Webmodul kopiert (siehe Abbildung 8.22), ebenso wie die für die DBExpress-Anbindung erforderliche Komponente *SQLConnection1*.

Nachdem Sie über das *DataSet*-Property die Datenbankverbindung hergestellt haben, können Sie in den anderen Properties einige Beschriftungen (*Caption, Header, Footer*) und Attribute (*CaptionAlignment*, *MaxRows*, *RowAttributes*) angeben, die weitgehend selbsterklärend sind.

Der zentrale Ort, an dem Sie die Tabelle schon zur Entwurfszeit gestalten können, ist jedoch der in Abbildung 8.26 gezeigte Editor, den Sie über das Property *Columns* des TableProducers oder über das lokale Menü dieser Komponente (Punkt *Antwort-Editor*) aufrufen können. Hier können Sie festlegen, welche Spalten der Datenmenge in der Tabelle angezeigt werden und mit welcher Ausrichtung und welchen Farben diese versehen werden sollen. Auch weitere allgemeine Eigenschaften der Tabelle, etwa die Rahmen betreffend, können hier eingestellt werden.

Abbildung 8.26: Zur Entwurfszeit wird ein TDataSetTableProducer in einem speziellen Editor mit Vorschaufunktion editiert.

Im Beispielprogramm wurden Adress- und Aktionsnummer zur Demonstration rechtsbündig dargestellt (die Ausrichtung lässt sich für Tabellenüberschrift und den restlichen Spalteninhalt getrennt festlegen) und zwei verschiedene Spaltenfarben gewählt. Der Text für die Beschriftung in der Titelzeile der Tabelle wird übrigens nicht im Spalteneditor festgelegt, sondern aus dem Property *DisplayName* der Feldkomponenten übernommen, falls solche in der Datenmenge definiert wurden (zur Definition persistenter Felder siehe Kapitel 7.3.2).

Diese externe Definition der Spaltenüberschriften hat einen großen Vorteil: Wenn die gleiche Datenmenge auch noch von anderen Tabellenproduzenten oder in einer Windows-GUI-Anwendung, beispielsweise in einem *TDBGrid*, angezeigt werden soll, müssen die Spaltenüberschriften nicht jedes Mal neu definiert werden, da auch *TDBGrid* und andere Tabellenkomponenten sich nach dem *DisplayName*-Property der Feldkomponenten richten. In Kapitel 8.8.3 beispielsweise wirken sich die *DisplayName*-Properties der Felder von *cdsTermine* auch auf die Beschriftung in einem WebSnap-Seitengenerator aus (wobei WebSnap sogar noch die Möglichkeit bietet, den *DisplayName*-Text zu überschreiben).

Hinweis: Wenn Ihnen die vordefinierten Gestaltungsoptionen der Tabelle nicht ausreichen, können Sie im Property *Custom* einer jeden Tabellenspalte weitere HTML-Tags bzw. -Optionen hinzufügen.

Die Aktivierung des Tabellengenerators zur Laufzeit läuft so ab wie bei den Seitengeneratoren: Sie können einen Tabellengenerator z.B. dem *Producer*-Property einer Webmodul-Aktion zuweisen, dann besteht der Seiteninhalt, der durch diese Aktion abgerufen werden kann, nur aus der vom Generator erzeugten Tabelle. Im Beispielprogramm ist die Tabelle jedoch nur ein Teil des HTML-Ergebnisdokuments. Die bereits gezeigte Methode *PageProducer1HTMLTag* ruft die vom Generator erzeugte Tabelle über das Property *TableProducer.Content* ab und fügt ihn in das HTML-Dokument ein.

Weitere Generatoren für Datenbank-Informationen

Der im Beispiel verwendete Tabellengenerator ist nur einer von vier HTML-Generatoren, die Sie in der Komponentenpalette unter der Rubrik *Internet* finden. Die anderen drei sind ebenfalls sehr leistungsfähig und nützlich und ebenfalls sehr einfach zu verwenden. Mit der folgenden Kurzvorstellung soll das Thema der Generatoren fürs Erste abgeschlossen werden:

TDataSetPageProducer arbeitet wie *TPageProducer* mit einer HTML-Schablone, deren auszufüllende Felder durch transparente Tags gekennzeichnet sind. Jedoch brauchen Sie bei *TDataSetPageProducer* kein *OnHTMLTag*-Ereignis zu bearbeiten, da sich die Komponente die in die Schablone einzusetzen-

den Werte aus einer Datenmenge holt. Der Name des transparenten Tags gibt dabei das Feld der Datenmenge an, das von der Komponente abgefragt werden soll (z.B. das Feld *Stueckzahl* für das Tag <*#Stueckzahl*>). Ihre Aufgabe ist es also lediglich, in der Datenmenge vorher den gewünschten Datensatz zu selektieren, dessen Daten in das HTML-Dokument »eingeblendet« werden sollen.

TQueryTableProducer erzeugt wie *TDataSetTableProducer* eine Tabelle, die alle Datensätze einer Datenmenge auflistet. Im Unterschied zur im Beispielprogramm verwendeten Variante ist die *Query*-Version jedoch in der Lage, automatisch eine Datenbank-Abfrage durchzuführen. Dabei werden die Parameter der HTTP-Anfrage in die Parameter einer *TQuery*-Komponente übertragen und die *TQuery*-Anfrage ausgeführt. Aus dem Ergebnis der Anfrage wird dann wie bei *TDataSetTableProducer* eine Tabelle aufgebaut.

TSQLQueryTableProducer funktioniert wie *TQueryTableProducer*, arbeitet jedoch statt mit *TQuery* mit der in Delphi 6 neu eingeführten DBExpress-Komponente *TSQLQuery* zusammen.

Die Datenbank-Funktion des Beispielprogramms

Oben wurde bereits gezeigt, wie die Funktion *GetNextDate* zum Ausfüllen der HTML-Schablonen verwendet wird, an dieser Stelle soll es um die Implementierung der Funktion gehen, wobei die Funktion neben der als »Suchgebiet« zu verwendenden Datenmenge noch um zwei weitere Parameter erweitert wird:

```
function GetNextDate(DataSet: TSQLClientDataSet; BaseDate: String;
                     NoSort: Boolean) : String;
var
  SearchedDate: TDateTime;
  OldPosition: TBookmark;
  OldIndexFields: String;
begin
  if BaseDate = '' then SearchedDate := Now
  else try
    SearchedDate := StrToDateTime(BaseDate);
  except
    on EConvertError do begin
      Result := 'Fehlerhafter Parameter: ungültige Datumsangabe';
      exit;
    end;
  end;
  OldPosition := DataSet.GetBookmark;
  OldIndexFields := DataSet.IndexFieldNames;
  DataSet.IndexFieldNames := 'Datum';
  DataSet.SetKey;
  DataSet.FieldByName('Datum').Value := SearchedDate;
  DataSet.GotoNearest;
```

```
    if DataSet.FieldByName('Datum').Value < SearchedDate then
      DataSet.Next;
    if DataSet.FieldByName('Datum').Value < SearchedDate then
      Result := 'Es konnte kein Termin nach dem angegebenen '+
                'Datum gefunden werden.'
    else Result := DataSet.FieldByName('Datum').AsString + ': '
                   + DataSet.FieldByName('Beschreibung').AsString;
    if NoSort = True then begin
      // Durch das Suchen wurden Sortierung und Cursor in der
      // Datenmenge verändert. Zur Anzeige der vollständigen Tabelle
      // durch den DataSetTableProducer die Datenmenge zurücksetzen:
      DataSet.IndexFieldNames := OldIndexFields;
      DataSet.CancelRange; // durch GotoNearest vorgenommene
                           // Filterung abschalten
      DataSet.GotoBookmark(OldPosition);
    end;
    DataSet.FreeBookmark(OldPosition);
  end;
```

Die Funktionsweise der Terminsuche soll an dieser Stelle nur kurz erläutert werden. Das Gespann aus *SetKey* und *GotoNearest* wurde bereits in Kapitel 7.4.3 vorgestellt; *GotoNearest* geht zu dem Datensatz, der in den eingestellten Suchfeldern (*IndexFieldNames*) den Suchwerten am nächsten kommt. Da *GetNextDate* jedoch auf jeden Fall einen Termin zurückliefern soll, der *nach* dem angegebenen Termin stattfindet, muss ggf. mit *Next* noch der nächste Datensatz aufgerufen werden.

Am Schluss berücksichtigt *GetNextDate* noch den Parameter *NoSort*, mit dem es folgende Bewandnis hat: Eine Utility-Funktion, die nur einen String zurückliefern soll wie diese hier, sollte normalerweise keine weiteren Nebeneffekte haben. Im vorliegenden Fall sollte *GetNextDate* nur das Datum suchen, aber nicht das äußere Erscheinungsbild der durchsuchten Datenmenge verändern. Durch das Setzen von *IndexFieldNames* wird jedoch die Sortierung der Datensätze geändert und durch *GotoNearest* wird die Datenmenge gefiltert, so dass danach nur noch die Termine »sichtbar« sind, die nach dem gewählten Datum stattfinden. Diese Änderungen würden sich auf die Darstellung der Termintabelle durch den *DataSetTableProducer* auswirken. Im Beispielprogramm erweist sich diese Darstellung sogar als vorteilhaft (siehe Abbildung 8.25), daher wurde dieser »unsaubere« Nebeneffekt von *GetNextDate* zu einem »Feature« umdefiniert. Jedoch können Sie den Nebeneffekt abschalten, indem Sie den Parameter *NoSort* auf *True* setzen. *GetNextDate* macht dann am Schluss die bewirkten Effekte rückgängig, indem sie die *IndexFieldNames* zurücksetzt und die per Bookmark gespeicherte aktuelle Position in der Datenmenge wiederherstellt.

Erweiterung der Beispielanwendung mit einer Formulareingabe

Zum Abschluss des Beispielprogramms soll das HTML-Dokument, das als Schnittstelle zum Benutzer dient, durch eine Eingabemöglichkeit erweitert werden (Abbil-

dung 8.27): In einem Textfeld kann der Benutzer ein Datum angeben, nach dem ein Termin gesucht wird (lässt er diese Angabe weg, wird weiterhin das aktuelle Datum als Basis für die Terminsuche verwendet), über die darunter befindliche Checkbox kann er den im letzten Abschnitt beschriebenen *NoSort*-Parameter setzen. Beide Eingaben können mit dem Schalter *Erneuern* an die Delphi-Anwendung gesendet werden.

Abbildung 8.27: Die erweiterte Datenbankseite enthält ein HTML-Formular zur Einstellung der Parameter der HTTP-Anfrage.

Die zuletzt gedruckte HTML-Datei wird dafür um das nachfolgend abgedruckte Fragment ergänzt. Es enthält zwei neue transparente Tags *#BaseDateVorgabeText* und *#NoSortVorgabeWert*, die von der Delphi-Anwendung durch einen Vorgabetext für das Editierfeld und einen Vorgabewert für die Checkbox ersetzt werden:

```
<FORM ACTION="/delphiapp/termin_info" METHOD=GET>
<P>
Basisdatum: <INPUT name="BaseDate" type=text value="<#BaseDateVorgabeText>">
<input type="checkbox" name="NoSort" value="1" <#NoSortVorgabeWert>>
Tabelle nicht sortieren<BR><BR>
<input type=submit value="Erneuern">
</P>
</FORM>
```

Um die transparenten Tags zu ersetzen, wird die bereits bekannte Methode für das *OnHTMLTag*-Ereignis des PageProducers erweitert. Als Vorgabewerte sollen die aktuellen Anfrageparameter dienen, die aus *Request.QueryFields* abgerufen werden können.

Die Vorgabewerte stehen somit erst zur Laufzeit fest, was der Grund dafür ist, dass wir sie oben nicht von vorneherein statisch in den HTML-Code eintragen konnten.

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'AktuelleZeit' then ... (siehe Listing oben)
  else if TagString = 'NaechsterTermin' then ...
  // zwei neue transparente Tags:
  else if TagString = 'BaseDateVorgabeText' then
    ReplaceText := Request.QueryFields.Values['BaseDate']
  else if TagString = 'NoSortVorgabeWert' then begin
    if Request.QueryFields.Values['NoSort'] = '1' then
      ReplaceText := 'checked';
end;
```

Hinweis: Die aktuellen Parameter der Anfrage entsprechen den zuletzt vom Benutzer im HTML-Formular gemachten Einstellungen. Wenn die Delphi-Anwendung diese Einstellungen *nicht* als Vorgabewerte in das generierte HTML-Dokument übernehmen würde, würden die Einstellungen nach dem Drücken des Schalters ERNEUERN verloren gehen (das Editierfeld würde leer sein und die Checkbox wäre nicht markiert) und ein weiterer Druck auf ERNEUERN hätte eben nicht die erwartete erneuernde, sondern eher eine löschende (zurücksetzende) Wirkung.

Debuggen von Web-Anwendungen

Grundsätzlich lassen sich Web-Anwendungen, die in Form von DLLs vorliegen, wie normale DLLs debuggen (siehe R142 in Kapitel 8.5.3 auf Seite 1109). Sie können also in der IDE unter START | PARAMETER und unter *Host*-Anwendung beispielsweise *Apache.exe* (mit zugehörigem Verzeichnispfad natürlich) angeben und diese dann mit START | START ausführen. Der Apache-Server besteht jedoch aus mehreren Prozessen und der Start der DLL findet in einem anderen Prozess statt als dem, der direkt von der Delphi-IDE gestartet wird. Daher müssen Sie in den Debugger-Optionen (TOOLS-Menü) auch noch die Option DEBUG IN SPAWNED-PROZESSEN anwählen, sonst kann der Debugger Ihre Anwendung nicht bei den gewünschten Haltepunkten anhalten.

Hinweise: Um die so gestartete Ausführung des Apache-Servers zu beenden, drücken Sie in seinem Konsolenfenster Strg+C. Gegebenenfalls kommt es beim Ablauf des Apache-Servers im Debugger zu Unterbrechungen, die nicht auf einen Ihrer Haltepunkte zurückzuführen sind. Diese können Sie ignorieren, indem Sie das Programm einfach weiterlaufen lassen.

Der Web-Anwendungs-Debugger

Eine nützliche Neuerung von Delphi 6 ist der Web-Anwendungs-Debugger (Abbildung 8.28). Bei diesem handelt es sich um einen Web-Server, der Sie auf Ihrem Entwicklungs-PC unabhängig von ausgewachsenen Web-Servern wie Apache oder IIS macht und der vor allem mit einem Protokoll der empfangenen HTTP-Anfragen sowie der von der Web-Anwendung zurückgelieferten Antworten aufwarten kann.

Um eine Anwendung mit diesem Debugger untersuchen zu können, muss zunächst ein spezielles COM-Objekt für diese Anwendung erzeugt und registriert werden. Letztlich führt dies dazu, dass Sie für die Debugging-Version der Anwendung ein neues Web-Anwendungs-Projekt anlegen müssen, und zwar mit dem gleichen Experten, den Sie auch für das Anlegen der »normalen« Anwendungsversion verwendet haben. Für die Beispielanwendung etwa wurde mit DATEI | NEU | WEITERE | NEU | WEB-SERVER-ANWENDUNG der in Abbildung 8.21 gezeigte Dialog aufgerufen (für WebSnap-Anwendungen wird ein anderer Dialog verwendet, der jedoch über die gleiche Wahlmöglichkeit verfügt, was den Typ der Anwendung betrifft). Rufen Sie den für Ihre Anwendung passenden Dialog auf und wählen Sie diesmal die Option WEB-DEBUGGER-ANWENDUNG, um ein neues Anwendungsgerüst zu erzeugen. Unter NAME DER COCLASS tragen Sie einen Namen ein, unter dem die Debugging-Version der Anwendung im Windows-COM registriert werden soll (dies ist die erwähnte Voraussetzung für die Funktion des Debuggers).

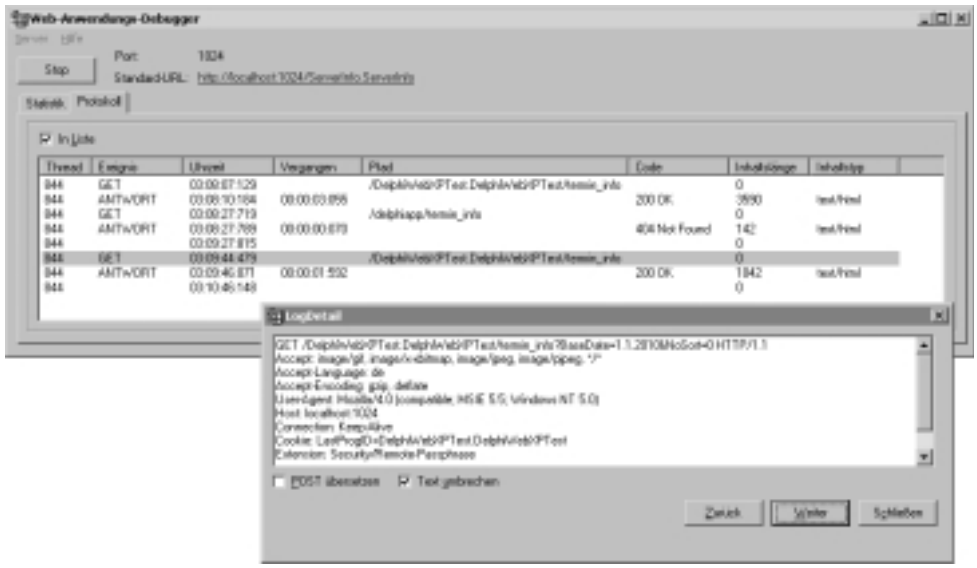


Abbildung 8.28: Der Web-Anwendungs-Debugger beim Protokollieren des bei der Nutzung des Beispielprogramms entstehenden HTTP-Verkehrs

Das neu erzeugte Projekt enthält ein leeres Formular, das zur Laufzeit der Anwendung im Debugger als zusätzliche Rückmeldung erscheint, und ein leeres Webmodul. Wenn Sie auf dieser Grundlage ein neues Projekt starten, können Sie das vom Experten erzeugte Gerüst so verwenden, wie es ist. Wenn Sie eine bestehende Web-Server-Anwendung zur Untersuchung im Debugger umstellen wollen, sehen die zusätzlich erforderlichen Schritte wie folgt aus:

- Das leere Webmodul entfernen Sie und ersetzen es durch das Webmodul Ihrer zu untersuchenden Anwendung.
- Binden Sie die Unit *WebReq* in die Unit ein (falls noch nicht geschehen) und stellen Sie sicher, dass der *initialization*-Teil der Unit eine Anweisung nach dem folgenden Muster enthält (*TWebModule1* ist durch die Klasse Ihres Webmoduls ersetzen):

```
if WebRequestHandler <> nil then
  WebRequestHandler.WebModuleClass := TWebModule1;
```

In jedem Fall müssen Sie Ihr Debugging-Projekt einmal aus der Delphi-IDE starten, denn es handelt sich um eine EXE-Datei, die bei ihrer ersten Ausführung die für den Debugger notwendige Co-Klasse registriert. Dies schließt die Vorbereitungsmaßnahmen ab – die Anwendung ist nun für den Ablauf im Debugger bereit.

Starten Sie jetzt den Debugger aus der Delphi-IDE mit TOOLS | WEB-ANWENDUNGS-DEBUGGER und aktivieren Sie ihn mit dem in Abbildung 8.21 erkennbaren START-Schalter. Nun können Sie in einem Web-Browser Seiten der zu untersuchenden Web-Anwendung aufrufen, allerdings unter einer anderen Adresse, als wenn Sie die Anwendung etwa mit Apache gestartet hätten.

Als Beispiel soll die Debugger-Version der in diesem Kapitel besprochenen Anwendung *DelphiWebXP* dienen. Sie finden diese Version auf der CD unter dem Namen *DelphiWebXPTest*. Sie wurde wie oben beschrieben aus der normalen Apache-Version der Anwendung erzeugt. Die Adresse der Anwendung im Debugger lautet:

```
http://localhost:1024/DelphiWebXPTest.DelphiWebXPTest/
```

... wobei diese wie gehabt durch die Pfad-Erweiterungen `termin_info` oder `request_info` ergänzt werden kann. Sofern Sie die Anwendung vorher durch einmaligen Start registriert haben, startet der Debugger die Anwendung bei Aufruf dieser Adressen selbstständig (die Anwendung muss also nicht vorher manuell aus der Delphi-IDE gestartet werden). Für jede Anfrage des Web-Browsers erzeugt der Debugger wie in der Abbildung gezeigt einen Protokolleintrag. Durch einen Doppelklick darauf können Sie den kompletten Anfrage- bzw. Antworttext abrufen.

Mit den bisher beschriebenen Maßnahmen wird der Web-Anwendungs-Debugger praktisch zu einem Ersatz eines richtigen Web-Servers auf dem lokalen PC. Er befähigt die Anwendungen dazu, im Kontext eines Web-Servers zu laufen und Anfragen von

Web-Browsern zu beantworten. Letztlich sollte es jedoch auch der Sinn des Debuggers sein, eine Anwendung schrittweise untersuchen zu können.

Dafür muss noch der Debugger der Delphi-IDE so aktiviert werden, dass er in den laufenden Betrieb des Web-Anwendungs-Debuggers eingreifen kann. Hierfür sieht Delphi den Menüpunkt START | MIT PROZESS VERBINDEN vor. Zunächst muss jedoch der Web-Anwendungs-Debugger gestartet und das zu untersuchende Projekt in der Delphi-IDE geladen sein. Daraufhin können Sie den Debugger über den genannten Menüpunkt mit dem Prozess *webappdbg.exe* VERBINDEN. Setzen Sie einen Haltepunkt in Ihrer Web-Anwendung und rufen Sie die passende Adresse in einem Web-Browser auf. Der Debugger sollte Ihr Programm nun am Haltepunkt anhalten (eventuell zusätzliche Unterbrechungen bei einem *ret*-Befehl im CPU-Fenster können ignoriert werden).

8.8.3 Seitenproduzenten von WebSnap

Die Datentabelle aus dem Beispiel im letzten Abschnitt wurde mit einer *TDataSetTableProducer*-Komponente erstellt, die bereits zum Lieferumgang der Professional-Version von Delphi gehört. In diesem Abschnitt wollen wir einen Blick auf die exklusiven Neuerungen der Enterprise-Version von Delphi 6 werfen. Hier finden Sie eine zusätzliche Toolbar, die Sie über ANSICHT | SYMBOLLEISTEN | INTERNET einschalten können sowie die Palettenseite *WebSnap* mit einer Vielzahl neuer Komponenten. Den Sinn und Zweck dieser Komponenten müssen Sie sich jedoch nicht gleich einzeln einprägen, sondern ihre Verwendung ergibt sich später mehr oder weniger von selbst, wenn Sie die Internet-Toolbar und das Objekt-Hierarchie-Fenster verwenden.

Auch in der WebSnap-Bibliothek sind *PageProducer*-Komponenten für die Generierung von HTML- bzw. XML-Seiten zuständig, jedoch sind diese wesentlich komplexer als die im vorigen Kapitel gesehenen Generatoren. Zu den durch sie bereitgestellten Zusatzfunktionen gehören zunächst ein sehr vielseitiger Editor und eine Vorschaufunktion, wie sie unter Delphi 5 schon von den InternetExpress-Komponenten bekannt war. Im Endeffekt führen die WebSnap-Komponenten zu einem völlig neuen Grad an Automatisierung, was die Funktion der Web-Anwendung zur Laufzeit betrifft. Und bei einem Blick auf die interne Arbeitsweise könnte man fast zu der Vermutung kommen, Borland wolle die Object-Pascal-Programmierung abschaffen: Die von WebSnap generierten Seiten werden Server-seitig durch Active Scripting gesteuert. Die Skripte sind standardmäßig in JScript verfasst – praktisch bedeutet dies, dass hier über JScript-Anweisungen auf die Objekte von Object Pascal zugegriffen wird. Damit dieser Zugriff möglich wird, muss WebSnap zahlreiche Adapter-Komponenten bereitstellen, was die hohe Zahl von Komponenten mit dem Wort *Adapter* im Namen erklärt. Diese Implementierungs-Details können Sie bei der Erstellung einer WebSnap-Anwendung jedoch getrost unbeachtet lassen.

Auch bei der Erstellung einer WebSnap-Anwendung ist zunächst nicht so viel von Programmierung die Rede: Wenn Sie das mit Delphi mitgelieferte WebSnap-Tutorial durcharbeiten, erhalten Sie in über 100 Einzelschritten eine komplette Web-Server-Anwendung, mit der der Endanwender über seinen Web-Browser eine Datenbank nicht nur ansehen, sondern auch editieren kann. Bei all diesen Einzelschritten wird jedoch nicht eine einzige Zeile Code verfasst, ja die Tastatur wird sogar generell kaum verwendet. Man kann also sagen, dass der Begriff »Snap« sehr passend von Borland gewählt wurde – die Maus klickt und schiebt und die Komponenten schnappen zusammen.

Editieren von Datenbankinformationen über den Web-Server

Das Hauptaugenmerk in diesem Abschnitt soll auf der wichtigsten Funktion liegen, die WebSnap den in Kapitel 8.8.2 beschriebenen Web-Broker-Fähigkeiten hinzufügt: Die Möglichkeit, die angezeigten Datenmengen auch editieren zu können.

Das auf der CD befindliche Beispielprojekt *TermineWebEdit* bietet zunächst wieder eine Tabelle aller Termine (Abbildung 8.29), die jedoch diesmal über einen Seitengenerator von WebSnap generiert wurde. Wenn Sie den EDIT-Schalter neben einem der aufgelisteten Datensätze drücken, gelangen Sie auf eine Seite, in der Sie die Felder des Datensatzes editieren können (Abbildung 8.30).

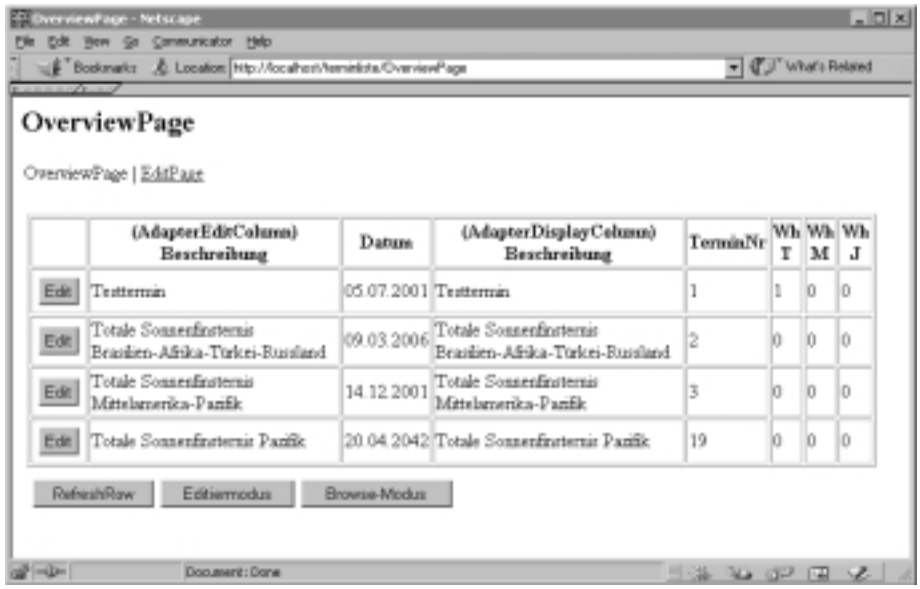


Abbildung 8.29: Die »Startseite« der WebSnap-Anwendung wird durch das Haupt-Webmodul bereitgestellt (Basisklasse TWebAppPageModule); die Komponente, die die Tabelle generiert, ist ein TAdapterGrid, welches einem TAdapterPageProducer untergeordnet ist.

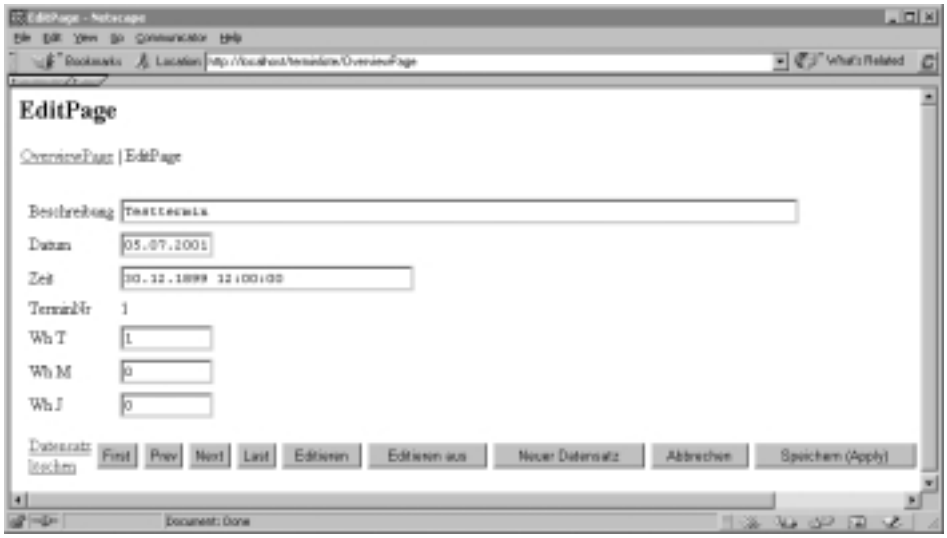


Abbildung 8.30: Die zum Editieren gedachte Seite der Anwendung wird von einem zweiten AdapterPageProducer in einem zweiten Webseitenmodul (Basisklasse TWebPageModul) erzeugt, für den Editierbereich ist eine AdapterFieldGroup zuständig.

Drücken Sie auf dieser Editierseite den Schalter APPLY, werden die Inhalte der Eingabefelder von WebSnap automatisch in die Datenbank übernommen, nur bei Verwendung von DBExpress müssen Sie noch einen kleinen manuellen Arbeitsschritt selbst durchführen. Bis auf diese Ausnahme funktioniert die Anwendung ganz ohne selbst geschriebenen Code.

Exkurs Professional-Version: Auch wenn Sie nicht über die Enterprise-Version von Delphi verfügen, können Sie natürlich immer noch recht einfach solche Editierfähigkeiten in Ihre Web-Broker-Anwendung einbauen. Eine der vielen denkbaren Varianten, das Editieren mit Hilfe des in Kapitel 8.8.2 beschriebenen Rüstzeugs zu implementieren, könnte etwa wie folgt aussehen: Sie definieren eine neue Aktion im Webmodul, beispielsweise mit dem *PathInfo* »Edit« , und erweitern Ihre HTML-Seite (etwa die HTML-Schablone eines Seitengenerators) um ein HTML-Formular. Dieses sollte Eingabefelder für die editierbaren Felder eines Datensatzes enthalten sowie (ganz wichtig!) ein Feld, das den zu editierenden Datensatz per eindeutigem Schlüssel festlegt (z.B. die Terminnummer). In der *OnAction*-Methode für die anfangs definierte Webmodul-Aktion können Sie nun die neuen Werte der Datenfelder aus den Parametern der HTTP-Anfrage auslesen (*Request.QueryFields*) und mit Hilfe von Datenbank-Operationen das Update vornehmen. Etwaige Erfolgs- oder Fehlermeldungen schreiben Sie in den Parameter *Response.Content* der Webmodul-Aktion.

Grundtechniken beim Entwurf von WebSnap-Seitenmodulen

Im Folgenden wird nun der fertige Aufbau des Beispielprogramms vorgestellt. Die Schritte, mit denen Sie es nachbauen können, werden nicht im Detail aufgezählt, denn wie bei der Konstruktion von Windows-Formularen genügt es auch zum Aufbau eines WebSnap-Seitenmoduls, einige wenige Grundtechniken zu kombinieren und wiederholt anzuwenden, bis die gewünschte Struktur erreicht ist. Eine Schritt-für-Schritt-Anleitung zu WebSnap ist zudem Teil der mit Delphi mitgelieferten Dokumentation[24].

Die Grundtechniken, die Sie zum Entwurf eines WebSnap-Moduls benötigen, stammen aus dem Entwurf von Formularen:

- Zusammenschalten mehrerer Komponenten aus der Komponentenpalette durch Setzen von Properties im Objektinspektor: Da für WebSnap eine relativ hohe Zahl von Komponenten zusammengeschaltet wird, hat Borland Delphi mit der Fähigkeit versehen, die grundlegenden Gerüste der WebSnap-Module automatisch zu generieren.
- Editieren der Komponenten mit Hilfe des Objekt-Hierarchie-Fensters: Dieses Fenster erleichtert nicht nur durch seine Hierarchie-Darstellung den Überblick über die einzelnen Komponenten erheblich, sondern lässt Sie auch bei verschiedenen Gelegenheiten Komponenten hinzufügen, die sich nicht in der Komponentenpalette befinden.
- Editieren von Komponenten in speziellen Editoren, die auf verschiedenen Wegen aufgerufen werden können: im Webmodulfenster über das Kontextmenü der Komponente, in der Objekthierarchie über das Kontextmenü und im Objektinspektor über die altbekannten Schalter mit den drei Punkten.

Die Module einer WebSnap-Anwendung

Abbildung 8.31 zeigt die oberste Ebene der Komponenten des Beispielprogramms. Manche dieser Komponenten enthalten weitere Komponenten (oder allgemein gesagt Objekte), die Sie über das Fenster der Objekt-Hierarchie erreichen (die Detailstruktur der beiden Beispielmodule wird in den Abbildungen 8.32 und 8.34 gezeigt). Beide Beispielmodule sind ähnlich wie Datenmodule zu handhaben und zu jedem Modul gibt es natürlich eine Pascal-Unit (die beiden Units des Beispielprogramms enthalten allerdings insgesamt nur eine einzige selbst definierte Methode).

Die Aufgabe jedes Moduls ist die Bereitstellung jeweils einer Web-Seite, weshalb sie genauer als *Seitenmodule* bezeichnet werden. Ganz genau genommen handelt es sich beim Beispiel-Modul *OverviewPage* um ein Anwendungs-Seitenmodul (*TWebAppPage-*

24 Die ursprünglich Mitte 2001 ausgelieferte Dokumentation wurde von Borland sehr schnell überarbeitet, unter *www.borland.com/techpubs/delphi* können Sie sich jeweils den aktuellen Stand der Dokumentation herunterladen.

Module), beim Modul *EditPage* um ein normales Seitenmodul (*TWebPageModule*). Jede WebSnap-Anwendung benötigt genau ein Anwendungsmodul, welches zentrale Aufgaben, die für die gesamte Anwendung gelten, übernimmt (beispielsweise das Login der Benutzer; eine im Beispielprogramm nicht genutzte Option).

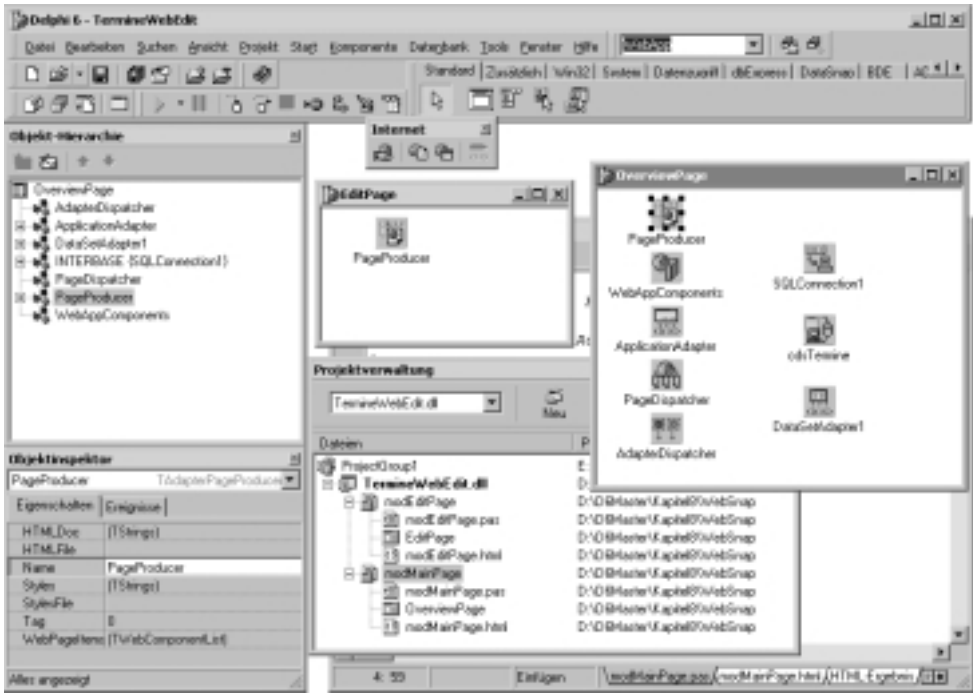


Abbildung 8.31: Die Module der Beispielanwendung in der Delphi-IDE wurden mit Experten-Hilfe aus der Internet-Toolbar (im Bild als abgetrennte Toolbar hervorgehoben) erzeugt. Die Toolbar kann über das Kontextmenü der anderen Toolbars eingeschaltet werden.

Hinweis: Für den Fall, dass Sie einige Komponenten in einem Modul zusammenfassen wollen, ohne dass dabei eine Webseite herauskommt, bietet WebSnap noch zwei Arten von Datenmodulen: *TWebAppDataModule* und *TWebDataModule.*

Das Hauptmodul der Beispielanwendung

Das Gerüst eines Anwendungsmoduls wird normalerweise mit Hilfe der Delphi-IDE erzeugt: Im Dialog von DATEI | NEU | WEITERE finden Sie auf der Seite *WebSnap* die Option *WebSnap-Anwendung*, die dem gleichnamigen Schalter aus der in Delphi 6 neu eingeführten Internet-Toolbar entspricht (siehe Abbildung 8.31). Im Dialog können Sie daraufhin wie bei einer WebBroker-Anwendung auswählen, für welchen Web-Server bzw. welche Schnittstelle (CGI/Win-CGI) die Anwendung gedacht ist. Zusätzliche

Optionen gestatten Ihnen die Auswahl verschiedener WebSnap-Features und die Benennung von Komponenten. Diese Optionen spielen jedoch hier keine Rolle, da für das Beispielprogramm die Standardeinstellungen verwendet wurden (als Web-Server wurde wieder Apache gewählt). Und die Benennung der Komponenten kann ohnehin noch im Objektinspektor geändert werden.

Das Ergebnis des Experten sind die fünf untereinander angeordneten Komponenten vom *PageProducer* bis zum *AdapterDispatcher* (Abbildung 8.31). Wir werden hier nur mit dem Seiten-Generator arbeiten und die restlichen Komponenten als Black-Box betrachten. Die weiteren Komponenten dienen dem Zugriff auf die Termintabelle der Datenbank: *SQLConnection1* und *cdsTermine* sind Kopien aus dem Terminverwaltungsprogramm von Kapitel 7.5 (diese Komponenten wurden ja auch schon im letzten Beispiel, *DelphiWebXP* in Kapitel 8.8.2, für den Zugriff auf die Termintabelle verwendet). Hinzu kommt mit dem *DataSetAdapter* eine manuell hinzugefügte Komponente von der WebSnap-Seite der Komponentenpalette. Er wird benötigt, um das *cdsTermine* mit den anderen WebSnap-Komponenten zu verbinden, und funktioniert weitgehend automatisch. Wir müssen nur sein *Dataset*-Property auf *cdsTermine* setzen.

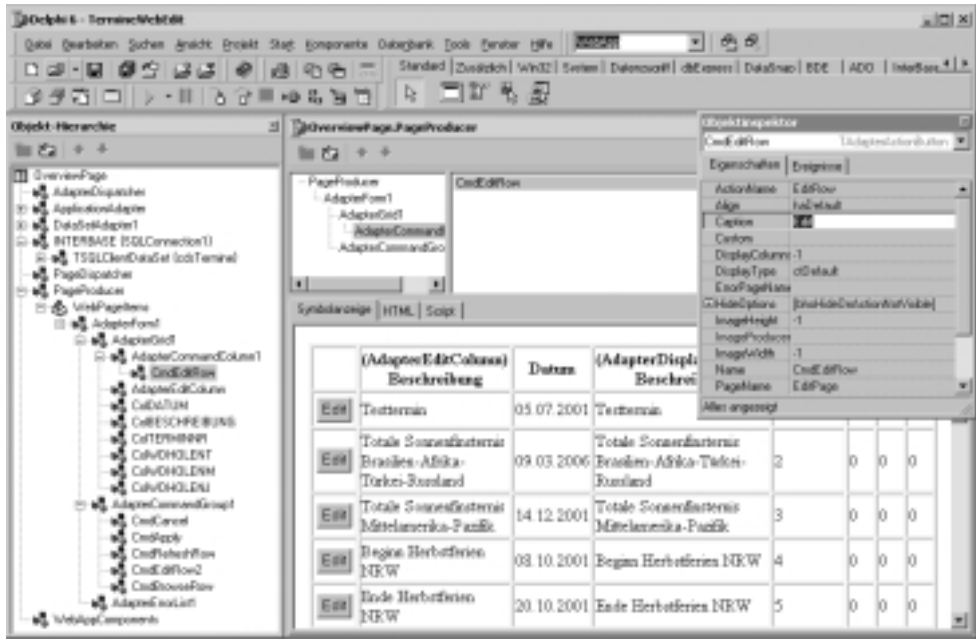


Abbildung 8.32: Die Objekt-Hierarchie der Tabellenseite (OverviewPage) und der dafür zuständige Seiteneditor

Nachdem dieses Komponentengerüst steht, können wir uns der Feinarbeit im Seiteneditor widmen. Das bedeutet, all das, was sich in Abbildung 8.32 unter dem Knoten *WebPageItems* befindet, hinzuzufügen. Dabei wird jeweils das Kontextmenü des Kno-

tens aufgerufen, unter dem die neuen Elemente angefügt werden sollen, und dann das gewünschte neue Element ausgewählt.

Hinweis: In der Abbildung ist der Knoten *WebPageItems* einmal im Fenster *Objekt-Hierarchie* und einmal im Seiteneditor zu sehen. Beide Darstellungen der Objekthierarchie verfügen über die gleiche Funktionalität und die gleichen Kontextmenüs. Allerdings werden im Seiteneditor die Objekte, denen keine weiteren Unterobjekte zugeordnet sind, nicht im TreeView, sondern in einer eigenen Liste rechts davon angezeigt.

Die im Beispielprogramm zu findenden Komponenten sind im Einzelnen:

- *AdapterForm1* dient lediglich als Container verschiedener anderer Seitenelemente und ist hier ansonsten nicht weiter von Bedeutung (Sie brauchen sich diese Komponente noch nicht einmal zu merken, weil Sie als Unterknoten von *WebPageItems* sowieso kaum etwas anderes wählen können).
- Ein *AdapterGrid* als wichtigsten Bestandteil der *AdapterForm*. In diesem Grid werden die einzelnen Datensätze angezeigt (Abb. 8.29). Einzig wichtig ist hierbei, dass Sie das Property *Adapter* des AdapterGrids auf den *DataSetAdapter* des Seitenmoduls setzen. (Auch das brauchen Sie sich übrigens nicht zwingend zu merken, denn wenn Sie das Property vergessen zu setzen, wird in der Seitenvorschau eine entsprechende Warnung vor der Tabelle angezeigt!)
- Dem *AdapterGrid* untergeordnet sind für jede anzuzeigende Spalte eine Komponente der Klasse *TAdapterDisplayColumn*. Wenn eine Spalte auch editiert werden soll, wählen Sie statt dessen die Klasse *TAdapterEditColumn* (hierzu später mehr). Wie im Editor von *TDataSetTableProducer* (Kapitel 8.8.2) können Sie hier für jede Spalte diverse Darstellungsattribute einstellen. Die Beschriftung der Spalten wird per Voreinstellung aus in der Datenmenge definierten persistenten Feldern übernommen, kann aber im Objektinspektor noch angepasst werden. Wenn Sie keine Feldkomponenten definiert haben, zeigt das *AdapterGrid* einfach alle Spalten der Tabelle an.
- Für die Aktionsschalter, die im HTML-Formular in jeder Zeile der Tabelle erscheinen sollen, fügen Sie eine *AdapterCommandColumn* ein, im Beispiel hat diese nur ein weiteres Unterelement für den *Edit*-Schalter.
- Aktionsschalter, die unterhalb der Tabelle erscheinen sollen, werden über eine *AdapterCommandGroup* definiert, die dem *AdapterGrid* untergeordnet wird.
- Schließlich enthält das Beispielformular noch eine *AdapterErrorList*, in der etwaige Fehler bei der Verarbeitung (z.B. Zugriffskonflikte der in Kapitel 7.5.5 beschriebenen Art) automatisch angezeigt werden.

Es sei noch einmal daran erinnert, dass sämtliche der oben aufgelisteten Komponenten über das Kontextmenü der Objekt-Hierarchie hinzugefügt werden.

Die Funktion der Anwendung bis hier stellt sich wie folgt dar: Wenn Sie die Adresse der Anwendung in einem Internet-Browser angeben (zur Adresse siehe den folgenden Abschnitt über die Installation), erscheint eine vollständige Tabellendarstellung der Datenmenge *cdsTermine*. Wenn Sie den Edit-Schalter drücken, wird die gesamte Tabelle in den Editiermodus geschaltet. Sofern nicht für alle Spalten *AdapterDisplayColumn*-Komponenten, sondern auch *AdapterEditColumn*-Komponenten verwendet wurden, können Sie die Datensätze nun in der Tabelle editieren (siehe Abbildung 8.33). Und sofern es sich nicht um eine DBExpress-Anwendung handelt, können Sie die veränderten Daten auch mit dem *Apply*-Schalter in der Datenbank speichern (bei DBExpress ist ein später beschriebener, minimaler Kodier-Aufwand erforderlich).

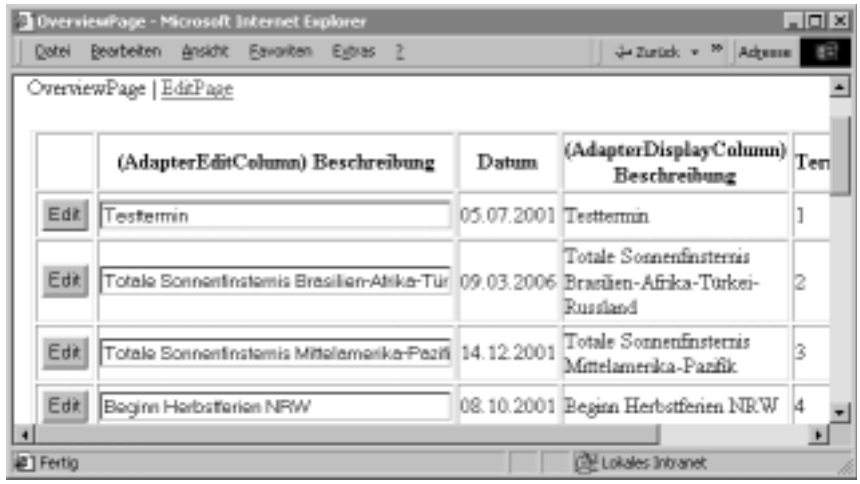


Abbildung 8.33: Die Termintabelle im Editiermodus. Da nur eine AdapterEditColumn-Komponente verwendet wurde, lässt sich auch nur eine Spalte editieren.

Hinweis: Sie können im Seiteneditor immer alle Arten von Aktionsschaltern hinzufügen, sowohl wenn diese für jeden Datensatz angezeigt werden (im Falle einer *AdapterCommandColumn*) als auch wenn sie in einer eigenen Gruppe untergebracht sind (*AdapterCommandGroup*). Allerdings sind nicht immer alle Aktionsschalter sinnvoll: So haben die Schalter *Next* und *Prev* nur dann eine Wirkung, wenn auf der Seite nur ein einzelner Datensatz angezeigt wird. Ein Editierenschalter funktioniert nur dann, wenn editierbare Formularelemente auf der Seite vorhanden sind. Und ob der Editieren-Schalter unterhalb der Tabelle oder in jeder einzelnen Zeile erscheint, spielt für die Funktion der Anwendung keine Rolle; es wird immer die gesamte Tabelle in den Editiermodus geschaltet.

In Beispiel aus Abbildung 8.33 macht der Editier-Schalter neben jedem Datensatz daher bisher noch keinen Sinn. Dies ändert sich, wenn demnächst eine zweite Seite hinzugefügt wird, in der nur noch dieser eine Datensatz dargestellt wird.

Installation der Beispielanwendung unter Apache

Da die Anwendung schon jetzt mit nur einem Seitenmodul funktioniert, sei als Nächstes beschrieben, wie Sie eine solche Anwendung ausprobieren können. Die Installation funktioniert ähnlich wie die Installation der einfachen Web-Broker-DLL aus Kapitel 8.8.2:

Die ausführbare DLL der Anwendung wird wieder in den Apache-Verzeichnisbaum kopiert (oder – während der Entwicklung – vom Compiler direkt dort hineingeschrieben). Angenommen, es wird das Verzeichnis `Apache\Modules` gewählt, müssen die folgenden Zeilen zur Konfigurationsdatei `conf\httpd.conf` hinzugefügt werden:

```
LoadModule terminewebedit_module Modules/TermineWebEdit.dll
<Location /terminliste>
  SetHandler terminewebedit-handler
</Location>
```

Der manuell angepasste Name des Projekts muss eventuell auch manuell in die Projektdatei (*TermineWebEdit.dpr*) eingearbeitet werden:

```
exports
  apache_module name 'terminewebedit_module'
```

Zusätzlich müssen die von Delphi automatisch erzeugten HTML-Schablonen für die beiden Seitenmodule ebenfalls in das *Modules*-Verzeichnis kopiert werden (in diesem Beispiel die Dateien *modOverviewPage.html* und *modEditPage.html*).

Wenn Apache dann gestartet wurde, können die Seiten im Internet-Browser unter den Adressen `http://localhost/terminliste/OverviewPage` und (sobald auch das zweite Seitenmodul hinzugefügt wurde) unter `http://localhost/terminliste/EditPage` erreicht werden.

Hinweis: Wenn Sie Apache nicht installiert haben, können Sie die Anwendung auch wie am Ende von Kapitel 8.8.2 beschrieben in eine Web-Debugger-Anwendung umwandeln und mit Hilfe des Web-Anwendungs-Debuggers laufen lassen (die zu verwendende URL ändert sich entsprechend, wie im genannten Kapitel gezeigt).

Das Seitenmodul für die Editierseite

Um eine WebSnap-Anwendung, die bisher nur aus dem Hauptmodul besteht, um eine zweite Seite zu erweitern, verwenden Sie entweder den Eintrag *WebSnap-Seitenmodul* aus der Objektgalerie des DATEI | NEU-Dialogs oder den zweiten Schalter aus der Internet-Toolbar. Für das Beispielprogramm wurde im darauf erscheinenden Dialog als Generator-Typ *TAdapterPageProducer* gewählt, alle anderen Dialogoptionen blieben bei ihren Voreinstellungen. In Abbildung 8.34 sehen Sie die zum Seiteneditor alternative Darstellung der Seitenvorschau im Editorfenster.

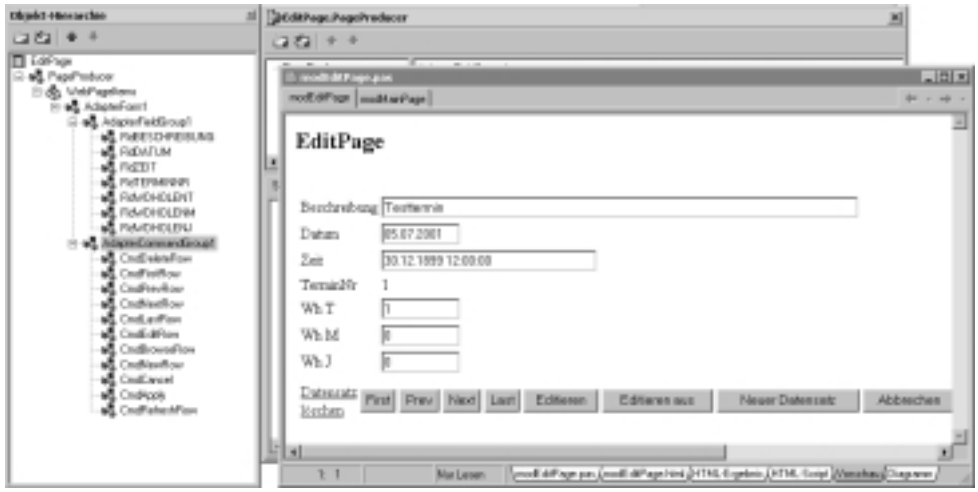


Abbildung 8.34: Die Objekt-Hierarchie der Editierseite EditPage. Der Seiteneditor ist hier im Hintergrund verborgen, dafür ist das in Delphi 6 neu eingeführte Register für das Editorfenster zu sehen, hier mit aufgeschlagener Vorschau-Seite.

Der neue Seitengenerator wird auf die gleiche Weise editiert wie der Generator der ersten Seite. Alle im Beispielprogramm verwendeten Komponenten sind in der Abbildung gezeigt. Als Unterelemente der *AdapterForm* wurde nun statt des *AdapterGrids* eine *AdapterFieldGroup* gewählt. Darunter wurden alle Spalten der Termintabelle bis auf Aktions- und Adressnummer eingefügt. Die Beschriftung der Schalter der *AdapterCommandGroup* wurde im Objektinspektor angepasst. Dies alles hat jedoch auf die Funktion der Anwendung keine Auswirkungen und kann in der Praxis meist nach persönlichem Geschmack variiert werden.

Es gibt aber noch eine für die Funktion der Anwendung wichtige Property-Einstellung: Der Anwender soll ja nun, wenn er auf der Tabellenseite den *Edit*-Schalter neben einem Datensatz klickt, auf diese zweite Seite gelangen, um den Datensatz dort editieren zu können. Hierzu genügt es, dass wir zum Hauptmodul (*OverviewPage*) wechseln,

in der Objekthierarchie die Komponente auswählen, die den EDITIEREN-Schalter repräsentiert (*CmdEditRow* in Abbildung 8.32) und im Objektinspektor dessen Property *PageName* auf *EditPage* setzen.

Sofern die Anwendung bereits wie oben beschrieben installiert wurde und die Compiler-Ausgabe auf das in Apache konfigurierte Verzeichnis stattfindet, genügt ein Neustart aus der Delphi-IDE und ein Neuladen der Seite im Internet-Browser, um die erweiterte Anwendung auszuprobieren. Sie können nun auf der Editierseite auch mit einem Schalter zurück in den Browse-Modus schalten (dieser *Browse*-Schalter ist in der Abbildung mit »Editieren aus« beschriftet). Die Eingabefelder wandeln sich dann in statische Textanzeigefelder um.

ApplyUpdates in der Web-Server-Anwendung

Damit das Editieren der Daten in der WebSnap-Anwendung wie erwartet funktioniert, d.h., damit die Daten auch in der Datenbank gespeichert werden, müssen zwei Voraussetzungen erfüllt sein:

- Bei der Datenbank-*Post*-Operation müssen die Daten auch in die Datenbank geschrieben werden. *Post* wird von der WebSnap-Anwendung automatisch ausgeführt, wenn ein Apply-Schalter gedrückt wird (in der Beispielanwendung kommt ein solcher Apply-Schalter auf der Editierseite vor, er ist dort mit *Speichern* beschriftet[25]), aber auch implizit bei anderen Operationen, etwa wenn eine Grid-Ansicht vom Editier- in den Browse-Modus zurückgeschaltet wird. Wenn für den Zugriff auf die Datenbank die BDE verwendet wird, werden die Daten durch das *Post* automatisch in der Datenbank gespeichert (sofern Sie dieses Verhalten nicht selbst abgeschaltet haben, etwa durch Verwendung von Cached Updates). Anders in einer DBExpress-Anwendung: Hier findet ein Speichern der Daten grundsätzlich *nicht* automatisch statt.
- Damit Delphis Datenzugriffskomponenten beim Speichern auch wissen, welcher Datensatz gemeint ist, müssen sich alle Datensätze durch einen Schlüssel eindeutig identifizieren lassen. Wenn ein solcher Schlüssel nicht schon durch eine Datenbankdefinition vorgegeben ist, können Sie ihn mit Hilfe der statischen Feldkomponenten definieren, indem Sie im *ProviderFlags*-Property aller Felder, die im Schlüssel enthalten sein sollen, das Flag *pfInKey* einschalten.

25 Ob ein Schalter ein Apply-Schalter ist, richtet sich nicht nach der Beschriftung, sondern nach dem Property *ActionName* des zugehörigen *AdapterActionButtons*. Dieses Property wird automatisch eingestellt, wenn Sie die AdapterActionButtons wie oben beschrieben über das Kontextmenü der Objekthierarchie erzeugen.

Hintergrund: Der Schlüssel für das Finden der Datensätze ist deshalb erforderlich, da WebSnap-Anwendungen zustandlose Anwendungen sind – sie merken sich also nicht, welcher Datensatz gerade im HTML-Formular angezeigt bzw. editiert wird. Konkret bedeutet das, dass die WebSnap-Anwendung unter Umständen für jede an sie gerichtete HTTP-Anfrage neu gestartet wird.

So wird dann zum Beispiel im ersten Programmlauf die Anzeige der zu editierenden Daten durch die Seitengeneratoren aufgebaut und an den Browser-Client gesendet. Während der Client die Daten editiert, wird die Web-Anwendung vielleicht beendet (falls es sich um eine CGI-Anwendung handelt, wird sie danach auf jeden Fall beendet, andernfalls kommt es darauf an, ob sie gleichzeitig noch andere Anfragen von anderen Clients zu bearbeiten hat, welche dann in getrennten Ausführungs-Threads in eigenen Kopien der Seitenmodule bearbeitet würden).

Wenn der Client die editierten Daten absendet, kann man also davon ausgehen, dass die Anwendung neu gestartet wird und nichts mehr weiß, welche Daten sie zuletzt an den Client gesendet hat. Daher muss der Client als Zusatzinformation angeben, auf welchen Datensatz sich die durchzuführende Aktion bezieht, und dies geschieht über die Schlüsselfelder (die Übermittlung der Schlüsselfelder wird natürlich automatisch von WebSnap bzw. von den durch WebSnap generierten HTML-Seiten übernommen).

Während letztere Voraussetzung im Beispielprogramm automatisch erfüllt ist, da die Terminnummer in der Interbase-Tabelle als Schlüssel definiert wurde (siehe SQL-Listing in Kapitel 7.1.3), muss die erste Voraussetzung muss durch zwei selbst geschriebene Code-Zeilen erfüllt werden. Ähnlich wie schon in Kapitel 7.5.4 wird einfach bei jeder *Post*-Operation die *ApplyUpdates*-Methode der Datenmenge aufgerufen. Die folgende Methode wird also mit dem *AfterPost*-Ereignis von *cdsTermine* verknüpft:

```
procedure TOverviewPage.cdsTermineAfterPost(DataSet: TDataSet);
begin
  if cdsTermine.ApplyUpdates(-1) > 0 then
    cdsTermine.CancelUpdates
end;
```

Falls es hierbei zu Fehlern kommen sollte, wird allerdings nicht wie in Kapitel 7.5.5 ein aufwändiges Rückmeldeformular angezeigt, sondern die Änderungen werden mit *CancelUpdates* einfach verworfen.

Eine Alternative zu der obigen Ereignisbearbeitung wäre übrigens, das Ereignis *AfterExecuteAction*-Ereignis des *DataSetAdapters* zu bearbeiten:

```
procedure TOverviewPage.DataSetAdapter1AfterExecuteAction(Sender,
  Action: TObject; Params: TStrings);
begin
  if cdsTermine.ChangeCount > 0 then
```

```
    if cdsTermine.ApplyUpdates(-1) > 0 then
      cdsTermine.CancelUpdates
end;
```

Da dieses Ereignis bei *jeder* Aktion der Adapterkomponente auftritt, kann es sein, dass gar keine Änderungen gemacht wurden, weshalb vorher noch *ChangeCount* abgefragt wird.

8.8.4 Web Services

Während es im weltweiten Datenverkehr über das Internet bereits als alltäglich aufgefasst werden kann, dass Menschen untereinander Informationen im HTML-Format austauschen, haben die Programme auf verschiedenen Rechnern immer noch gewisse Schwierigkeiten, sich gegenseitig zu verstehen. »Web-Dienst« ist eine allgemeine Bezeichnung für eine Anwendung, die ihre Funktionalität anderen Anwendungen über das HTTP-Protokoll zugänglich macht. In diesem Kapitel geht es um die spezielle Form von Web-Diensten, die von Delphi 6 unterstützt wird: den Diensten, die das Simple Object Access Protocol (SOAP) verwenden.

SOAP-Anwendungen stellen ihre Dienste in Form von Objekten zur Verfügung, in gewisser Weise vergleichbar mit dem in Kapitel 8.7.4 behandelten DCOM. Ein fundamentaler Unterschied zu DCOM liegt jedoch darin, dass SOAP ein zustandsloses Protokoll ist, was bedeutet, dass der Zustand der Objekte zwischen einzelnen Methodenaufrufen nicht im Objekt selbst gespeichert wird – die Objekte werden bei jedem Funktionsaufruf neu initialisiert.

Randbemerkung: Wenn SOAP-Objekte einen Zustand zwischen mehreren Aufrufen aufrechterhalten sollen, müssen sich die Objekte selbst um diese Speicherung kümmern, etwa indem sie den Status einer Anfrage des Clients in einer Datei speichern. Bei der nächsten Anfrage des Clients können sie die vorher gespeicherten Informationen dann aus der Datei wieder einlesen, allerdings muss noch beachtet werden, dass es eventuell mehrere Clients gleichzeitig geben kann. In diesem Fall muss auch noch ein Mechanismus eingeführt werden, die einzelnen Clients zu unterscheiden (z.B. über die IP-Adresse oder eine ID, die bei jedem Funktionsaufruf übergeben wird).

SOAP-XML-Dokumente

Die Aufrufe der Dienst-Methoden werden in Form von XML-Nachrichten von Rechner zu Rechner transportiert; als Transportprotokoll dient bisher nur HTTP, allerdings sind weder SOAP noch Delphi für alle Zeiten auf dieses Protokoll festgelegt. Wenn ein SOAP-Objekt aufgerufen wird, sendet also der SOAP-Client eine HTTP-Anfrage,

bestehend aus einem HTTP-Header und einem XML-Dokument als Inhalt, an den Server. Folgendes Beispiel ist eine Anfrage des Beispielprogramms dieses Kapitels:

```
POST /GetNextDateDebug.GetNextDateDebug/soap/ HTTP/1.1
Accept: application/octet-stream, text/xml
SOAPAction: "urn:GetNextDateIntf-IGetNextDate"
Content-Type: text/xml
User-Agent: Borland SOAP 1.1
[...weitere Standard-HTTP-Header-Infos gekürzt...]

<?xml version="1.0" encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <NS1:GetNextDate xmlns:NS1="urn:GetNextDateIntf-IGetNextDate"
      SOAP-ENV:encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/">
      <NS1:BaseDate xsi:type="xsd:string">1.1.2002</NS1:BaseDate>
    </NS1:GetNextDate>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Das vom SOAP-Standardisierungsgremium vorgeschlagene Papier legt hier unter anderem Regeln fest, nach denen der »Umschlag« (Envelope) und der Hauptteil (Body) der SOAP-Anfrage aufgebaut sein muss. Die oben fett gedruckten Teile kennzeichnen die Informationen, die man auch in Object Pascal gebraucht hätte: *IGetNextDate.GetNextDate* soll aufgerufen werden mit einem *string*-Parameter (Name: *BaseDate*), der den Wert »1.1.2002« enthält.

Der SOAP-Server schickt daraufhin im Idealfall eine Antwort wie die Folgende zurück; die vier *xmlns*-Angaben im *Envelope* sind der Übersichtlichkeit halber gekürzt:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 572
Content:

<?xml version="1.0" encoding='UTF-8'?>
<SOAP-ENV:Envelope
  [...Envelope-Attribute wie im ersten Listing...]
  <SOAP-ENV:Body>
    <NS1:GetNextDateResponse
       xmlns:NS1="urn:GetNextDateIntf-IGetNextDate"
       SOAP-ENV:encodingStyle=
                 "http://schemas.xmlsoap.org/soap/encoding/">
       <NS1:return xsi:type="xsd:string">
          05.01.2002: Ende Weihnachtsferien NRW
```

```
        </NS1:return>
      </NS1:GetNextDateResponse>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

Eine wichtige Rolle spielt außerdem die *Web Services Description Language* (WSDL). Aus der Sicht von Object Pascal dient diese Sprache dazu, die Schnittstellen und die Aufrufformate der Methoden zu beschreiben, die ein Web-Dienst anbietet. Eine solche Beschreibung ist erforderlich, damit Web-Dienste von verschiedenen Programmiersprachen aus komfortabel genutzt werden können. Als Beispiel soll ein kleiner Auszug aus der WSDL-Datei des Beispielprogramms dienen (das vollständige Listing wäre ca. eine Seite lang und soll Ihnen nicht zugemutet werden, zumal Sie aus obigen Listings bereits einen Eindruck gewinnen konnten, wie »ausführlich« diese XML-Dateien formuliert sind).

```
<definitions ...>
  ... (Definitionen von Bindings, Port Typen u.a. weggelassen)
  <service name="IGetNextDateservice">
    <port name="IGetNextDatePort" binding="IGetNextDatebinding">
      <soap:address location
        ="http://localhost/GetNextDate/soap/IGetNextDate"/>
    </port>
  </service>
</definitions>
```

Die Aufgabe der Entwicklungs-Tool-Hersteller besteht nun darin, die Handhabung dieser nicht gerade auf die Verarbeitung durch einen Menschen zugeschnittenen XML-Dokumente zu automatisieren, so dass der Software-Entwickler gar nichts mehr davon mitbekommt, sondern die entfernten SOAP-Objekte auf anderen Rechnern so aufrufen kann wie seine eigenen Objekte – und das unabhängig davon, in welcher Programmiersprache die externen Objekte geschrieben wurden oder auf welchem Betriebssystem sie laufen.

Das SOAP-Toolkit von Delphi hat zumindest das erste dieser Ziele schon in der ersten Version erreicht: Der Entwickler braucht sich nicht um den Aufbau und die Interpretation von XML-Dokumenten zu kümmern. Der zweite Teil – die Interoperabilität von SOAP-Anwendungen, die mit verschiedenen Entwicklungssystemen erstellt wurden, hält sich allgemein noch in Grenzen, wie es bei einer so jungen Technologie auch kaum anders zu erwarten gewesen wäre.

SOAP in Delphi

Die Enterprise-Ausgabe von Delphi 6 enthält spezielle Komponenten und Tools zur Implementierung von eigenen und zum Aufruf bestehender SOAP-Web-Services. In dem Zeitraum, in dem Delphi 6 auf den Markt kam, wurde SOAP vom Standardisierungsgremium W3C gerade von der »Note« zum »Working Draft« erhoben (Versions-

nummer 1.2[26]). Dies lässt noch einen langen Weg und viele neue Versionen bis zur Standardisierung vermuten, zumal das W3C nach dem Working Draft noch drei weitere Stufen von Spezifikationen kennt, bevor mit der »Recommendation« eine Art endgültiger Status erreicht wird. Entsprechend wird ja bereits bei der Installation von Delphi 6 Enterprise auf den vorläufigen Status von SOAP hingewiesen und auf die Tatsache, dass sich hier in Zukunft Änderungen ergeben können.

Jedoch lässt sich vermuten, dass sich solche Änderungen nicht groß auf den mit Delphi entwickelten Code auswirken werden, denn wie das Beispiel zeigen wird, kann man beim Codieren eines Web Services jeglichen direkten Kontakt mit XML und dem Aufrufprotokoll vermeiden. Sollte sich also der SOAP-Standard ändern, wird ein Update der SOAP-Implementierung in Delphi nötig sein, die Programme selbst werden sich möglicherweise mit minimalen Änderungen neu übersetzen lassen.

Wenn es nur darum geht, bestehende Dienste zu nutzen, kann man übrigens theoretisch auch mit Hilfe des frei erhältlichen SOAP-Toolkits von Microsoft in den kleineren Delphi-Versionen Web-Service-Clients entwickeln (die Schnittstellen dieses Toolkits können auf ähnliche Weise wie etwa auch der Internet Explorer in Kapitel 8.6.3 in Delphi importiert und dort über COM-Objekte aufgerufen werden). Dieses Kapitel wird sich jedoch ausschließlich auf die Delphi-eigenen Tools für Web Services konzentrieren, zumal es wie erwähnt um die Interoperabilität zwischen verschiedenen Tools noch nicht so gut bestellt ist – dies kann sich jedoch schnell ändern, weshalb sich aktuellere Zeitschriftenartikel mit diesem Thema befassen mögen.

Das Gerüst einer SOAP-Anwendung

Wie jedes Mal, wenn Borland einen bestimmten neuen Anwendungstyp unterstützt, finden Sie auch für eine SOAP-Anwendung einen neuen Eintrag in der Objektgalerie (DATEI | NEU | WEITERE | WEBSERVICES | SOAP-SERVER-ANWENDUNG). Zuerst gilt es wie schon in Kapitel 8.8.2, den Typ des Web-Servers auszuwählen (ISAPI / NSAPI, CGI, Win-CGI, Apache oder Web-Debugger) und wie in den vorangegangenen Kapiteln finden Sie auch das Web-Services-Beispiel auf der CD in einer Apache- bzw. Web-Debugger-Version.

Nach Auswahl des Server-Typs erhalten Sie ein neues WebBroker-Projekt, dessen äußere Schnittstelle wieder vom Server-Typ abhängt (CGI- und Web-Debugger-Anwendungen sind EXE-Dateien, die anderen Web-Server erfordern DLLs). Das neue Projekt enthält bereits ein Webmodul, wie es ebenfalls schon in Kapitel 8.8.2 in Erscheinung getreten ist. Neu sind die drei Komponenten, die Delphi automatisch in diesem Webmodul unterbringt (Abbildung 8.35).

26 *http://www.w3.org/TR/2001/WD-soap12-20010709*

Abbildung 8.35: Die Infrastruktur einer SOAP-Anwendung wurde von Borland modular angelegt und scheint auf zukünftige Entwicklungen gut vorbereitet zu sein.

Für die Aufrufbarkeit der Anwendung von außen über SOAP-Nachrichten sind die Komponenten *HTTPSoapDispatcher* und *HTTPSoapPascalInvoker* zuständig; wir können sie im Folgenden einfach so hinnehmen, wie Delphi sie in das WebModul abgelegt hat. Es ist nicht mehr erforderlich, einzelne Teile der HTTP-Anfrage zu interpretieren, wie wir es etwa in Kapitel 8.8.2 über Webmodul-Aktionen für zusätzliche Pfadangaben in der URL oder für HTTP-Anfrage-Parameter gemacht haben, denn die komplette Interpretation der HTTP-Botschaft wird von den Delphi-Komponenten übernommen, bis schließlich der *SoapPascalInvoker* die Methoden der von Ihnen bereitgestellten Objekte aufruft.

Neben der SOAP-SERVER-ANWENDUNG enthält die Seite WEBSERVICES der Objektgalerie übrigens noch die Objekte SOAP SERVER DATA MODULE und WEB SERVICES IMPORTER. Sie sind zur Entwicklung eines SOAP-Datenbankservers in einer mehrschichtigen Datenbankanwendungen bzw. zur Entwicklung von SOAP-Clients für Dienste, die nicht mit Delphi entwickelt wurden, gedacht, und können im Rahmen dieses Buchs nicht behandelt werden.

Vom transparenten Tag zum Web Service

Als Beispiel soll eine Information, die in Kapitel 8.8.2 noch über eine HTML-Seite an den Browser des Endanwenders geliefert wurde, nun über einen Web-Dienst abrufbar gemacht werden, und zwar die Information über den nächsten Termin, die im Programm *DelphiWebXP* an Stelle des transparenten Tags *<#NaechsterTermin>* eingesetzt wurde (Name des Projekts auf der CD: *GetNextDateService*). Dafür wird zunächst wie oben beschrieben eine neue SOAP-SERVER-ANWENDUNG erzeugt. Das Gerüst dieser Anwendung benötigt zweierlei Erweiterungen, um als SOAP-Anwendung funktionieren zu können:

- Ein im SOAP-Framework der VCL registriertes Interface, in dem die Methoden definiert sind, welche von den SOAP-Clients aufgerufen werden können.
- Eine ebenfalls registrierte Klasse, welche die Methoden des Interfaces implementiert.

Das Interface sollte in einer eigenen Unit definiert werden, damit es auch von eventuellen SOAP-Clients eingebunden werden kann. Und da diese SOAP-Clients ja die Implementation der Interfaces *nicht* enthalten sollen, sondern diese auf einem entfernten Rechner aufrufen, sollte auch die Implementation in einer separaten Unit untergebracht werden. Insgesamt werden also zwei Units benötigt, darin ein Interface und eine Klasse, sowie der Code, der beide registriert. Wenn Sie Delphi bei Borland registriert haben, können Sie vom Bereich für registrierte Benutzer auf Borlands Webseite ein kleines Paket von Zusatzkomponenten herunterladen, in dem sich auch der *Invokable Wizard* befindet. Mit ihm können Sie die beiden Unit-Gerüste dialoggesteuert erzeugen, was aber auch per Hand leicht möglich ist – nach dem Muster der nächsten beiden Listings. Zunächst die Unit mit dem Interface des Beispiel-Servers:

```
unit GetNextDateIntf;

interface

type
  IGetNextDate = interface(IInvokable)
  ['{5C6354DF-0FD1-4553-BB95-EE46D9C2BA67}']
     function GetNextDate(BaseDate: String): String; stdcall;
  end;

implementation

uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(IGetNextDate));
end.
```

Beim Anlegen einer solchen Interface-Unit gilt es die folgenden Punkte zu beachten:

- Das neue Interface muss von *IInvokable* abgeleitet werden, um mit umfangreichen Typinformationen kompiliert zu werden (alternativ dazu können Sie das Interface mit der Compileroption $M+ übersetzen, ohne es von *IInvokable* abzuleiten).
- Die Standard-Aufrufkonvention von Methoden ist für diese Art von Interfaces nicht geeignet, daher wird die Methode oben *stdcall* deklariert. (Vorsicht: Wenn Sie diesen Zusatz vergessen, erhalten Sie keine Fehlermeldung vom Compiler, sondern lediglich einen nicht funktionierenden SOAP-Server!)
- Die Typen der Parameter und Rückgabetypen müssen einfach sein, damit sie automatisch über SOAP-Botschaften übertragen werden können. Dynamische Arrays und komplexe Typen wie Arrays, Records und Mengentypen können nur dann übertragen werden, wenn Sie die VCL-Bibliotheken speziell darauf vorbereiten.
- Schließlich muss zur Registrierung des Interfaces die Methode *InvRegistry.RegisterInterface* aufgerufen werden.

Als Nächstes wenden wir uns der Implementierungs-Unit zu, zunächst jedoch nicht in der endgültigen Version, sondern in einer Test-Version, in der die Implementation der Methode *GetNextDate* zunächst auf die Rückgabe eines Test-Strings beschränkt wird, damit wir die Funktionsweise der SOAP-Infrastruktur testen können, bevor sich durch den Einbau der Datenbank-Funktionalität weitere Fehlermöglichkeiten ergeben. Die Unit *GetNextDateImpl* bindet die oben gezeigte Unit *GetNextDateIntf* ein und enthält eine von *TInvokableClass* abgeleitete Klasse, die das neue Interface implementiert:

```
unit GetNextDateImpl;

interface

uses
  InvokeRegistry, GetNextDateIntf;

type
  TGetNextDate = class(TInvokableClass, IGetNextDate)
     function GetNextDate(BaseDate: String): String; stdcall;
  end;

implementation

{ TGetNextDate }

function TGetNextDate.GetNextDate(BaseDate: String): String;
begin
  Result := 'Hello SOAP-Client!';
end;

initialization
  InvRegistry.RegisterInvokableClass(TGetNextDate);
end.
```

Die Ableitung der Klasse von *TInvokableClass* ermöglicht es dem SOAP-Framework, die Objekte dieser Klasse automatisch zu erzeugen, wenn sie von einem SOAP-Client angefordert werden. Sie können auch auf *TInvokableClass* verzichten (und Ihre Klasse beispielsweise von *TInterfacedObject* ableiten), müssen dann aber eine Factory-Funktion bereitstellen und registrieren, welche die Erzeugung neuer Objekte übernimmt (siehe hierzu die Online-Hilfe).

Installation und Test des Web Services

Bevor wir zum Einbau der eigentlichen Funktion in den Beispiel-Server kommen, geht es im Folgenden um den Test des Programmgerüsts, denn schon in diesem Gerüst gibt es verschiedene Fehlerquellen, die in schwer interpretierbaren Fehlermeldungen resultieren können, welche nicht fälschlicherweise dem später hinzukommenden Datenbank-Teil der Anwendung zugerechnet werden sollen.

Zunächst zur Installation des Web Services auf dem lokalen PC. Die mit Hilfe der SOAP-SERVER-ANWENDUNG aus der Objektgalerie erzeugte, um die beiden gezeigten Units erweiterte Anwendung wird kompiliert und dann so installiert, wie das für die Anwendungen in Kapitel 8.8.2 bereits ausführlich beschrieben wurde:

- Falls die Anwendung die Form eines Apache-Moduls hat, wird sie in das *Modules*-Verzeichnis kopiert und die Apache-Konfigurationsdatei wird um einen *LoadModule*-Abschnitt erweitert. Darin wird auch der Name des Moduls genannt, wie er in der *exports*-Klausel der Projektdatei festgelegt ist (siehe Kapitel 8.8.2). Sie können die Erreichbarkeit des SOAP-Servers schon ohne einen SOAP-Client in einem Web-Browser testen, der XML-Dateien anzeigen kann (wie dem Internet Explorer). Wenn Sie den Server in der Apache-Konfigurationsdatei etwa für die Pfadangabe *GetNextDate* registriert haben, erreichen Sie Delphis SOAP-Framework unter der Adresse `localhost/GetNextDate/soap/`, das selbst definierte Interface ist unter `localhost/GetNextDate/soap/IGetNextDate` zu erreichen. In beiden Fällen wird zwar eine Fehlermeldung geliefert, weil der Internet-Explorer eben kein SOAP-Client ist, aber zum Test ist dies dennoch hilfreich (siehe Abbildung 8.36).
- Auch im Falle einer Web-Debugger-Anwendung lässt sich der in Kapitel 8.8.2 beschriebene Ablauf auf einen Web Service übertragen. Je nach Namen der ausführbaren Anwendung und des darin enthaltenen COM-Objekts ist die Anwendung dann etwa unter der Adresse `localhost:1024/GetNextDateDebug.GetNextDateDebug/soap/` zu erreichen.

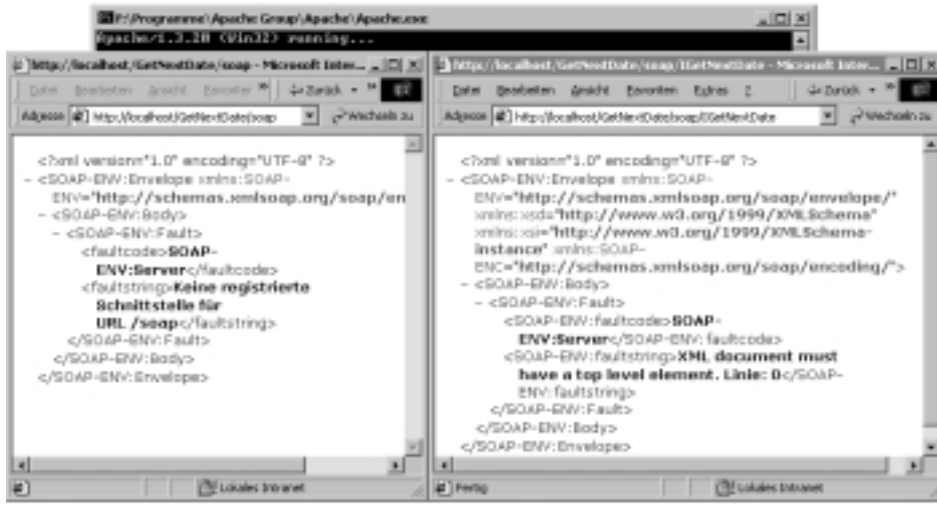


Abbildung 8.36: Test der Erreichbarkeit eines SOAP-Servers mit dem Internet Explorer. Die erste Fehlermeldung beweist, dass die Delphi-SOAP-Anwendung erreichbar ist, die zweite zeigt, dass das selbst definierte Interface zur Verfügung steht. Jetzt muss beides nur noch durch einen echten SOAP-Client angesprochen werden.

Ein Web-Service-Client

Wenn die Anwendung korrekt installiert ist, kann ein kleiner in Delphi programmierter SOAP-Client schnell ihre vollständige Funktion testen. Eine SOAP-Client-Anwendung erfordert kein spezielles Gerüst, Sie können also im Prinzip jede beliebige Anwendung zu einem SOAP-Client machen. Um das von Borland bereitgestellte Framework zu nutzen, fügen Sie eine *HTTPRIO*-Komponente von der Seite WEBSERVICES der Komponentenpalette in ein Formular ein (siehe Abbildung 8.37 für das Beispielprogramm *GetNextDateClient*).

Das *URL*-Property dieser Komponente stellen Sie sodann auf die URL des Web Services ein, beispielsweise `http://localhost/GetNextDate/soap` bei einer lokalen Apache-Installation wie der oben beschriebenen (das Anhängen von `/IGetNextDate` an diese URL erfolgt automatisch). Im Beispielprogramm wird das *URL*-Property zur Laufzeit gesetzt, wenn Sie über die Radioschaltergruppe einstellen, wo der Web Service installiert wurde.

Der Aufruf des Web Services gestaltet sich nun äußerst simpel: Sie binden die Unit ein, in der das Interface des Dienstes definiert ist, erhalten über den *as*-Operator ein solches Interface von der *HTTPRIO*-Komponente und rufen dann über dieses Interface die Methoden des Dienstes auf:

```
// uses GetNextDateIntf;

procedure TForm1.Button1Click(Sender: TObject);
var
  Intf: IGetNextDate;
begin
  Intf := HTTPRIO1 as IGetNextDate;
  Caption := Intf.GetNextDate(edtBaseDate.Text);
end;
```

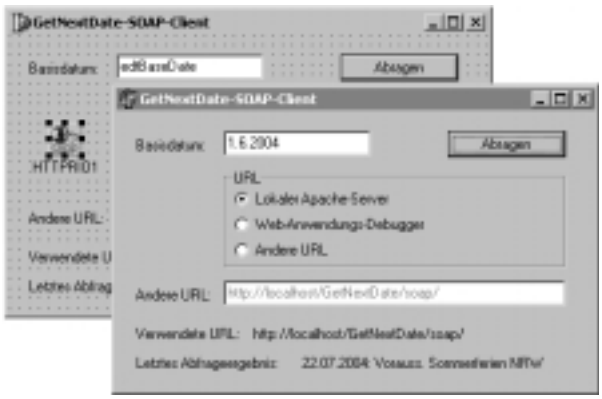


Abbildung 8.37: Das Formular des SOAP-Clients zum Test des GetNextDate-Services zur Entwurfszeit und zur Laufzeit

Hintergrund: So einfach diese beiden Anweisungen aussehen, intern muss *HTTPRIO* hier einiges an Arbeit verrichten, denn es gibt ja innerhalb der Client-Anwendung gar kein Objekt, welches das *IGetNextDate*-Interface implementiert und welches der Interface-Variable *Intf* zugewiesen werden könnte. *HTTPRIO* muss also schnell und »heimlich« die Laufzeit-Typinformationen von *IGetNextDate* untersuchen und ein Objekt konstruieren, dessen Tabelle virtueller Methoden (VMT) diesem Interface entspricht (dies geschieht bei der Anwendung des *as*-Operators, bzw. in der intern durch diesen aufgerufenen Methode *THTTPRIO.QueryInterface*).

Der folgende Aufruf von *Intf.GetNextDate* erfordert (wie jeder Aufruf von normalen Interface-Methoden auch) eine solche VMT. Wenn nun *Intf.GetNextDate* aufgerufen wird, geht die Arbeit für *HTTPRIO* weiter: Weil nämlich die Methoden des SOAP-Servers nicht direkt in der Client-Anwendung verfügbar sind, musste *HTTPRIO* die VMT so konstruieren, dass alle Aufrufe dieser Methoden an *HTTPRIO* selbst geleitet werden[27]. Um sie zu bearbeiten, muss *HTTPRIO* nun eine SOAP-Botschaft konstruieren, bei der wieder die Laufzeit-Typinformationen des Interfaces benötigt werden, um Namen und Typen von Methode und Parametern ermitteln und in Textform in die SOAP-Botschaft schreiben zu können. Die Botschaft wird dann über HTTP an den Server gesendet; *HTTPRIO* wartet dann auf die SOAP-Antwort und dekodiert sie, bevor der Aufruf von *Intf.GetNextDate* beendet wird.

Dass all dies hinter den Kulissen stattfindet und für uns Delphi-Nutzer wie ein trivialer Methoden-Aufruf aussieht, ist eines der faszinierendsten Merkmale der SOAP-Implementation von Borland (allerdings ist dies nicht vollkommen neu, denn auch die in Kapitel 8.7.4 beschriebenen Aufrufe entfernter DCOM-Objekte sehen genauso einfach aus, obwohl intern ähnliche Verwaltungsarbeit geleistet werden muss – allerdings in diesem Fall zum großen Teil von den Microsoft-DCOM-Bibliotheken).

Exkurs zur Verwendung von Clients und Servern, die nicht mit Delphi geschrieben sind: Um in einer Delphi-Anwendung auf einen beliebigen Server zuzugreifen, dessen Schnittstelle nicht in einer Delphi-Unit (wie im Beispiel die Unit *GetNextDateIntf*) definiert ist, müssen Sie die *HTTPRIO*-Komponente mit anderen Informationen versorgen: Statt des *URL*-Properties setzen Sie die Properties *WSDLLocation* (URL einer WSDL-Datei zur Beschreibung des Dienstes), *Service* und *Port* (WSDL-Dokumente definieren *Services*, die aus einem oder mehreren *Ports* bestehen. Hierbei entspricht ein Port einem Object-Pascal-Interface.) *WSDLLocation*, *Service* und *Port* werden üblicherweise in der Dokumentation des zu nutzenden Dienstes angegeben (siehe etwa die Dienstliste auf `www.xmethods.com`).

27 Hier sei *HTTPRIO* stellvertretend für den Teil des SOAP-Frameworks genannt, der diese Arbeit tatsächlich ausführt. Da Borland die Unit *rio.pas* nicht im Quelltext mitliefert, lässt sich dies von außen schwer untersuchen.

Wenn Sie einen Delphi-SOAP-Server auch für Nicht-Delphi-Clients zur Verfügung stellen wollen, so benötigen diese eine Dienst-Beschreibung in Form eines WSDL-Dokuments. Eine Delphi-Web-Service-Anwendung ist automatisch dazu in der Lage, solche WSDL-Dokumente für alle über die *InvRegistry* registrierten Schnittstellen zu erzeugen, und zwar dank der *WSDLHTMLPublish*-Komponente, die automatisch im Anwendungsgerüst eingefügt wird (siehe Abbildung 8.35). Das Property *WsdlHtmlPublish.WebDispatch.PathInfo* legt fest, unter welcher Pfadinformation das WSDL-Dokument abgerufen werden kann, per Voreinstellung ist das die Pfadinformation *wsdl*. Im vorliegenden Beispiel ist das Dokument somit unter `http://localhost/GetNextDate/wsdl/IGetNextDate` zu erreichen. Wenn Sie die Angabe von `IGetNextDate` weglassen, gelangen Sie statt zum WSDL-Dokument zu einer HTML-Übersichtsseite, die alle von dieser Web-Service-Anwendung abrufbaren WSDL-Dokumente auflistet.

Einbau der Datenbank-Funktionalität *R189*

Nun können wir zum Abschluss die Datenbank-Funktionalität in das Gerüst einbauen. Da die Funktion *GetNextDate*, welche die eigentliche Arbeit erledigt, schon in der Web-Broker-Anwendung von 8.8.2 verwendet und dort auch abgedruckt wurde (siehe Seite 1210), müssen wir uns hier nur noch um die »Verdrahtung« der Funktion mit dem Web-Service-Gerüst kümmern. Einzige besondere Frage dabei ist, wo wir die Datenzugriffskomponenten unterbringen sollen.

In Kapitel 8.8.2 wurde die Datenbank-Funktionalität noch aus der Methode *TWebModule1.PageProducer1HTMLTag* gestartet, die Teil eines Webmoduls war. So war es ein Leichtes, die Datenzugriffskomponenten in diesem Webmodul unterzubringen.

Im aktuellen Fall geht es jedoch um die Funktion *TGetNextDate.GetNextDate*, die nicht Teil eines Daten- oder Webmoduls ist. Zwar gibt es in unserem Projekt auch ein Webmodul (dies ist bereits Teil der automatisch erzeugten SOAP-SERVER-ANWENDUNG), aber zur Laufzeit wird für jede eintreffende Anfrage eine eigene Kopie dieses Webmoduls erzeugt, so dass unter Umständen mehrere Kopien gleichzeitig existieren[28]. Die Frage ist also, wie wir in der Funktion *GetNextDate* das für die aktuelle Anfrage passende Webmodul ermitteln.

Die Lösung liegt in der Unterscheidung der Threads: Für jede eintreffende Anfrage wird nämlich nicht nur ein eigenes Webmodul verwendet, sondern auch automatisch ein eigener Thread gestartet. Jede Anfrage wir also komplett in einem eigenen Thread

28 Grund dafür ist, dass es sich bei dem Beispiel um ein DLL-Projekt handelt. Wenn die Anwendung als CGI-Anwendung ausgelegt wäre, würde für jede Anfrage eine Kopie der gesamten Anwendung gestartet, so dass in jeder Anwendung nur ein einziges Webmodul existieren würde.

bearbeitet, und pro Thread kann es immer nur ein Webmodul geben. Wir können nun beim Aufruf des Webmoduls diese Instanz des Moduls in einer Thread-Variablen speichern:

```
threadvar
  ThreadWebModule: TWebModule1;

procedure TWebModule1.WebModuleBeforeDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  ThreadWebModule := self;
end;
```

Von einer als *threadvar* deklarierten Variable existiert für jeden Thread eine eigene Kopie. Wenn also diese Variable später in unserer *GetNextDate*-Methode abgefragt wird, so enthält sie einen Verweis auf das Webmodul, das im selben Thread erzeugt wurde:

```
function TGetNextDate.GetNextDate(BaseDate: String): String;
begin
  Result := SharedDBCode.GetNextDate(ThreadWebModule.cdsTermine,
              BaseDate, False);
end;
```

Standardmäßig nimmt die VCL übrigens eine »Zwischenspeicherung« von Webmodulen vor. Das bedeutet, dass ein Webmodul nach der Bearbeitung einer Anfrage nicht gleich freigegeben, sondern eine Zeit lang im Speicher gehalten wird, damit es eventuell eine weitere Anfrage bearbeiten kann. Auch wenn ein Webmodul nach Abarbeitung einer Anfrage weiter existiert, wird der Thread, in dem die Abarbeitung stattfand, beendet. Für eine neue Anfrage wird also evtl. das gleiche Webmodul, immer aber ein neuer Thread verwendet. Und jeder neue Thread verfügt über einen eigenen Satz von Thread-Variablen. Es ist daher wichtig, dass die Thread-Variable *ThreadWebModule* nicht nur einmal bei der ersten Initialisierung des Web-Moduls im *OnCreate*-Ereignis gesetzt wird, sondern in jedem neuen Thread, also bei jeder neuen Anfrage. Das passende Ereignis dazu heißt *TWebModule.BeforeDispatch*.

Hinweis zur BDE: Das gezeigte Beispiel verwendet dbExpress für den Datenzugriff. Wenn Sie statt dessen die BDE verwenden, erreichen Sie die Trennung der Datenzugriffe der einzelnen Threads, indem Sie die *TSession*-Komponente verwenden. Setzen Sie deren *AutoSessionName*-Property auf *True*, damit jede Session automatisch mit einem eindeutigen Namen versehen wird.

Hinweis zu WebSnap: In Delphi 6 gibt es auch eine neue Klasse *TWebContext*, von deren Funktion *FindModuleClass* Sie ebenfalls das für den aktuellen Thread gültige Webmodul erfragen können. Diese Klasse ist jedoch Teil der WebSnap-Funktionalität und steht daher in einer Nicht-WebSnap-Anwendung wie der unsrigen nicht zur Verfügung. Ein Beispiel für die Verwendung des globalen *WebContext*-Objekts finden Sie in jeder Unit eines WebSnap-Seitenmoduls. Die globale Variable, die sonst von Delphi für jedes Formular, jedes Datenmodul und jedes Webmodul angelegt wird (z.B. *Form1: TForm1*) wird hier durch eine Funktion ersetzt, in der das Webmodul des aktuellen Threads abgefragt wird.

Für das Ereignis *TWebModule.OnCreate* bleibt dennoch eine kleine Aufgabe, und zwar die Aktivierung der Datenbankzugriffs-Komponenten:

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
begin
  SQLConnection1.Connected := True;
  cdsTermine.Open;
end;
```

Aufgrund der erwähnten Zwischenspeicherung von Webmodulen hält eine solche Initialisierung eventuell für mehrere Anfragen vor.

Alternative Thread-Architektur

Die oben beschriebene Vorgehensweise verdeutlicht gut, wie sich mehrere Threads konfliktfrei nebeneinander betreiben lassen, indem jeder Thread mit einer eigenen Kopie der relevanten Daten arbeitet. Im Beispiel gibt es für jeden Thread eine Kopie der Variable *ThreadWebModule*, eine automatisch von der VCL erzeugte Kopie des Webmoduls und dabei auch eine automatisch miterzeugte Kopie aller in diesem Modul enthaltenen Komponenten, also auch der Datenzugriffskomponenten. Jedoch sind auch Nachteile zu bedenken, wenn Datenzugriffskomponenten einfach so dupliziert werden. So nimmt etwa jede *SQLConnection*-Komponente im Beispiel eine eigene Verbindung zu einer Interbase-Datenbank auf, aber die Zahl gleichzeitiger Verbindungen kann je nach Lizenz engen Grenzen unterworfen sein. Auch der Verwaltungsaufwand für das Anlegen der Kopien kann je nach Komponente und Anwendungsfall unverhältnismäßig groß sein. So wird es also häufig von Vorteil sein, wenn alle Threads einer Web-Server-Anwendung mit nur einer Kopie der Datenzugriffskomponenten arbeiten.

Um dies zu erreichen, müssen die Datenzugriffskomponenten von dem automatisch vervielfältigten Webmodul ausgelagert werden; es bietet sich an, hierfür ein separates Datenmodul zu verwenden. Wenn Sie ein solches Datenmodul anlegen, müssen Sie jedoch im Projekt-Quelltext den automatisch von Delphi hinzugefügten *CreateForm*-

Aufruf entfernen, weil es sonst beim Start der Anwendung zu einem Fehler kommt (in einer WebBroker-Anwendung ist nur ein mit *CreateForm* erzeugtes Daten- oder Webmodul erlaubt):

```
begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
// Diesen Aufruf wirkungslos machen:
//  Application.CreateForm(TDataModule1, DataModule1);
  Application.Run;
end.
```

Statt dessen muss das Datenmodul selbst erzeugt werden, was beispielsweise im Initialisierungsteil der Unit geschehen kann:

```
initialization
  DataModule1 := TDataModule1.Create(nil);
finalization
  DataModule1.Free;
end.
```

Wenn jedoch mehrere Threads gleichzeitig auf ein Objekt zugreifen können, müssen Sie aufpassen, dass die Threads sich nicht gegenseitig in die Quere kommen. Am sichersten wäre es im vorliegenden Beispiel, den gesamten Datenbankzugriff zu einem kritischen Code-Abschnitt zu machen, der immer nur von einem Thread gleichzeitig bearbeitet werden kann:

```
// Benötigt: uses SyncObjs;
// Initialisierung: CriticalSection := TCriticalSection.Create;
function TGetNextDate.GetNextDate(BaseDate: String): String;
begin
  CriticalSection.Enter;
  Result := SharedDBCode.GetNextDate(ThreadWebModule.cdsTermine,
              BaseDate, False);
  CriticalSection.Leave;
end;
```

Das Objekt *CriticalSection* könnte z.B. wie *DataModule1* im Initialisierungsteil der Unit erzeugt werden. Weitere Informationen und ein ausführlicheres Beispiel zu TCriticalSection finden Sie in Kapitel 4.7.3, *R*125 auf Seite 579.

Arrays und eigene Typen

Unsere bisherigen praktischen Versuche mit dem Datenmodul lassen erahnen, dass Web Services ähnlich leistungsfähige (nicht-visuelle) Funktionen zur Verfügung stellen können wie normale Delphi-Anwendungen. Ein wesentlicher, bisher nicht beachteter Faktor liegt jedoch in den Parametertypen, die in den Methoden des Dienstes verwendet werden können. Wie erwähnt können nur einfache Typen ohne weiteres Zutun automatisch übertragen werden. Die ausschließliche Verwendung einfacher Typen

wäre jedoch eine große Beschränkung für Web Services, weshalb Ihnen Delphis Bibliotheken die Möglichkeit bieten, komplexe Typen zur Übertragung in SOAP-Botschaften zu registrieren.

Im Folgenden werden ein dynamisches Array und eine Klasse registriert – die Details, auf die es dabei besonders ankommt, sind fett hervorgehoben:

```
type
  // Der Name "FieldContent" soll im Folgenden
  // ohne Bedeutung sein.
  TFieldContentClass = class(TRemotable)
  private
    FData: Integer;
  public
    constructor Create(aContent: TFieldContent);
    function GetFieldContent: TFieldContent;
  published
    property Data: Integer read FData write FData;
    // hier weitere Datenelemente
  end;
  TFieldContentArray = array of TFieldContentClass;

implementation

{ TFieldContentClass }
... // Anwendungsabhängige Klassen-Implementation

initialization
  RemClassRegistry.RegisterXSClass(TFieldContentClass, '',
     'fieldContentClass');
  RemClassRegistry.RegisterXSInfo(TypeInfo(TFieldContentArray), '',
     'fieldContentArray');
end.
```

Die Klasse muss also von *TRemotable* abgeleitet werden und alle Daten, die mit SOAP übertragen werden sollen, müssen als Property im *published*-Bereich der Klasse deklariert sein. Für die Registrierung ist das vordefinierte Objekt *RemClassRegistry* aus der bereits bekannten Unit *InvokeRegistry* zuständig. Die Parameterlisten der Funktionen zur Registrierung von Klassen (*RegisterXSClass*) und dynamischen Arrays (*RegisterXSInfo*) gleichen sich (weitere optionale Parameter sind in der Online-Hilfe dokumentiert):

- Die Angabe, was registriert werden soll, findet im ersten Parameter statt und ist entweder die Klasse selbst oder ein *TypeInfo*-Zeiger auf den zu registrierenden Typen (mit *RegisterXSInfo* können Sie außer dynamischen Arrays übrigens auch noch Aufzählungstypen und boolesche Typen registrieren).

- Der zweite Parameter betrifft dass hier nicht weiter interessante XML-Detail des Namespace URIs – der obige Code nimmt die Option, diesen automatisch generieren zu lassen, dankend an, indem er nur einen leeren String übergibt.
- Im dritten Parameter folgt der Name, unter dem der Typ in den XML-Dokumenten (SOAP-Nachrichten und WSDL-Dokumente) in Erscheinung treten soll.

Die Registrierung einer Klasse wie der oben gezeigten ist übrigens die einzige Möglichkeit, Record-artige Datenstrukturen zu registrieren. Wenn Sie also einen Record übertragen wollen, müssen Sie dafür eine Klasse definieren, in der jedes Record-Element über ein Property angesprochen werden kann, das im *published*-Bereich der Klasse deklariert ist (hier ist wieder einmal der Typinformations-Hunger der SOAP-Bibliotheken der Grund – für *published*-Properties legt der Compiler schon seit Delphi 1 besondere Typinformationen an).

Wenn es nur um einen einzelnen Record geht, lässt sich die Registrierung des Records natürlich vermeiden, indem jedes Record-Element als einzelner Parameter übertragen wird. Sollen jedoch mehrere Records übertragen und hierfür ein dynamisches Array verwendet werden, dann führt kein effektiver Weg mehr an der Klassen-Registrierung vorbei.

Mit Arrays lassen sich übrigens die Grenzen von SOAP ausloten, was die Geschwindigkeit betrifft. Schon ein Array aus wenigen hundert Integer-Werten, wie es etwa von folgender Funktion zurückgeliefert werden könnte:

```
function GetField: TFieldContentArray; stdcall;
```

kann mehrere Sekunden Rechenzeit benötigen, sowohl für die Codierung des Datenpakets in ein XML-Dokument auf der Seite des Servers als auch für die Dekodierung des Dokuments im Client. Damit dürfte SOAP zumindest für die Implementation von Action-Spielen schlecht geeignet sein.

Für weitere Details zu den einzelnen Klassen und Methoden muss hier auf die Online-Hilfe verwiesen werden. Sicherlich wird es nach Erscheinen dieses Buchs noch viele aktuelle Artikel oder Bücher zum Thema »Web Services« geben und es könnte interessant sein, diese Entwicklung zu verfolgen: sowohl allgemein, was die generelle Nutzung dieser Dienste betrifft, als auch speziell, was die Borland-Implementation angeht – insbesondere, wenn Borland wie angekündigt die Web-Service-Unterstützung nach Linux portiert und dort unter Kylix zur Verfügung stellt.

A Erweiterung der Delphi-IDE

Das OpenTools-API ist wohl nicht nur für Bastler-Naturen eines der faszinierendsten Merkmale von Delphi, denn es gibt dem Entwickler Gelegenheit, in interne Abläufe der Delphi-IDE einzusehen, einzugreifen und die IDE nahezu grenzenlos um neue Funktionen zu erweitern. Die von Borland bereitgestellte Dokumentation für dieses API besteht im Wesentlichen aus den Kommentaren in den Units, die sich ab den Professional-Ausgaben im Verzeichnis SOURCE\TOOLSAPI befinden.

Experten und Wizards

Nachdem die IDE-Erweiterungen in den frühen Delphi-Versionen noch schlicht als »Experten« bezeichnet wurden, taucht in der Programmierschnittstelle seit Delphi 4 auch der Begriff »Wizard« auf. Er ist darin begründet, dass viele Delphi-Erweiterungen von ihrem Äußeren und in der Bedienung dem von Microsoft geschaffenen Wizard-Standard entsprechen – so beispielsweise der mit Delphi mitgelieferte Datenbankformular-Experte (in der englischen Version als »Wizard«, in der deutschen als »Experte« bezeichnet).

Dies ändert jedoch nichts daran, dass Sie Delphi auf eine erheblich vielseitigere Weise erweitern können als nur durch Dialoge, die sich nach dem Wizard-Standard richten. Ein Wizard besteht lediglich aus einem Dialog, in dem Sie mit zwei Schaltern nach vorne und nach hinten blättern können, verschiedene Seiten mit Angaben ausfüllen, dies alles unter ausführlicher Anleitung durch Text und Bild, und in dem Sie am Schluss auf einen *Fertigstellen*-Schalter drücken, um den »Zaubervorgang« zu starten (offenbar stellt sich Microsoft die Arbeit von Zauberern als sehr geordnet vor). Eine Delphi-IDE-Erweiterung kann jedoch auch eine ganz andere Benutzerschnittstelle haben oder sogar ganz ohne Dialog arbeiten.

Eine in einer Klasse abgeschlossene Erweiterung der Delphi-IDE wird im Folgenden auch als »Experte« bezeichnet. Dieser Expertenbegriff rührt daher, dass eine solche Erweiterungsklasse schon seit Delphi 1 von der Klasse *TIExpert* abgeleitet werden kann. Seit Delphi 4 lassen sich Experten auch von einer Klasse namens *IOTAWizard* ableiten. Da die in diesem Kapitel vorgestellten Erweiterungen jedoch keine Wizard-Dialoge verwenden, wird im Folgenden auf den Begriff des »Wizards« verzichtet.

Die IDE-Erweiterungen auf der CD

Dieses Kapitel soll einen Überblick über das ToolsAPI geben und dann anhand einiger Beispiel-Experten einige Bereiche des ToolsAPI genauer untersuchen. Die wichtigsten Funktionen der Beispielexperten sind:

- Der *ToolsAPI-Erforscher* gibt Informationen über das aktuelle Projekt, die geladenen Module und die geöffneten Editorfenster. Er listet die Namen von Delphis Menüpunkten auf und demonstriert die Schnittstelle, mit der Sie den Inhalt der Editorfenster auslesen und verändern können.
- Der *Formular-TreeView* zeigt Ihnen eine hierarchische Darstellung der Eltern-Kind-Hierarchie der Komponenten des aktuellen Formulars. In Delphi-Versionen, die noch nicht wie Delphi 6 über das Objekthierarchie-Fenster verfügen, kann dies hilfreich sein, wenn Komponenten vollständig durch Kindelemente verdeckt sind und nicht durch Mausklicks angesprochen werden können (durch den TreeView könnten Sie etwa den Namen einer verdeckten Komponente erfahren, um sie dann gezielt aus der aufklappbaren Liste des Objektinspektors auszuwählen).
- Der *Editor-Assistent* bietet zwei Funktionen für den aktiven Delphi-Editor: Mit Shift+Strg+H springen Sie zur Deklaration des an der Eingabeposition befindlichen Bezeichners und mit Shift+Strg+D deklarieren Sie das Wort an der Eingabeposition als Variable.
- Eine Nachbildung des *CodeExplorers* der Delphi-IDE erlaubt Ihnen, Methoden und andere Klassenelemente gezielt per Auswahl aus einem TreeView anzusteuern. Zwar bietet die Nachbildung weniger Konfigurationsmöglichkeiten als der Code-Explorer von Delphi und ist auch nicht andockbar, dafür läuft sie aber auch mit den Personal/Standard-Ausgaben von Delphi.
- Der *Formular-Assistent* erzeugt eine neue Steuerelementgruppe im aktuellen Formular, und zwar bestehend aus Eingabeelementen, die zu den Typen der Variablen einer Record- oder Klassenstruktur der aktuellen Unit passen. Vor der Erzeugung erhalten Sie in einem Dialog Gelegenheit, nicht benötigte Eingabeelemente abzuwählen und Beschriftungen festzulegen.

Um die Erweiterungen unter Delphi 6 zu installieren, laden Sie die Packages `ToolsApiExperten.dpk` (für die ersten beiden Funktionen aus obiger Liste) und `IteaExperten.dpk` (für alle weiteren) aus dem Verzeichnis EXPERTS in den Package-Editor und drücken jeweils den INSTALLIEREN-Schalter. Die ersten Funktionen der *ToolsApiExperten* finden Sie im Menü HILFE, die der *IteaExperten* am Ende des Menüs EDITOR.

Für Delphi 4 und 5 liegen alle Erweiterungen in einem gemeinsamen Package namens *VierExperten* vor, das entsprechend in den Package-Editor geladen und von dort installiert werden kann.

Die folgende Tabelle zeigt die wichtigsten Units der Packages, also die Units, in denen die Experten/Wizard-Objekte definiert werden, und die Units, welche die zugehörigen Formulare implementieren:

Unit	Registrierungsobjekt	Formular-Units
ToolsApiExplorer	TWizard1/TExpert1	ToolsApiExplorer
FormExplorer	TFormExplorer (TIExpert)	FormExplorer
EditorAssistants	TEditorAssistants (TIExpert)	VarDeclDlg, IteaCodeExplorer, ErrorWin (nur für den CodeExplorer)
FormAssistant	TFormAssistant (IOTAWizard)	MaskSpecDlg, TestFormAssistant (nur zu Testzwecken)

Zwei grundlegende Units des Packages, die in fast allen anderen Units verwendet werden, sind *ToolUtil*, in der einige mit dem OpenTools-API zusammenhängende Hilfsfunktionen versammelt sind, und *Itea*, die als Import-Unit für die Itéa-DLL fungiert und so die Funktionalität zum Parsen der Object-Pascal-Units zugänglich macht.

Eine detaillierte Beschreibung der Experten, insbesondere der Itéa-Experten, würde sehr viel Platz beanspruchen, daher konzentrieren sich die folgenden Abschnitte vorwiegend auf Code-Auszüge, in denen wichtige ToolsAPI-Grundlagen demonstriert werden. Viele Funktionen der als Beispiel dienenden Experten können dagegen nur sehr grob erläutert werden, mehr erfahren Sie durch den vollständigen Quelltext auf der CD.

A.1 Grundlagen

Anhand des ersten Beispielexperten soll dieser Abschnitt zunächst die Grundlagen der Expertenprogrammierung und des ToolsAPI vorstellen. Ziel des *ToolsApi-Explorers* ist, einige der OpenTools-Schnittstellen am Bildschirm zu visualisieren bzw. Ihnen zu erlauben, sie interaktiv auszuprobieren (im Falle der Editorschnittstelle). Der ToolsApi-Explorer dient damit auch als Vorbereitung für die im weiteren Verlauf vorgestellten Add-In-Experten.

Der Weg in die Delphi-IDE

In Kapitel 6 wurde gezeigt, wie Sie mit einer *Register*-Prozedur selbst geschriebene Komponenten, Property- und Komponenten-Editoren in der Delphi-IDE registrieren können. Statt dieser Objekte (oder zusätzlich zu diesen) können Sie in dieser Prozedur auch Experten installieren:

```
procedure Register;
begin
  // Registrierung über das Experten-Interface:
  RegisterLibraryExpert(TToolsApiExplorerE.Create);
  // Oder: Registrierung über das Wizard-Interface (nur ab Delphi 4):
  RegisterPackageWizard(TToolsApiExplorerW.Create as IOTAWizard);
end;
```

Wenn Ihre Unit Teil eines Packages ist (also im *Contains*-Abschnitt des Package-Editors aufgeführt ist) und Sie das Package unter Delphi installiert haben (über KOMPONENTE | PACKAGES INSTALLIEREN oder den entsprechenden Schalter im Package-Editor), ruft Delphi die *Register*-Prozedur bei jedem Start der IDE automatisch auf. Da sich ein Experten-Package (wie jedes andere Package auch) zur Laufzeit mit der Delphi-IDE in einem gemeinsamen Adressraum befindet, steht nach dieser Registrierung einer ausgiebigen Kommunikation zwischen Delphi und dem Experten nichts im Wege.

IDE-Erweiterungen in DLLs

Bezüglich des binären Formats sind zwei Arten von IDE-Erweiterungen zu unterscheiden. Während die bisher erwähnten Experten in Packages nur von Borland-Systemen erzeugt werden können, die Packages kennen, steht das zweite Format des Experten allen Entwicklungssystemen offen, die DLLs erzeugen können. Ein in einer DLL vorliegender Experte wird nicht über das Menüsystem von Delphi (KOMPONENTE | PACKAGES INSTALLIEREN) installiert, sondern über einen Eintrag in der Registry (Delphi 1: `delphi.ini`-Datei). So finden Sie beispielsweise für Delphi 6 ab der Professional-Version im Knoten `Software\Borland\Delphi\6.0\Experts` bereits eine Experten-DLL eingetragen, die bei jedem Start von Delphi geladen wird. Der Demo-Experte sorgt für zwei Einträge in der Objektgalerie von Delphi: für den Anwendungs-Experten auf der Seite *Projekte* und für den Dialog-Experten auf der Seite *Dialoge*. Bis Delphi 5 können Sie übrigens den Quelltext beider Experten einsehen (`Delphi5\Demos\Experts\ExptDemo.dpr`); unter Delphi 6 fehlt dieser Quelltext – vielleicht ist er mittlerweile zu alt und harmoniert nicht mehr mit der von Borland vorgesehenen zukünftigen Entwicklung der ToolsAPI-Schnittstelle.

Bei Experten-DLLs gibt es bezüglich des Aufbaus der Kommunikation mit der Delphi-IDE einige formale Unterschiede zu Package-Experten zu beachten, die vom angesprochenen Beispielprojekt EXPTDEMO aufgezeigt werden und in diesem Kapitel nicht weiter untersucht werden sollen.

Virtuelle Methoden

Das Fundament der Kommunikation zwischen Delphi und seinen IDE-Erweiterungen bilden virtuelle Methoden. Sowohl Delphi als auch die Erweiterung stellen dem jeweils anderen Partner ihre Dienste in Form solcher virtuellen Methoden zur Verfügung:

- So muss zunächst die *TWizardKlasse* bzw. die *TExpertenKlasse* der Erweiterung von einer durch das ToolsAPI vorgegebenen Klasse abgeleitet werden, in der alle virtuellen Methoden definiert sind, die Delphi bei Experten und Wizards aufruft. Neben dem Wizard bzw. Experten gibt es noch ein paar andere Gelegenheiten, bei denen Sie der Delphi-IDE eine Klasse bzw. ein Objekt zur Verfügung stellen können, dessen Methoden Delphi aufrufen soll. Diese Objekte sind »Benachrichtigungs-Objekte«, im ToolsAPI als *Notifier* bezeichnet. Sie werden von Delphi beispielsweise benachrichtigt, wenn eine andere Seite des Quelltexteditors aufgeschlagen wurde.
- In umgekehrter Richtung stellt Delphi Ihnen eine Vielzahl von Schnittstellen und eine noch viel größere Menge von Methoden zur Verfügung, die Sie aus Ihrem Wizard bzw. Experten heraus aufrufen können, um z.B. Informationen über das aktuelle Projekt zu erhalten, den Editorinhalt zu modifizieren oder ein neues Formular automatisch zu erzeugen.

Interfaces

Da die Wurzeln des ToolsAPI bereits unter Delphi 1 zu finden sind, wo *Interfaces* noch nicht zum Sprachumfang von Object Pascal gehörten, finden Sie in den neueren Delphi-Versionen (ab Delphi 4) zwei Arten von Schnittstellendefinitionen:

- Die mit dem Schlüsselwort *interface* deklarierten Schnittstellen nach dem Component Object Model. Einer der Vorteile dieser Interfaces liegt in der automatischen Verwaltung des Referenzzählers durch den vom Compiler erzeugten Code. Wenn Sie die Delphi-IDE über Schnittstellenvariablen ansprechen, brauchen Sie sich also um die Freigabe der Objekte nicht zu kümmern, da schon der vom Compiler erzeugte Code die notwendigen Aufrufe von *_Release* enthält.
- Die zweite Art der Schnittstellendefinition sind abstrakte Basisklassen, in denen abstrakte virtuelle Methoden deklariert werden. Die abstrakten Klassen des ToolsAPI haben mit der Klasse *TInterface* eine gemeinsame Wurzel. *TInterface* ist quasi die Simulation eines echten *Interfaces*, denn sie deklariert die Methoden *AddRef* und *Release*, die auch jedes Interface (durch die Vererbung von *IUnknown*) unterstützen muss. Da *TInterface* aber kein echtes Interface ist, müssen Sie *Release* und gegebenenfalls *AddRef* selbst aufrufen.

Alte und neue Schnittstellen

Bis Delphi 3 befanden sich die Klassendeklarationen der Experten in verschiedenen Units des TOOLSAPI-Verzeichnisses: *VirtIntf*, *EditIntf*, *DsgnIntf*, *ExptIntf*, *FileIntf*, *IStreams*, *VCSIntf* und schließlich *ToolIntf*. Diese Units sind auch noch unter Delphi 6 verfügbar, werden dort jedoch durch die in Delphi 4 komplett neu hinzugekommene

(und seitdem stark erweiterte) Unit *ToolsAPI* ergänzt, die einen großen Teil der alten Units durch neue Schnittstellen unnötig macht und die ganz auf der Verwendung von Interfaces aufbaut.

Was die in diesem Anhang verwendete Funktionalität betrifft: Diese liegt zum größten Teil parallel einmal in abstrakten Basisklassen der Units *EditIntf*, *ExptIntf*, *ToolIntf* und ein weiteres Mal in Interfaces der Unit *ToolsAPI* vor. Damit die Beispielexperten auch zu älteren Delphi-Versionen kompatibel bleiben, verwenden sie größtenteils auch die älteren Units. Ganz mit den neuen *ToolsAPI*-Schnittstellen arbeitet lediglich der Formular-Assistent aus Abschnitt A.3. Da die Methoden zwischen alten und neuen Units jedoch überwiegend übereinstimmen, ist die Umstellung von alten auf neue Units und umgekehrt relativ unproblematisch.

OTA und NTA-Schnittstellen

Die *ToolsAPI*-Unit, die ja wie erwähnt keine Klassen, sondern nur echte *Interface*-Deklarationen enthält, trifft eine weitere Unterscheidung der Schnittstellen:

- Schnittstellen, die nur in einem Delphi-Package korrekt funktionieren. Diese sind durch das Präfix *INTA* gekennzeichnet, was so viel bedeutet wie »Interface des Native Tools API«. Der Grund für die Exklusivität liegt darin, dass die Methoden der NTA-Interfaces auch Referenzen auf VCL-Objekte zurückliefern können, wie beispielsweise eine *TMainMenu*-Komponente für das Delphi-Hauptmenü oder ein *TComponent*-Objekt für eine Komponente innerhalb eines Formulars. Die VCL-Klassen stellen ihre Funktionalität jedoch nicht über COM-Interfaces zur Verfügung und können daher nur von Programmiersprachen verwendet werden, die selbst über die VCL verfügen, wie etwa dem Borland C++Builder.
- Schnittstellen, die aus allen anderen Programmiersprachen genutzt werden können, beginnen mit dem Präfix *IOTA*, wobei das »OTA« für »Open Tools API« steht. Die OTA-Methoden liefern keine Object-Pascal-Objekte zurück, sondern nur COM-kompatible Interfaces, die innerhalb der Open-Tools-Units deklariert sind. So werden beispielsweise keine *TComponent*-Objekte zurückgeliefert, sondern nur *TIComponentInterface*-Schnittstellen, die quasi als Vermittler zu den tatsächlichen Object-Pascal-Objekten dienen.

Überblick über die Schnittstellen

Abbildung A.1 zeigt einen grafischen Überblick über die Interfaces der ToolsAPI-Unit. Die meisten Schnittstellen werden von der IDE selbst bereitgestellt und geben Ihnen Zugriff auf die Module des in der IDE geladenen Projekts, auf die damit verbundenen Editoren, auf die Inhalte dieser Editoren, auf die Objekte des Debuggers (Haltepunkte, Prozesse, Threads) und auf die Optionen von Umgebung und Projekt.

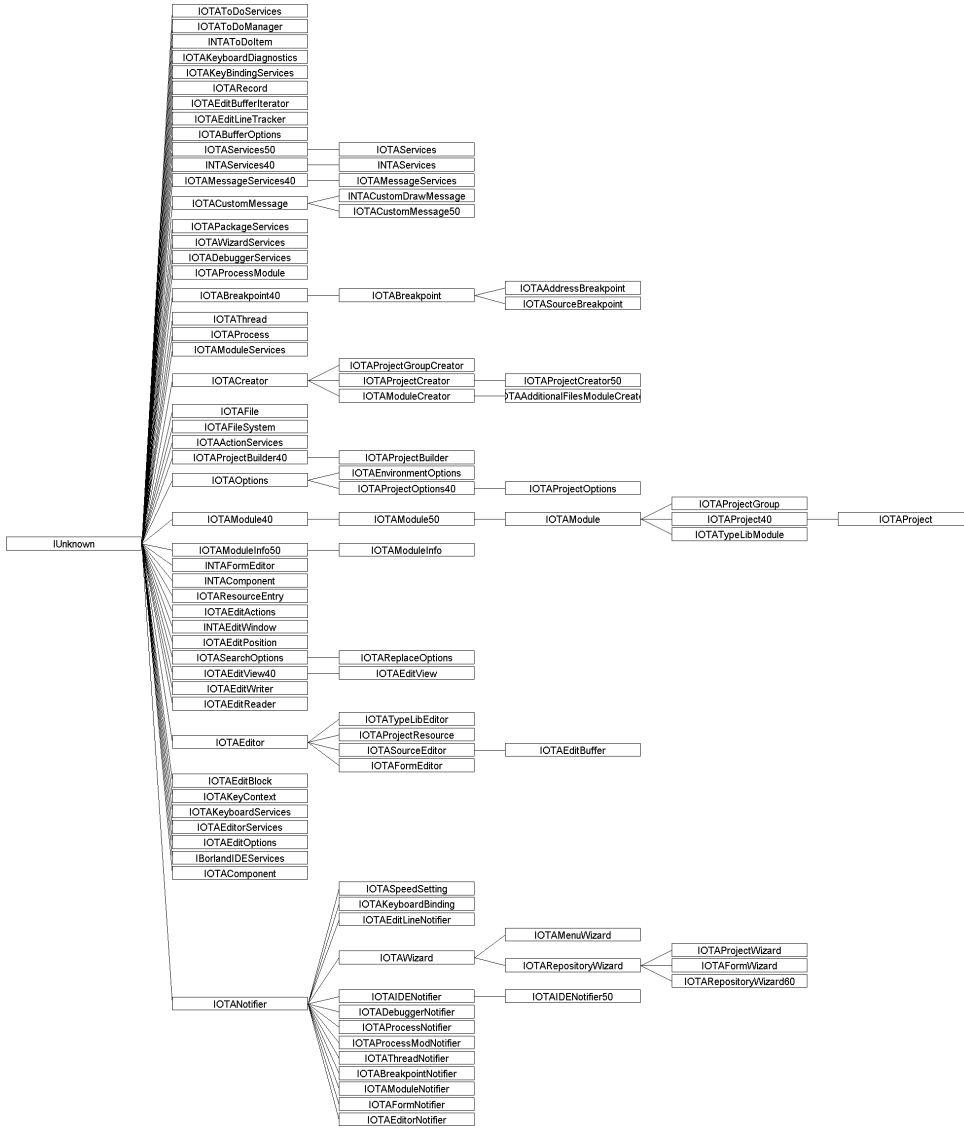


Abbildung A.1: Die Interface-Hierarchie der ToolsAPI-Unit

Im Gegensatz dazu gibt es auch einige Schnittstellen, die dazu da sind, von Experten und Wizards implementiert zu werden. So können Sie beispielsweise mit *IOTACustomMessage* neue Meldungsarten für das Meldungsfenster der Delphi-IDE definieren (das *IOTAMessageServices*-Interface besitzt die Methoden zum Hinzufügen solcher Nach-

richten in das Meldungsfenster). *IOTACreator* und *IOTAFile* sind Schnittstellen, die Sie implementieren können, um einen Wizard für die Objektablage zu schreiben, der neue Projekte oder Formulare generiert.

Sehr interessant sind auch die von *IOTANotifier* abgeleiteten Schnittstellen. Über sie benachrichtigt die Delphi-IDE ein Experten-Package von den verschiedensten Ereignissen innerhalb der IDE. So wird beispielsweise ein *IOTAEditorNotifier* informiert, wenn eine andere Seite im Editor aufgeschlagen wurde, und ein *IOTAProcessNotifier* erfährt vom Erzeugen und Löschen von Prozessen und Threads. Die Schnittstellen der Delphi-IDE enthalten zum Registrieren eines solchen Notifiers entsprechende *AddNotifier*-Methoden (so z.B. *IOTAEditor* für *IOTAEditorNotifier* und *IOTAProcess* für *IOTAProcessNotifier*).

Die Beispielexperten dieses Anhangs machen jedoch nur von den Schnittstellen im *IOTAWizard*-Teilbaum Gebrauch – wobei man sich fragen könnte, warum *IOTAWizard* überhaupt von *IOTANotifier* abgeleitet ist, werden doch laut Dokumentation drei der vier *IOTANotifier*-Methoden für einen *Wizard* niemals aufgerufen.

Die folgende Tabelle nennt die im Rest dieses Anhangs teilweise genauer untersuchten Schnittstellen sowohl aus dem alten als auch aus dem neuen »Schnittstellensortiment« und gibt jeweils an, welche Schnittstelle im jeweils anderen Sortiment die gleiche Aufgabe (und teilweise auch die gleichen Methoden) hat:

abwärtskompatible Klasse	**neue Schnittstelle in *ToolsAPI.PAS* (ab Delphi 4):**
aus der Unit *ExptIntf:*	
TIExpert	*IOTAWizard* und Nachfolger
aus der Unit *ToolIntf:*	
TIToolServices	verschiedene Schnittstellen mit der Endung *Services*
TIMainMenuIntf	für Native Tools: *TMainMenu*
TIMenuItemIntf	für Native Tools: *TMenuItem*
aus der Unit *EditIntf:*	
TIModuleInterface	*IOTAModule* und Nachfolger
TIEditorInterface	*IOTAEditor* und Nachfolger
TIEditReader	IOTAEditReader
TIEditWriter	IOTAEditWriter
TIEditView	IOTAEditView
TIFormInterface	INTAFormEditor
TIComponentInterface	für Native Tools: *TComponent*

Schnittstellen von Wizards und Experten

Als erste dieser Klassen nehmen wir uns natürlich die Experten bzw. Wizards selbst vor. Um alle von Delphi vorgesehenen Arten von Wizards und Experten auf einen Blick zu sehen, eignet sich die schon aus älteren Delphi-Versionen bekannte *TIExpert*-Klasse am besten. Sie ist in der Unit *ExptIntf* wie folgt definiert:

```
TIExpert = class(TInterface)
  public
  { Expert UI strings }
  function GetName: string; virtual; stdcall; abstract;
  function GetAuthor: string; virtual; stdcall; abstract;
  function GetComment: string; virtual; stdcall; abstract;
  function GetPage: string; virtual; stdcall; abstract;
  function GetGlyph: HICON; virtual; stdcall; abstract;
  function GetStyle: TExpertStyle; virtual; stdcall; abstract;
  function GetState: TExpertState; virtual; stdcall; abstract;
  function GetIDString: string; virtual; stdcall; abstract;
  function GetMenuText: string; virtual; stdcall; abstract;

  { Launch the Expert }
  procedure Execute; virtual; stdcall; abstract;
end;
```

Ein Expertenobjekt, das Sie an die oben bereits gezeigte Funktion *RegisterLibraryExpert* übergeben wollen, muss eine Klasse haben, die von *TIExpert* abgeleitet ist. Diese abgeleitete Klasse muss je nach Art des Experten bestimmte der abstrakten *TIExpert*-Methoden definieren.

Die neun *Get*-Methoden liefern der IDE Informationen, die besagen, wie der Experte in der Delphi-IDE dargestellt werden soll. Die grundlegende Information ist *GetStyle*, die besagt, welcher der vier möglichen Arten der Experte ist. Die drei Strings *GetName* (Name des Experten), *GetAuthor* (Autor) und *GetIDString* (Identifikation) helfen, den Experten und seinen Autor zu identifizieren. Der von *GetIDString* zurückgelieferte String sollte zudem eindeutig sein, was erreicht werden kann, wenn er nach dem Muster »Firmenname.Expertenfunktion« aufgebaut wird.

Ob der Rückgabewert der anderen *Get*-Funktionen eine Bedeutung hat, entscheidet sich anhand der von *GetStyle* festgelegten Art des Experten:

- *esStandard*-Experten werden über das Hilfe-Menü von Delphi aufgerufen. Die Methode *GetMenuText* gibt in diesem Fall den Text an, der im Menü angezeigt werden soll. Dieser Text kann sogar kontextsensitiv sein, denn *GetMenuText* wird bei jedem Öffnen des Hilfe-Menüs neu abgefragt. Mit *GetState* können Sie ebenso kontextsensitiv festlegen, ob der Menüpunkt aktiviert oder eventuell sogar mit einer Markierung versehen werden soll.
- *esForm*- und *esProject*-Experten werden über die Objektgalerie (DATEI | NEU | WEITERE) aufgerufen. Folglich benötigt Delphi den Namen der Seite, auf welcher der

Experte erscheinen soll (*GetPage*), ein Icon (*GetGlyph*, Rückgabe von 0 bewirkt Anzeige eines Standard-Icons) und einen Kommentar (*GetComment*). Auch der Name des Autors (*GetAuthor*) wird in der Objektgalerie angezeigt, in den neueren Delphi-Versionen jedoch nur, wenn Sie über das Kontextmenü die Detail-Ansicht einschalten. Das bereits erwähnte, mit Delphi mitgelieferte Beispielprojekt *Expt-Demo* gibt jeweils ein Beispiel für diese beiden Expertentypen.

- *esAddIn*-Experten klinken sich meistens selber irgendwo in Delphis Menü ein (sie sind also nicht auf das Hilfe-Menü angewiesen). Jedenfalls bindet Delphi sie nicht von sich aus irgendwo in die Benutzeroberfläche ein, daher brauchen Experten dieses Typs eigentlich nur *GetStyle*, *GetName* und *GetIDString* zu bearbeiten.

Die für alle Experten – mit Ausnahme der Add-In-Experten – wichtigste Methode ist *Execute*. Diese wird von Delphi aufgerufen, wenn der Menüpunkt des Experten bzw. sein Symbol in der Objektablage vom Benutzer aktiviert wurde. *Execute* hat also die Aufgabe, den Expertenjob auszuführen. Add-In-Experten werden nicht über *Execute* aufgerufen, sondern müssen selbst für ihren Aufruf sorgen, beispielsweise, indem sie sich selbst in Delphis Menü eintragen und eine *OnClick*-Bearbeitungsmethode für ihren Menüpunkt zur Verfügung stellen.

Wizards

Wie Abbildung A.1 zeigt, sind die neuen Wizard-Interfaces in einer Vererbungshierarchie angeordnet. Dies ändert nichts daran, dass ein Wizard im Prinzip die gleichen virtuellen Methoden braucht wie ein Experte. Jedem Stil eines Experten steht nun eine eigene Wizard-Schnittstelle gegenüber:

Experten-Stil	**Wizard-Interface**
esAddIn	IOTAWizard
esForm	IOTAFormWizard
esProject	IOTAProjectWizard
esStandard	IOTAMenuWizard

Sicherlich ist die Wizard-Hierarchie eleganter als die Alles-in-einem-Klasse *TIExpert*, die Implementation der notwendigen Methoden wird jedoch durch diese Eleganz nicht vereinfacht. Dies gilt erst recht, da die Vorgängerklasse von *IOTAWizard*, *IOTA-Notifier*, vier zusätzliche virtuelle Methoden vorschreibt, die es in *TIExpert* nicht gibt und die ein *IOTAWizard* normalerweise gar nicht braucht.

Eine Auflistung der in den jeweiligen Klassen benötigten virtuellen Methoden soll Ihnen hier erspart bleiben, da Sie diese ja in Delphis CodeExplorer auf eine viel übersichtlichere und zudem interaktive Weise erhalten, wenn Sie die Unit `ToolsApi.pas` in den Editor laden.

Das folgende Beispiel zeigt eine Wizard-Klasse, die über das Hilfe-Menü von Delphi aufgerufen werden soll, also die *IOTAMenuWizard*-Schnittstelle implementieren muss:

```
TWizard0 = class(TInterfacedObject, IOTANotifier, IOTAWizard,
                 IOTAMenuWizard)
 { von IOTANotifier... }
  procedure AfterSave;
  procedure BeforeSave;
  procedure Destroyed;
  procedure Modified;
 { von IOTAWizard... }
  function GetIDString: string; virtual;
  function GetName: string; virtual;
  function GetState: TWizardState; virtual;
  procedure Execute; virtual;
  { von IOTAMenuWizard... }
  function GetMenuText: string; virtual;
end;
```

Durch die Implementation der drei aufgezählten Interfaces wird *TWizard0* zu einer Co-Klasse, für die bestimmte Anforderungen gelten. Hierzu und zur Lösung dieser Anforderungen durch *TInterfacedObject* siehe Kapitel 2.7.4.

Wie bereits erwähnt, sind die von *IOTANotifier* geforderten Methoden für einen Wizard nicht notwendig. Um im Beispiel-Package nicht für jede der (bisher nur zwei) Wizard-Klassen diese vier unnötigen Methoden implementieren zu müssen, definiert die zum selben Package gehörende Unit *ToolUtil* eine Klasse *TWizardBase*, die von *TInterfacedObject* abgeleitet ist und die die vier *IOTANotifier*-Methoden mit leeren Rümpfen implementiert. So kann der erste Beispiel-Wizard statt von *TInterfacedObject* wie folgt von *TWizardBase* abgeleitet werden:

```
TWizard1 = class(TWizardBase, IOTAWizard, IOTAMenuWizard)
```

Die restlichen Zeilen stimmen bis auf die gesparten *IOTANotifier*-Methoden mit dem Listing von *TWizard0* überein, daher können wir gleich zur Implementation des ersten Beispiel-Experten kommen:

```
procedure TWizard1.Execute;
begin // Ausführen der Experten-Funktion durch Aufruf eines Formulars
  Expert1Form := TExpert1Form.Create(nil); // ... das dynamisch erzeugt
  Expert1Form.ShowModal; // ... und modal angezeigt wird.
  Expert1Form.Free;
  Expert1Form := nil;
end;

function TWizard1.GetIDString: String;
begin // Identifizierung durch einen eindeutigen String
  Result := 'WarkenElmar.Ide4ExplorerExpert1AsWizard';
end;
```

```
function TWizard1.GetMenuText: String;
begin // Text, der im Hilfe-Menü angezeigt wird
  Result:='IDE Explorer zur Expertenprogrammierung (Wizard-Interface)...';
end;

function TWizard1.GetName: String;
begin // Name des Experten
  Result := 'Delphi IDE ToolsApi-Explorer (per Wizard)';
end;

function TWizard1.GetState: TWizardState;
begin // Aktivierungs- und Markierungszustand des Menüpunkts
  Result := [wsEnabled]; // wsEnabled -> aktivierbar, aber nicht markiert.
end;
```

Hinweis: Die von *TWizard1* zur Verfügung gestellte Funktionalität wird im Beispiel-Package gleich zweimal in die Delphi-IDE eingebunden. Einmal durch die gezeigte Wizard-Klasse, ein weiteres Mal durch eine von *TIExpert* abgeleitete Expertenklasse im Stil eines *esAddIn*-Experten. Dieses Beispiel zeigt zum einen die Unterschiede zwischen Wizards und Experten und macht zum anderen deutlich, dass das, was im Programm als »Experte« und »Wizard« bezeichnet wird, gar nicht der wirkliche Experte bzw. Wizard ist, denn die eigentliche Funktion steckt normalerweise ganz woanders, im vorliegenden Beispiel im Formular *TExpert1Form*.

Schnittstellen von Modulen

Ein Experte, der nicht zur Erzeugung neuer Formulare und Projekte, sondern zur Bearbeitung existierender Module gedacht ist, muss sich zunächst einmal in der Delphi-IDE orientieren, welche Module, Units, Formulare usw. überhaupt vorhanden sind und welches das aktuelle Modul ist. Hierzu stehen ihm sowohl die Delphi 3-kompatiblen als auch die von Delphi 4 neu eingeführten Schnittstellen zur Verfügung. Grundsätzlich benötigen Sie in beiden Fällen zunächst einmal Zugriff auf ein Basisobjekt bzw. auf eine Basisschnittstelle:

- Wenn Sie die herkömmlichen OpenTools-Klassen verwenden, das Objekt *ToolServices*, welches in der Unit *ExptIntf* deklariert ist. Die Klasse dieses Objekts heißt *TIToolServices* und ist in *ToolIntf* zu finden.
- Wenn Sie die neuen Interfaces verwenden, das Interface *BorlandIDEServices*, deklariert in der Unit *ToolsAPI*. Da dieses Interface keine eigenen Methoden besitzt, müssen Sie es zuerst in eines der anderen Interfaces mit der Endung *...Services* umwandeln – für die Modulinformationen beispielsweise in *IOTAModuleServices*.

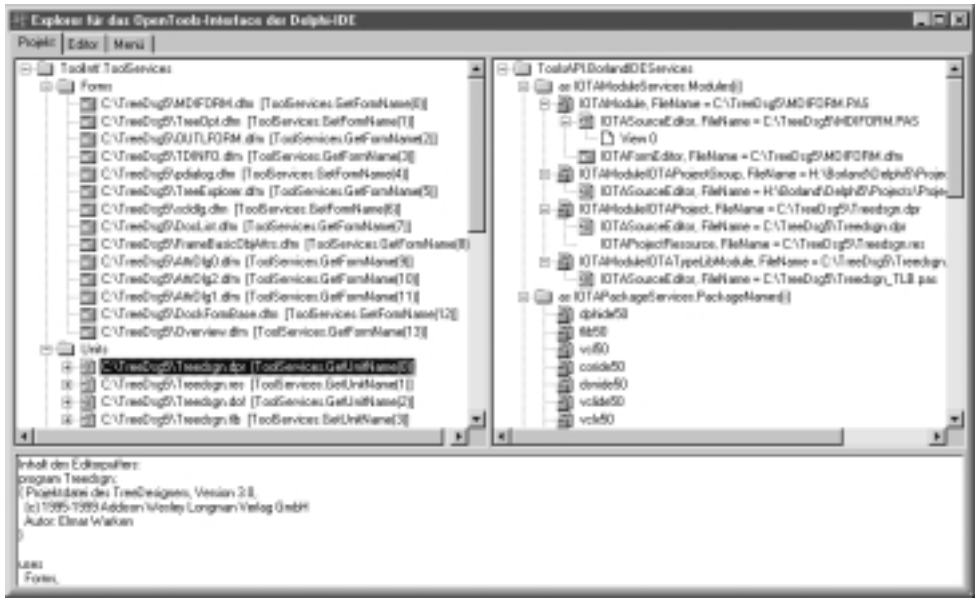


Abbildung A.2: Diese beiden TreeViews des ToolsAPI-Explorers zeigen gewissermaßen die »Besitzhierarchie« der ToolsAPI-Schnittstellen: links die Informationen der in allen Delphi-Versionen bekannten ToolServices, rechts die der neueren BorlandIDEServices.

Der ToolsAPI-Explorer-Beispielexperte visualisiert die Struktur, in der die Modulinformationen im OpenTools-API vorliegen (Abbildung A.2), auch hier gibt es kleine Unterschiede zwischen den alten und den neuen Schnittstellen:

- Die Methoden der unter allen Delphi-Versionen zur Verfügung stehenden Unit *ToolIntf* unterscheiden zwischen Formularen, Units und Modulen, wobei der Begriff *Module* sich hier auf die Komponentenpalette bezieht, seit Delphi 3 also eigentlich *Package* meint. Mit *ToolServices.GetFormName(i)*, *GetUnitName(i)* und *GetModuleName(i)* können Sie sich einfach alle Formulare und Units des gerade geladenen Projekts und alle geladenen Packages auflisten lassen.

  Um eine Schnittstelle zum aktuellen Modul zu erhalten, rufen Sie *ToolServices.GetModuleInterface* mit dem von *ToolServices.GetCurrentModule* zurückgelieferten Namen auf. Den Namen des aktuellen Projekts erhalten Sie mit *ToolServices.GetProjectName*.

- Die Schnittstellen der ab Delphi 4 existierenden *ToolsAPI*-Unit unterscheiden auf oberster Ebene nur zwischen Modulen und Packages. Ein Modul kann entweder eine einzelne Unit oder eine Einheit aus Formular und zugehöriger Unit sein. Die einzelnen zu einem Modul gehörenden Dateien werden mit *GetModuleFileEditor(i)* abgefragt, und zwar liefert diese Methode zunächst eine allgemeine *IOTAEditor-*

Schnittstelle. Der Editor kann entweder ein Formular- oder ein Quelltexteditor sein. Im ersten Fall unterstützt er eine *IOTAFormEditor*-, im zweiten eine *IOTASourceEditor*-Schnittstelle. Wie im COM üblich, müssen Sie diese Schnittstellen von der *QueryInterface*-Methode der *IOTAEditor*-Schnittstelle erfragen.

Das aktuelle Modul erhalten Sie durch einen Aufruf der Methode *IOTAModuleServices.CurrentModule*. Die Namen der Projekte und der Projektgruppe lassen sich herausfinden, wenn Sie die Liste der Module nach Modulen durchsuchen, die die *IOTAProject* bzw. *IOTAProjectGroup*-Schnittstelle unterstützen. Um das aktuelle Projekt herauszufinden, bietet sich jedoch eher die bereits erwähnte einfache Methode *ToolIntf.GetProjectName* an.

Für Details zum Aufruf der einzelnen Methoden sei auch hier wieder auf den definitiven OpenTools-Quelltext verwiesen, an dieser Stelle soll als Beispiel eine leicht gekürzte Methode des Beispielexperten genügen, und zwar die Methode, die beim Expandieren des *Module*-Knotens im rechten TreeView aufgerufen wird:

```
procedure TModulesData2.CreateSubElements;
// Expandieren des Module-Knotens im TreeView für die neuen Interfaces
// TModulesData2 besitzt 2 Datenelemente:
//   Node für den verbundenen TTreeNode und IModule für das Modul
var
  ChildNode: TTreeNode; // Speichert den jeweils neu erzeugten Knoten
  // Schnittstellen der Delphi-IDE:
  ModuleServices: IOTAModuleServices;
  IModule: IOTAModule;
  // Zum Testen der weiteren von IModule implementierten Schnittstellen:
  I1: IOTATypeLibModule; I2: IOTAProject;  I3: IOTAProjectGroup;
  i: Integer;
  Typ: String;
begin
  // Basis-Schnittstelle erfragen:
  ModuleServices := BorlandIDEServices as IOTAModuleServices;
  // Wenn 0 Module vorhanden, Knoten als nicht expandierbar darstellen:
  if ModuleServices.ModuleCount = 0 then Node.HasChildren:=false
  // Sonst alle Module durchlaufen und Knoten zum TreeView hinzufügen:
  else for i := 0 to ModuleServices.ModuleCount - 1 do begin
    IModule := ModuleServices.Modules[i];
    // Feststellen des Modul-Typs:
    IModule.QueryInterface(IOTATypeLibModule, I1);
    ... entsprechend wird mit I2 und I3 verfahren und Typ gesetzt ...
    // Neuen TreeView-Knoten erzeugen:
    ChildNode := TTreeView(Node.TreeView).Items.
      AddChildObject(Node, '', TModuleData2.Create);
    ChildNode.Text := Typ + ', FileName = ' + IModule.FileName;
    ChildNode.HasChildren := true; // Nächste Expandierstufe ermöglichen
    ChildNode.ImageIndex := 1; // Modul-Icon
    ChildNode.SelectedIndex := ChildNode.ImageIndex;
    // Den mit dem Knoten verknüpften Record setzen:
```

```
      TModuleData2(ChildNode.Data).Node := ChildNode;
      TModuleData2(ChildNode.Data).IModule := IModule;
    end;
  end;
```

Diese Methode erzeugt für jedes Modul einen *TModuleData2*-Record und speichert darin unter anderem die *IModule*-Schnittstelle des Moduls. Auf diese Weise kann beim Expandieren des Modulknotens sofort festgestellt werden, welches Modul gemeint ist. (Siehe hierzu auf der CD die Methode *TModuleData2.CreateSubElements*; auf ähnliche Weise geht das Programm auch beim Expandieren der anderen Knotentypen vor.)

Einbinden von neuen Funktionen in Delphis Menü

Die bereits abgedruckte Experten-Klasse mit ihrer Methode *GetMenuText* demonstrierte bereits, wie Sie einen Experten in das Hilfe-Menü einbinden und Delphi beim Anklicken des Menüpunkts für den Aufruf der Methode *Execute* sorgen lassen. Soll ein Experte jedoch in einem anderen von Delphis Menüs auftauchen und vielleicht auch über ein Tastenkürzel aufrufbar sein, müssen Sie diesen einfachen Weg des *IOTAMenuWizard* bzw. *esStandard*-Experten verlassen und einen allgemeinen *IOTAWizard* bzw. einen *esAddIn*-Experten schreiben.

Dieser muss selbst einen neuen Menüpunkt konstruieren und kann ihn dann an eine nahezu beliebige Position in Delphis Menü eintragen. Seit Delphi 4 gibt Ihnen das OpenTools-API sogar direkten Zugriff auf die *TMainMenu*-Komponente der IDE, und darüber hinaus können Sie sogar auf die *TToolbar*-Komponenten und auf die damit verbundenen Aktions- und Bilderlisten der IDE zugreifen (siehe *ToolsAPI.INTAServices*, wird in den Beispielexperten nicht genutzt).

Um das Hauptmenü-Objekt zu erhalten, muss die *BorlandIDEServices*-Schnittstelle dieses Mal in ein *INTAServices*-Interface umgewandelt werden:

```
var
  MainMenu: TMainMenu;
begin
  if BorlandIDEServices <> nil then
    MainMenu := (BorlandIDEServices as INTAServices).MainMenu;
    // Genauso wie MainMenu können übrigens auch ToolBar, ActionList
    // und ImageList gelesen werden.
```

Zur weiteren Veränderung des Menüs kann ein Experte nun genauso vorgehen wie eine normale Delphi-Anwendung, die ihr Menü zur Laufzeit anpasst. Dies wurde bereits in Kapitel 4.6 ausführlich beschrieben.

Komplizierter läuft hingegen die Einbindung eines Menüpunkts mit der Delphi-3-kompatiblen *ToolIntf*-Unit und der darin deklarierten Klasse *TIMenuItemIntf* ab. Für ein Beispiel sei hier auf die Funktion *InsertMenuItem* der Unit *ToolUtil* verwiesen.

Hinweis: Beim Erzeugen neuer Menüpunkte mit den *ToolIntf*-Klassen benötigen Sie den Namen der *TMenuItem*-Komponenten innerhalb der Delphi-IDE. Dies ist der Grund, warum der *ToolsAPIExplorer* auf seiner dritten Seite einen Baum aller Delphi-Menüpunkte mitsamt deren Namen angibt. Ein Beispiel zum Aufruf der *InsertMenuItem*-Hilfsfunktion mit einem solchen Namen folgt sogleich ...

Add-In-Experten

Add-In-Experten sind Experten, die selbst eine Möglichkeit schaffen, mit der der Benutzer sie aufrufen kann; sie sind also weder an das Hilfe-Menü noch an die Objektablage gebunden. Während *TIExpert*-Klassen über den Stil *esAddIn* zu einem Add-In-Experten werden, ist ein Add-In-»Wizard« dadurch gekennzeichnet, dass er nur die *IOTAWizard*-Schnittstelle implementiert (und nicht etwa die *IOTAMenuWizard*-Schnittstelle).

Zur Demonstration der Funktionsweise von Add-In-Experten sei hier schon einmal der im nächsten Abschnitt genauer erläuterte *TEditorAssistants*-Experte herangezogen. Im Gegensatz zum bisher gesehenen *TWizard1* definiert dieser auch seinen eigenen Konstruktor und Destruktor. Der Konstruktor eignet sich hervorragend zur Erzeugung der notwendigen Menüpunkte. Dabei ist *EditAddToInterfaceItem* der Name des Delphi-Menüpunkts BEARBEITEN | ZUR SCHNITTSTELLE HINZUFÜGEN, also des letzten Punkts im Bearbeiten-Menü. Hinter diesem sollen hier die neuen Menüpunkte angehängt werden:

```
constructor TEditorAssistants.Create;
begin
  inherited Create;
  MICodeExplorer := InsertMenuItem('EditAddToInterfaceItem',
     '&Itéa Code-Explorer', 'EWIteaCodeExplorer',
     ShortCut(Ord('I'),[ssCtrl, ssShift]), UpdateCodeExplorerClick);
  ... Eintragung weiterer Menüpunkte ...
```

UpdateCodeExplorerClick ist dabei die Methode, die beim Aufruf des Menüpunkts ausgeführt wird. Sie sorgt dafür, dass der im Package definierte CodeExplorer (siehe A.4) in den Vordergrund geholt und aktualisiert wird.

Wichtig ist, dass der Menüpunkt am Ende auch wieder gelöscht wird. Daher handelt es sich bei *MICodeExplorer* um eine Variable der Klasse *TEditorAssistants*, die im Destruktor wieder freigegeben werden kann:

```
destructor TEditorAssistants.Destroy;
begin
  if MICodeExplorer <> nil then MICodeExplorer.DestroyMenuItem;
  ... Löschen der weiteren Menüpunkte ...
  inherited Destroy;
end;
```

A.2 Erweiterung des Editors

Das Modul *EditorAssistants* aus dem Package *IteaExperten* implementiert drei Funktionen für Delphis Editor, die via Tastenkürzel aufgerufen und teilweise ganz ohne Dialog mit dem Benutzer ablaufen (daher kann auch hier schwerlich von einem »Wizard« gesprochen werden). Die drei Funktionen sind:

- Variablendeklaration (Strg+Shift+D): Soll allen Programmautoren, die nicht schon vor dem Schreiben einer Methode alle benötigten Variablen vorhersehen können, das lästige Hin- und Herlaufen zwischen Anweisungsblock und *var*-Deklarationsblock ersparen, indem der Bezeichner an der Eingabeposition als lokale Variable der aktuellen Methode deklariert wird. Die Auswahl des Variablentyps erfolgt über einen Dialog und kann für häufig benötigte Typen mit einem zweiten Tastenkürzel durchgeführt werden.
- »Intra-File-HyperJump« (Strg+Shift+H): Springt zur Deklaration des Symbols an der Cursor-Position, sofern es sich um ein Symbol des aktuellen Moduls handelt.
- Aufruf des Itéa-CodeExplorers, der in A.4 vorgestellt werden wird (Strg+Shift+I).

Die unter Delphi registrierte Experten-Klasse von *EditorAssistants* heißt *TAssistantCollection*. Die beiden benötigten Formulare sind das nicht-modale Formular des CodeExplorers in der Unit *IteaCodeExplorer* und das modale Dialogformular zur Deklaration einer Variablen in *VarDeclDlg*.

Itéa-DLL

Alle drei Funktionen arbeiten mit den Symbolinformationen der gerade in Delphis Editor geladenen Unit. Die 1998 eingeführten AppBrowser-Funktionen von Delphi zeigen, dass Delphi selbst über diese Informationen verfügt. Da das ToolsAPI jedoch auf diese Symbolinformationen noch keinen Zugriff erlaubt und der Beispielexperte außerdem nicht nur mit Delphi Professional und aufwärts funktionieren soll, greifen alle drei Funktionen von *EditorAssistants* auf die DLL-Version des in C++ geschriebenen Tools *Itéa* zurück, das bereits in Kapitel 5.1.5 als Entwicklungsumgebung in Erscheinung getreten ist.

`IteaDll.dll` beinhaltet alle Modul-Parse-Funktionen der eigenständigen Itéa-Umgebung und macht diese über einige ausgewählte Klassenschnittstellen nach außen hin zugänglich. Das Prinzip zum Aufruf von Klassenschnittstellen in DLLs wird in Kapitel 8.3.3 vorgestellt, daher soll an dieser Stelle nicht weiter beschrieben werden, wie die Unit *Itea* in ihrem *initialization*-Teil bereits ein globales C++-Objekt innerhalb der DLL erzeugt, auf das vom Delphi-Programm aus zugegriffen werden kann.

Schlüssel zur Verwendung der Unit *Itea* ist diese Funktion:

```
function GetIteaModule(SourceCode, SourceName: String): ModuleHandler;
```

GetIteaModule erwartet im ersten Parameter den kompletten Quelltext einer Unit in Form eines Strings, im zweiten Parameter den Namen der Unit. Sie bedient sich des bereits erwähnten globalen Parser-Objekts, um den Quelltext zu parsen, Symbolinformationen aufzubauen und in Form eines *ModuleHandler*-Objekts auf den Aufrufer zurückzugeben.

Die Klassen *ModuleHandler* und *ScopeHandler* sind die beiden einzigen C++-Klassen, die wir im Folgenden benötigen werden. Falls Sie selbst Experimente mit der Itéa-DLL unternehmen wollen, benötigen Sie ebenfalls minimal nur diese beiden Klassen und die Funktion *GetIteaModule*. Zum Einbinden der DLL in eine Anwendung genügt das Aufnehmen der Unit *Itea* in die *uses*-Anweisung. Natürlich muss sich `iteadll.dll` zur Laufzeit auch im Verzeichnis der EXE-Datei oder in einem Verzeichnis des Suchpfades befinden (um in einer Delphi-IDE-Erweiterung genutzt werden zu können, wird die DLL daher im einfachsten Fall in das *bin*-Verzeichnis von Delphi kopiert).

ModuleHandler

Das Ergebnis eines Parse-Vorgangs von Itéa ist also zunächst einmal ein Modul-Objekt, welches in Delphi über die *ModuleHandler*-Schnittstelle angesprochen wird. Da die tatsächlich in C++ verwendete Klasse *CModule* Dutzende virtueller Methoden enthält, wurde hier zur Vereinfachung die Klasse *ModuleHandler* angelegt, die eine einfache Hülle um ein *CModule*-Objekt bildet und in Delphi wie folgt deklariert ist:

```
ModuleHandler = class
  function GetModuleCount: Integer; virtual; cdecl; abstract;
  function GetModule(i: Integer): ModuleHandler; virtual; cdecl; abstract;
  function AsScope: ScopeHandler; virtual; cdecl; abstract;
  function GetFuncAtPos(Line, Col: Integer): ScopeHandler; virtual; ...
  function FindSymbol(Name: PChar): ScopeHandler; virtual; cdecl; ...
  procedure FreeWithModule; virtual; cdecl; abstract;
  procedure Free;{ in Object Pascal definiert, um nil-Objekte abzufangen }
end;
```

Für die weitere Verwendung von *ModuleHandler* in diesem Anhang sind vor allem die folgenden Methoden wichtig (zur Beschreibung der übrigen Methoden siehe den Quelltext auf der CD):

- Die Funktion *GetFuncAtPos* ermittelt, welche Methode oder Funktion sich an einer vorgegebenen Position des Editors befindet. Sie liefert das C++-Objekt, welches alle geparsten Informationen über die Methode bzw. Funktion enthält, in Form einer *ScopeHandler*-Schnittstelle zurück.

- *FindSymbol* sucht ein globales Symbol mit dem angegebenen Namen, auch sie liefert ein *ScopeHandler*-Objekt.
- *AsScope* liefert schließlich ein *ScopeHandler*-Objekt für das Modul selbst. Mit Hilfe dieses Objekts lassen sich beispielsweise alle globalen Objekte des Moduls aufzählen.
- Die Module, die Sie mit *GetModule* abrufen können, sind die Units, die per *uses*-Anweisung in das geparste Modul eingebunden werden. Diese Modulliste wird erst für den in A.4 beschriebenen CodeExplorer benötigt, der ja wie der Explorer von Delphi auch eine Unit-Liste haben soll.

Hinweis: In der derzeitigen Version der Itéa-DLL liegen für die von *GetModule* zurückgelieferten Module grundsätzlich keine weiteren Symbolinformationen vor. Der Grund dafür liegt in der langen Zeit, die Itéa benötigen würde, um alle beteiligten Module einzulesen.

ScopeHandler

Wir sind nun im Zentrum der Symbolinformationen von Itéa angekommen: *ScopeHandler* ist die Klasse, die die Symbolinformationen eines einzelnen Object-Pascal-Symbols beschreibt. Das folgende Listing zeigt nur exemplarisch einige der Methoden dieser Klasse, die im Folgenden wichtig sind:

```
ScopeHandler = class
  // GetName liefert den Namen des Symbols:
  function GetName: PChar; virtual; cdecl; abstract;
  // Bei Funktionen statt GetName die folgende Methode verwenden:
  procedure CopyName(Dest: PChar; DestLen: Integer); virtual; cdecl; ab...
  // Falls es sich um eine Klasse handelt, die Elternklasse ermitteln:
  function GetParent: ScopeHandler; virtual; cdecl; abstract;
  // Die Position der Deklaration des Symbols abfragen:
  procedure GetDeclPos(var Line, Col: Integer); virtual; cdecl; abstrac...
  // Falls es sich um eine Funktion oder Methode handelt, liefert die
  // folgende Methode die Position der letzten Deklaration einer lokalen
  // Variablen dieser Funktion:
  procedure GetLastDeclPos(var Line, Col: Integer); virtual; cdecl; abs...
  // Die Position der Definition des Symbols abfragen; dies bezieht
  // sich nur auf Methoden und Funktionen und meint die Position
  // des Methoden-/Funktionskopfes über der Implementation:
  procedure GetDefPos(var Line, Col: Integer); virtual; cdecl; abstract;
  // Zeile, in der der Rumpf der Funktion/Methode beginnt:
  function GetStartLine: Integer; virtual; cdecl; abstract;
  // Zahl der Variablen der Funktion/Methode/Klasse abfragen:
  function GetVarCount: Integer; virtual; cdecl; abstract;
  // Ein ScopeHandler-Objekt auf eine einzelne Variable zurückliefern:
  function GetVar(i: Integer): ScopeHandler; virtual; cdecl; abstract;
```

```
    // Das Symbol mit dem angegebenen Namen suchen. FindSymbol liefert im
    // Erfolgsfall einen ScopeHandler zurück. Die Suche beginnt im aktuellen
    // Gültigkeitsbereich. Der Aufruf von KlasseX.FindSymbol bewirkt z.B.,
    // dass zuerst in KlasseX nach dem Symbol gesucht wird, dann in der Unit
    // dieser Klasse.
    function FindSymbol(Name: PChar): ScopeHandler; virtual; cdecl; abstr...
    // Den ScopeHandler freigeben:
    procedure Free; { Diese Prozedur ist nicht in C++, sondern in der
                      Pascal-Unit Itea definiert. }
end;
```

Als Verwendungsbeispiel finden Sie im weiteren Verlauf die Implementation der HyperJump-Funktion des Beispielexperten. Zuerst ist jedoch weitere Vorarbeit notwendig, denn diese Funktion benötigt ja auch noch Zugriff auf Delphis Editor.

Editor-Schnittstellen

Mit dem OpenTools-API können Sie alle Editorinhalte von Delphi auslesen und sogar verändern, allerdings ist dies schon etwas schwieriger als der Umgang mit dem *Lines*-Property einer *TMemo*-Komponente. Ausgangspunkt für alle Aktivitäten bezüglich des Editorinhalts sind die Klasse *TIEditorInterface* in *EditIntf* bzw. die Interfaces *IOTAEditor* und *IOTASourceEditor* in *ToolsAPI*. Sowohl in den alten als auch in den neuen Schnittstellen finden Sie zunächst die folgenden grundlegenden Daten:

- Ein Flag, das angibt, ob der Editorpuffer seit der letzten Speicherung geändert wurde (*TIEditorInterface.BufferModified* und *IOTAEditor.Modified*).
- Angaben über den gerade markierten Textblock (GetBlockAfter, GetBlockStart, GetBlockType und GetBlockVisible sowohl in TIEditorInterface als auch in IOTASourceEditor).
- Angaben über die geöffneten Views, in denen der Editorinhalt angezeigt wird (*TIEditorInterface:* Methoden *GetViewCount* und *GetView*, *IOTASourceEditor:* Properties *EditViewCount* und *EditView*).

Ein View entspricht einer Seite in Delphis Editorfenster. (Um dieselbe Datei auf zwei verschiedenen Editorseiten, also in zwei Views, anzeigen zu lassen, müssen Sie aus dem lokalen Editormenü mit NEUES EDITIERFENSTER zuerst ein weiteres Editorfenster öffnen.)

Ein View wird durch ein *TIEditView*-Objekt bzw. durch eine *IOTAEditView*-Schnittstelle repräsentiert. Zu den Daten eines Views gehören in beiden Fällen die aktuelle Eingabeposition (*CursorPos)*, die Zeichenposition, die in die linke obere Ecke des Views gescrollt wurde (*TopPos*), und die Größe des Views (*ViewSize*). Wie Sie in der IDE ausprobieren können, können diese Daten für jede Editorseite verschieden sein, während beispielsweise die aktuelle Markierung auf allen Seiten des gleichen Inhalts übereinstimmt.

Die beiden verschiedenen Wege, mit denen Sie zur Schnittstelle des Editors und seiner einzelnen Views gelangen können, sind in Abbildung A.3 dargestellt. Der Beispielexperte geht den abwärtskompatiblen Weg über *ToolServices*, muss also zuerst den Namen der aktuellen Datei erfragen (*ToolServices.GetCurrentFile*), mit dem er von *GetModuleInterface* die *TIModuleInterface*-Schnittstelle des aktuellen Moduls verlangen kann. Dann ruft er die Methode *TIModuleInterface.GetEditorInterface* auf, um ein *TIEditorInterface* zu erhalten. Von diesem kann er im weiteren Verlauf Schnittstellen für die Views (*TIEditView*) erhalten und außerdem Schnittstellen, mit denen er den Editorinhalt lesen und beschreiben kann (*TIEditReader* und *TIEditWriter*). Das *TIEditorInterface* ist allerdings nur dann verfügbar, wenn der Benutzer die entsprechende Unit auch in einem Delphi-Editor geöffnet hat.

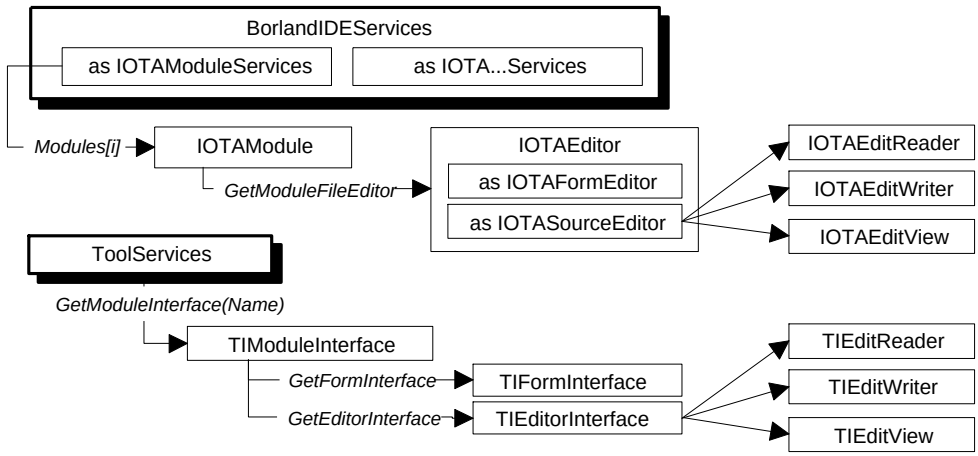


Abbildung A.3: Auf unterschiedlichen Wegen zum Ziel wichtiger Editorschnittstellen

Lesen und Schreiben in Delphis Editor

Der Beispielexperte *ToolsAPIExplorer* demonstriert bei zwei Gelegenheiten die Verwendung der *TIEditReader*- bzw. *TIEditWriter*-Schnittstelle: Wenn Sie auf der ersten Seite (Abbildung A.2) im linken TreeView eine Unit auswählen, die gerade in einem Editor bearbeitet wird, zeigt er den Editorinhalt im unteren Bereich des Fensters an. Interessanter ist die zweite Seite desselben Fensters (Abbildung A.4). Sie listet alle zur Verfügung stehenden Editorpuffer auf, zeigt natürlich wieder deren Inhalt an und bietet außerdem eine Möglichkeit, diese Editorpuffer zu verändern.

Wir beginnen mit der einfachen *TIEditReader*-Schnittstelle. Sie verfügt nur über eine einzige Methode namens *GetText*, die ab einer bestimmten Position eine bestimmte Anzahl von Zeichen ausliest und in einen Puffer schreibt. Um den gesamten Inhalt auszulesen, ist also ein wiederholter Aufruf von *GetText* notwendig, der prinzipiell wie folgt aussehen kann:

```
Reader := IEditor.CreateReader;
BufferPos := 0;
repeat
  ReadSize := Reader.GetText(BufferPos, @Buffer, PartSize);
  if ReadSize = PartSize then begin
    // Angeforderte Textmenge konnte gelesen werden,
    // mehr Speicher für den nächsten Textblock reservieren.
   ...
   inc(BufferSize, PartSize);
  end;
until ReadSize < PartSize; (* Wenn weniger gelesen wurde als
       angefordert, ist das Ende des Editorinhalts erreicht. *)
```

Die vollständige Methode finden Sie in der Unit *ToolUtil* unter dem Namen *GetEditorText*. Sie verwendet eine *PartSize*-Konstante mit dem Wert 20.000, liest also in jedem Schritt 20.000 weitere Zeichen aus Delphis Editorpuffer aus.

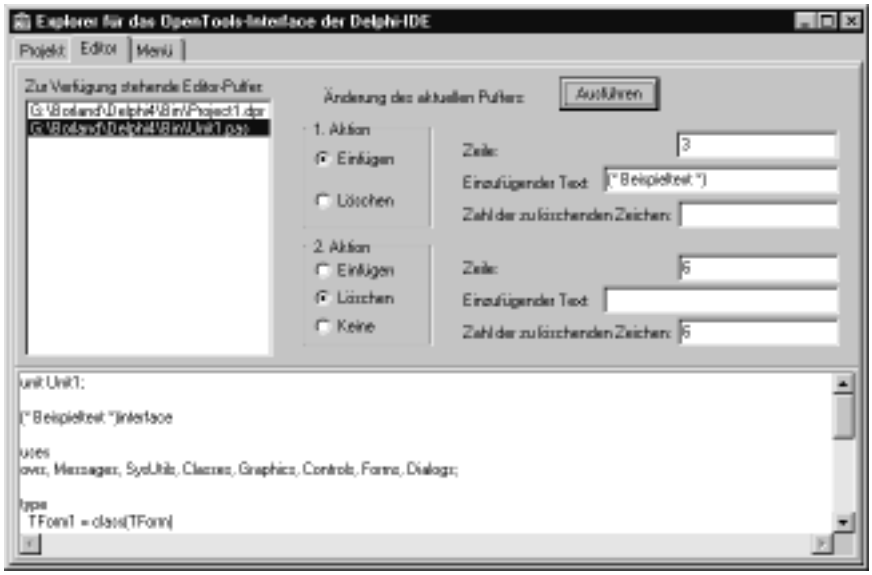


Abbildung A.4: Auf dieser Seite des Beispielexperten haben Sie Gelegenheit, die TIEditWriter-Schnittstelle der Delphi-IDE interaktiv zu testen.

Die Benutzung eines *TIEditWriter* kann man sich am besten so vorstellen, als würde man den gesamten Inhalt des Editors von vorne bis hinten durchlaufen und dabei alles, was nicht verändert werden soll, kopieren sowie bei Bedarf einige Änderungen vornehmen. *TIEditWriter* besitzt nämlich keine Methoden, um direkt Änderungen an einer bestimmten Position durchzuführen, sondern verfügt lediglich über einen Positionszeiger, der zu Beginn bei 0 steht und im weiteren Verlauf nur erhöht werden kann, und zwar mit den folgenden drei Methoden:

- *CopyTo(Pos)* übernimmt den Text vom aktuellen Positionszeiger beginnend bis zur angegebenen *Pos* unverändert in die neue Version des Textes. Der Positionszeiger wird auf *Pos* gesetzt.
- *DeleteTo(Pos)* löscht den Text zwischen Positionszeiger und der angegebenen Position, danach wird der Positionszeiger auf diese neue Position gesetzt. Man könnte auch sagen, *DeleteTo* bewirkt, dass beim Kopieren in die neue Textversion ein Teil des Textes übersprungen wird.
- *Insert(Text)* fügt den angegebenen *Text* an der aktuellen Position der neuen Textversion ein. Der Positionszeiger, der sich ja auf die alte Textversion bezieht, wird nicht verändert.

Um beispielsweise das vierte Zeichen einer Quelltextdatei zu löschen, müssten Sie mit *CopyTo(3)* die ersten drei Zeichen unverändert übernehmen und dann mit *DeleteTo(4)* das Zeichen löschen. Wenn Sie den *TIEditWriter* danach wieder freigeben, werden automatisch alle Zeichen, die noch nach dem Positionszeiger folgen, kopiert, als hätten Sie selbst *CopyTo(Position des Dateiendes)* aufgerufen.

Schon dieses Funktionsprinzip weist darauf hin, dass ein *EditWriter* immer nur dazu verwendet werden sollte, eine Editieroperation »am Stück« durchzuführen. Bevor irgendwelche anderen Programmteile (möglicherweise auch Delphi selbst) die Möglichkeiten haben, auf den Editorpuffer zuzugreifen, sollte der *EditWriter* wieder gelöscht werden. Außerdem verbietet es sich von selbst, gleichzeitig einen *EditWriter* und einen *EditReader* zu verwenden.

Im erwähnten Fenster des Beispielexperten können Sie jeweils ein oder zwei Editieroperationen ausprobieren, die Sie in den entsprechenden Eingabefeldern spezifizieren und dann mit AUSFÜHREN bestätigen. Das Ergebnis lässt sich sofort im Memo-Feld am unteren Fensterrand bewundern. Setzen Sie diese Testfunktion aber bitte nicht auf Quelltextdateien an, die Sie auf jeden Fall unbeschädigt weiterbenutzen wollen.

Hinweis: Neben der in Abbildung A.3 genannten Methode *CreateWriter* gibt es auch noch die Methode *CreateUndoableWriter*. Wenn Sie diese anstatt der erstgenannten verwenden, kann der Benutzer die Editieroperation über das bei ihm gültige Tastenkürzel rückgängig machen. Der Beispielexperte *EditorAssistants* verwendet einen solchen »rückgängigmachbaren« *EditWriter*. So komfortabel diese Funktion aber aussieht, in manchen Fällen ist es wünschenswert, wenn die Rückgängig-Funktion sich *nicht* auf automatisch veränderten Code bezieht. So können Sie beispielsweise Änderungen, die Delphi automatisch im Quelltext vornimmt (z.B. beim Einfügen einer Komponente in das Formular), nicht als Textoperation rückgängig machen.

Die Abfrage des Editorinhalts im Beispielexperten

Für den Zugriff auf den Editor verwendet das Package *IteaExperten* die Schnittstellen der Unit *EditIntf*: *TIEditorInterface*, *TIEditView*, *TIEditReader* und *TIEditWriter*. In der Unit *ToolsAPI* sind leicht die entsprechenden *IOTA*...-Interfaces zu finden (dabei stimmen *IOTAEditWriter*, *IOTAEditReader* und *IOTAEditView* sogar ziemlich genau mit den *EditIntf*-Klassen überein, für *IOTAEditor* gelten die in Zusammenhang mit Abbildung A.3 beschriebenen Unterschiede zur Auflistung von Modulen und Editoren).

Die für die Beispielexperten interessanten Daten des Delphi-Editors können wie folgt in einer neuen Datenstruktur zusammengefasst werden:

```
TEditorData = record
  FileName: String;    // Name der editierten Datei
  Position: TEditPos;  // Eingabeposition im aktuellen View
    // TEditPos besteht aus Col und Line, erste Spalte/Zeile = 1
  IntPos: Integer;     // Dieselbe Position, umgerechnet in Zeichen
                       // vom Beginn der Datei
  CurrentWord: String; // Wort an der Eingabeposition
  Content: PChar;      // Eine Kopie des gesamten Editorinhalts
end;
```

Der Abdruck der entsprechenden Methoden würde nach viel Arbeit aussehen, denn er besteht nicht nur aus einer Verwendung der genannten Properties und Methoden, sondern unter anderem auch aus vielen Sicherheitsabfragen, ob die entsprechenden *TI*...-Schnittstellen vorliegen, und aus Umrechnungen von Zeilen-/Spaltenangaben in Zeichenpositionen innerhalb der Datei. Der Quelltext kann daher bei Interesse auf der CD nachgelesen werden.

HyperJumps

Mit Hilfe der nun erworbenen Fähigkeiten, sowohl den Editorinhalt auszulesen als auch zu parsen, ist es nun ein Leichtes, eine Funktion für HyperJumps zu implementieren. Was fehlt, ist nur noch eine Möglichkeit, die Editorposition zu verändern.

Hierzu genügt es theoretisch, dem schon erwähnten *CursorPos*-Property der Editorschnittstelle einen neuen Wert zuzuweisen. In der Praxis ist es nicht ganz so einfach, denn es kommt auch darauf an, welcher Teil der Datei sichtbar ist. Wenn Sie beispielsweise über Delphis CodeExplorer zu einer Methode in den Editor springen, positioniert Delphi den Cursor auf den Beginn der Methode, sorgt aber wenn möglich auch dafür, dass der Methodenkopf am oberen Rand des sichtbaren Bereichs noch sichtbar ist. Hierfür muss auch das *TopPos*-Property angepasst und das *ViewSize*-Property in die Berechnung einbezogen werden.

Diese Details sind im Folgenden in der Prozedur *TUJumpTo* verborgen (ebenfalls in *ToolUtil* implementiert). Außerdem musste das Aufrufformat der bereits gezeigten Prozedur *GetEditorData* etwas angepasst werden, damit *HyperJump* die in *GetEditorData* verwendete *TIEditView*-Variable weiterverwenden kann:

```
procedure TEditorAssistants.HyperJump; { gekürzte Version }
var
  EditorData: TEditorData;
  IteaModule: ModuleHandler;
  V: TIEditView;
  DestModule, CurrentFunc, CurrentSym: ScopeHandler;
  y, x, StartLine: Integer;
begin
  CurrentSym := nil;
  DestModule := nil;
  V := GetCurrentEditorData(EditorData, True);
  IteaModule := GetIteaModule(EditorData.Content, EditorData.FileName);
  // Zuerst die Funktion suchen, innerhalb der sich der Cursor befindet:
  CurrentFunc := IteaModule.
       GetFuncAtPos(EditorData.Position.Line, EditorData.Position.Col);
  // Nun CurrentWord suchen, vom Gültigkeitsbereich dieser Funktion
  // beginnend:
  if Assigned(CurrentFunc) then
    CurrentSym := CurrentFunc.FindSymbol(PChar(EditorData.CurrentWord));
  if Assigned(CurrentSym) then begin // wenn gefunden...
    CurrentSym.GetDeclPos(y, x); // Position des gefundenen Symbols holen
    StartLine := CurrentSym.GetStartLine;
    if StartLine <> 0 then begin
      (* Es handelt sich um eine Methode, deren Rumpf in StartLine
         beginnt. Cursor auf StartLine setzen; in der linken oberen
         Ecke soll nach Möglichkeit die Deklaration der Methode
         sichtbar sein (y, x): *)
      CurrentSym.GetDefPos(y, x);
      TUJumpTo(V, StartLine, 1, y, 1)
    end (* sonst einfach zur Deklaration springen: *)
      else TUJumpTo(V, y, x, y, 1);
  end else
    ShowMessage('Keine Symbolinformationen verfügbar.');
  V.Free; // TIEditView-Schnittstellen müssen manuell freigegeben werden.
  CurrentSym.Free; CurrentFunc.Free; DestModule.Free;
```

```
    IteaModule.FreeWithModule;
    FreeMem(EditorData.Content);
  end;
```

Variablendeklaration

Zur Deklaration einer lokalen Variablen muss das Programm wie schon beim Hyper-Jump zunächst das Wort an der Cursorposition ermitteln. Der Aufruf des in Abbildung A.5 gezeigten Deklarationsdialogs ist altbekannte Delphi-Technik, allerdings bietet der Dialog in seiner bisherigen Version sicher noch viele Erweiterungsmöglichkeiten.

Wenn der Benutzer den Dialog mit OK abschließt, erzeugt das Programm zunächst eine Deklaration in einem String. Falls die aktuelle Funktion noch nicht über lokale Variablen verfügt, muss diese Deklaration auch das *var*-Schlüsselwort enthalten, ansonsten genügt die reine Deklaration der Variablen, gefolgt von einer Zeilenende-Markierung (#13#10), damit die Deklaration in eine eigene Zeile geschrieben wird.

Der fertig erzeugte String kann dann mit *TIEditWriter* leicht in den Quelltext eingefügt werden:

```
W := IEditor.CreateUndoableWriter;
Writer.CopyTo(Pos);
Writer.Insert(PChar(Text));
W.Free;
```

Für die vollständige Implementierung der Deklarationsfunktion ist nur eine Itéa-Methode notwendig, die im letzten längeren Listing noch nicht verwendet wurde: *GetLastDeclPos*, welche die Position der zuletzt deklarierten lokalen Variable zurückliefert und aus der das Programm den in den obigen Zeilen verwendeten *Pos*-Wert berechnet (welcher den Index der Einfügeposition relativ zum Dateianfang angibt).

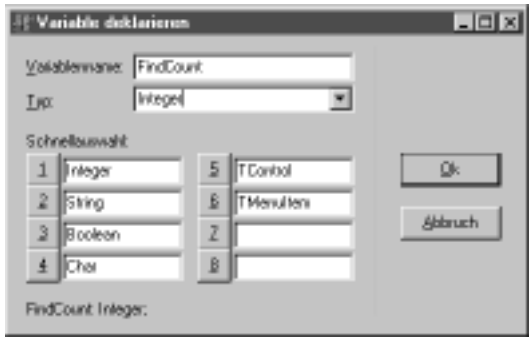


Abbildung A.5: Deklarieren von Variablen mit der zweiten Beispielfunktion von EditorAssistants

Mögliche Fehler beim Ausführen der Funktionen

Wenn Sie beim Aufruf der beiden Funktionen einmal eine Fehlermeldung erhalten, beispielsweise, dass keine Symbolinformationen verfügbar sind, so liegt das entweder daran, dass Sie die Funktion testen wollten und z.B. an einer Eingabeposition aufgerufen haben, an der sich gar kein Bezeichner befindet, oder daran, dass Itéa die Unit nicht vollständig parsen konnte und die Funktionsausführung an Informationsmangel scheitert. Eine provisorische Lösung, wie Sie in diesen Fällen zumindest erfahren können, bis wohin die Unit geparst wurde, soll in A.4 erwähnt werden.

A.3 Neue Funktionen für den Formularentwurf

Viele Experten für den Formularentwurf erzeugen ein komplett neues Formular, so beispielsweise der in Delphi eingebaute Datenbankformular-Experte oder der mit Delphi (ab Professional) mitgelieferte Beispielexperte auf der Seite *Dialoge* der Objektablage. Um in die Objektablage aufgenommen zu werden, muss ein *TIExpert*-Experte den Stil *esForm* haben und virtuelle Methoden implementieren, über die er Delphi Informationen wie das in der Objektablage anzuzeigende Bild liefern kann. Ein *IOTAWizard*-Experte muss entsprechend das Interface *IOTAProjectWizard* unterstützen. Solche Formularexperten haben jedoch zwei Nachteile:

- Um ein neues Formular zu erzeugen, ist es nicht mit einem Aufruf von *TForm.Create* getan. Sie müssen sich komplizierterer Schnittstellen bedienen, um das Formular mitsamt der zugehörigen DFM-Datei und einem PAS-Quelltext zu erzeugen und gleichzeitig Delphi dazu zu bringen, das Formular in das aktuelle Projekt aufzunehmen.
- Für den Benutzer bieten Formularexperten nicht unbedingt eine maximale Flexibilität. So könnte es zum Beispiel erwünscht sein, die vom Datenbankformular-Experten erzeugte Eingabemaske nicht in einem neuen, sondern in einem bestehenden Formular zu erhalten, was jedoch nur erreicht werden kann, indem die Komponenten aus dem vom Experten erzeugten neuen Formular in das bestehende Formular kopiert werden und dann das unnötig erzeugte Formular wieder aus dem Projekt gelöscht wird.

Die Beispielexperten dieses Kapitels gehen daher einen anderen Weg und arbeiten jeweils mit dem aktiven Formular. Wenn Sie an der Erstellung von *esForm* / *esProject*-Experten interessiert sind, können Ihnen bis Delphi 5 die schon erwähnten Demo-Experten (`Demos\Experts\ExptDemo.dpr`) ein Musterbeispiel dafür geben.

Zugriff auf Formulare

Ein Experte, der sich in einem Package befindet, benutzt mit der Delphi-IDE ein gemeinsames *Screen*-Objekt, über dessen *Forms*-Property Sie leicht alle Formulare aufzählen lassen können. Es ist also zum einen nicht unbedingt notwendig, über die Schnittstellen des OpenTools-API Zugriff auf ein Formular zu erhalten; zum anderen wäre es sogar möglich, die Fenster der Delphi-IDE grob zu verändern und z.B. ein Editierfeld über die Menüleiste einzufügen (dies jedoch erst ab Delphi 4, wo die Menüleiste in Wirklichkeit eine ToolBar ist).

Auch die Formulare, die nicht zur IDE gehören und statt dessen in einem Formulareditor editiert werden, lassen sich leicht herausfinden, denn bei diesen ist das *Designer*-Property gesetzt. Ein Beispiel hierzu finden Sie auf der CD: Die Funktion *GetDesignerInfo* aus der Unit *ToolUtil* erzeugt auf diese Weise einen String mit allen zurzeit editierten Formularen. Die beiden Beispiel-Experten verwenden jedoch das OpenTools-API, um in den Besitz einer *TForm* bzw. *TCustomForm*-Variablen zu gelangen.

Formular-Explorer

Der Formular-Explorer (Abbildung A.6) zeigt die Komponenten des aktiven Formulars nach der Besitzhierarchie geordnet an. Er ist zwar in Delphi 6 dank des segensreichen Objekthierarchie-Fensters nicht mehr erforderlich, jedoch kann er ein Beispiel geben, wie sich die im Formularentwurf befindlichen Komponenten von einer IDE-Erweiterung ansprechen lassen. In Kapitel A.2 wurde bereits beschrieben, wie man ausgehend vom ToolServices-Objekt zu einer *TIEditorInterface*-Schnittstelle gelangt. Abbildung A.3 zeigte auch den sehr ähnlichen Weg zu einem *TIFormInterface*, welches im Formular-Explorer benötigt wird. Und zwar besitzt dieses Interface die Methode *GetFormInterface*, welche wiederum ein *TIComponentInterface* für das Formular zurückliefert.

Von diesem *TIComponentInterface* erhält der Formular-Explorer nun über die Methoden *GetControlCount* und *GetControl* die einzelnen Komponenten, die dem Formular direkt untergeordnet sind. Jede dieser Komponenten wird als *TIComponentInterface* angegeben, so dass alle Komponenten rekursiv ermittelt werden können (Details siehe CD).

FormAssistant

Der Formular-Assistent (Unit *FormAssistant*, voreingestelltes Tastenkürzel: Strg+Shift+F) arbeitet ausschließlich mit den in Delphi 4 eingeführten Schnittstellen der Unit *ToolsAPI* und macht außerdem von den schon in A.2 vorgestellten Itéa-Funktionen Gebrauch. Seine Aufgabe besteht darin, eine Eingabemaske für die Elemente eines Records zu erzeugen, ähnlich wie der Datenbankformular-Experte von Delphi eine Eingabemaske für die Felder einer Tabelle erzeugt.

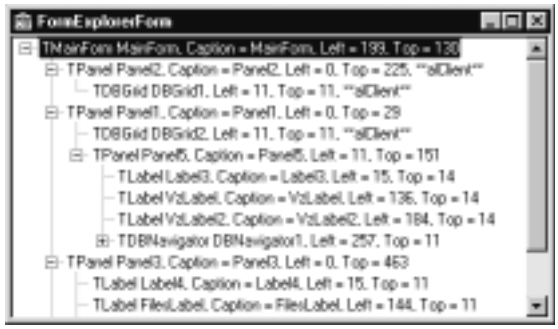


Abbildung A.6: Eine Darstellung der Besitzhierarchie eines Formulars und weiterer Informationen, die in Delphis Objekthierarchie-Fenster nicht angezeigt werden, gibt FormExplorer.pas.

Der als Vorlage dienende Record muss sich in der Unit befinden, die auch für das Formular verwendet wird, denn der Experte arbeitet nur mit dem aktuellen Formular und sucht, wenn er aufgerufen wird, mit Hilfe der Itéa-DLL alle Klassen des aktuellen Quelltextes.

Diese werden dann in der Kombobox eines Dialogs (Abbildung A.7) zur Wahl gestellt. Der Dialog gibt Ihnen auch die Möglichkeit, einige Record- oder Klassenelemente vor der Verwendung in der Maske auszuschließen, indem Sie die Markierungsfelder des ListViews anklicken. Außerdem können Sie über die Editierfunktion des ListViews die Beschriftung verändern, die dem Eingabeelement in der Maske vorangestellt wird. Die weiteren Spalten des ListViews sind nicht editierbar und zeigen den Namen des Record-Elements an (dieser wird auch für die später erzeugte Eingabekomponente verwendet) sowie die Typen der Komponenten, die für dieses Element erzeugt werden. In der derzeitigen Version unterstützt der Formular-Assistent lediglich die folgenden Typen:

- Für *Integer*-Variablen erzeugt er ein *TEdit*-Feld mit *TUpDown*-Komponente.
- Für *Boolean*-Variablen erstellt er eine *TCheckBox*-Komponente.
- Aufzählungstypen repräsentiert er in einer *TRadioGroup*, wobei die einzelnen Schalter der Gruppe die verschiedenen Werte des Aufzählungstyps sind.
- Bei allen anderen Variablentypen wird einfach ein Eingabefeld erzeugt.

Abbildung A.7: Der Dialog des Formular-Assistenten (links) und die nach Abschluss des Dialogs erzeugten Komponenten im Testformular TestFormAssistant

Itéa-Funktionen

Der Formular-Assistent benötigt Funktionen der Itéa-DLL bei verschiedenen Gelegenheiten, begonnen beim Finden aller Records und Klassen der aktuellen Unit, und zum Aufzählen der Elemente der gerade gewählten Klasse. Da sich diese Aufgaben auf eine simple Weise erledigen lassen, sei hier zur Veranschaulichung nur eine davon grob skizziert:

```
// Einfügen der Namen aller Variablen einer Klasse Cls in
// das ListView1 (Auszug aus TMaskSpecDialog.UpdateElementList)
for i := 0 to Cls.GetVarCount-1 do begin
  Vr := Cls.GetVar(i);
  Item := ListView1.Items.Add;
  Item.Caption := Vr.GetName;
  Item.Data := Vr;
  Item.SubItems.Add(Vr.GetName);
  ...
```

Komponenten-Erzeugung

Die größere Herausforderung für den Formular-Assistenten liegt sicher im Erzeugen der entsprechenden Komponenten. Hierzu eignet sich das im Formular-Explorer verwendete *TIComponentInterface* nicht, denn es erlaubt zwar das Erstellen neuer Komponenten, aber nur das Verändern von Properties mit einfachen Typen. Der Formular-Assistent muss jedoch auch die *Items*-Stringliste von *TRadioGroup*- und das *Associate*-Property von *TUpDown*-Komponenten setzen.

Wie erwähnt, verzichtet dieser Experte zugunsten der Unit *ToolsAPI* gänzlich auf die *ToolIntf*-Unit. Die entscheidenden Interfaces heißen diesmal *IOTAFormEditor* und *INTAFormEditor*.

- *IOTAFormEditor* liefert Informationen über die im Formular markierten Komponenten. Falls genau eine *TWinControl*-Komponente ausgewählt ist, fügt der Formular-Assistent die neuen Komponenten als Kindelemente in die gewählte Komponente ein.
- *INTAFormEditor* verschafft dem Assistenten den Zutritt zum bearbeiteten Formular, und zwar findet sich dies im Property *INTAFormEditor.FormDesigner.Form*.

Der Weg, der zu *IOTAFormEditor* und *INTAFormEditor* führt, lässt sich aus der schon bekannten Abbildung A.3 erkennen: Von *IOTAModuleServices* durch das Property *Modules* über *IOTAModule* durch die Methode *GetModuleFileEditor* zu einem *IOTAEditor*-Interface. Dieses kann mit *as* in einen *IOTAFormEditor* (und ebenso gut in einen *INTAFormEditor*) umgewandelt werden, *wenn* es sich um einen Formulareditor und nicht etwa um einen Quelltexteditor handelt. Dies gilt es jedoch vorher zu prüfen. Der Beispielexperte erledigt das wie folgt:

```
OTAModule := ModuleServices.CurrentModule;
// Finden eines OTA/NTAEditors für das aktuelle Formular:
for k := 0 to OTAModule.GetModuleFileCount-1 do begin
  OTAEditor := OTAModule.GetModuleFileEditor(k);
  OTAEditor.QueryInterface(IOTAFormEditor, OTAFormEditor);
  OTAEditor.QueryInterface(INTAFormEditor, NTAFormEditor);
  if Assigned(NTAFormEditor) and Assigned(OTAFormEditor) then break;
end;
```

Daraufhin steht also in *NTAFormEditor.FormDesigner.Form* das zu bearbeitende Formular zur Verfügung, das zum Besitzer der neu erzeugten Komponenten werden wird und im Folgenden unter *ParentForm* gespeichert ist. Zusammen mit *ParentComponent*, der gewählten *TWinControl*-Komponente (falls keine Komponente gewählt ist, weist auch *ParentComponent* auf das Formular), können nun die einzelnen Komponenten erzeugt werden, wie Sie dies von dynamisch erzeugten Komponenten gewohnt sind – ein Beschriftungs-Label beispielsweise wie folgt:

```
NewControl := TLabel.Create(ParentForm);
if Assigned(NewControl) then begin
  NewControl.Parent := ParentComponent;
  NewControl.Left := LeftMargin;
  NewControl.Top := TopMargin + VertPos * RowHeight + 2;
  ...
```

Dabei geben *Left/TopMargin* den Rand an, der zwischen dem Rand der Komponenten und dem der Elternkomponente gelassen wird, *RowHeight* die Höhe einer »Zeile« von Komponenten. Alle drei Werte sind in der derzeitigen Expertenversion fest voreingestellt. *RowHeight* ist der Zähler, der mit jeder neu erzeugten Komponentenzeile erhöht wird.

Auf Itéa-Funktionen angewiesen ist der Experte wieder beim Erzeugen der verschiedenen Einträge einer *RadioGroup*, falls eine Variable einen Aufzählungstypen hat. Die in diesem Typen aufgezählten Werte können mit zwei noch nicht erwähnten Itéa-Funktionen, im Folgenden kursiv dargestellt, in Erfahrung gebracht werden:

```
Choices := TStringList.Create;
for i:=0 to InnerType.GetEnumCount-1 do
  Choices.Add(InnerType.GetEnum(i).GetName);
//Choices kann nun an dem Items-Prop. der RadioGroup zugewiesen werden.
```

So liefert *GetEnum(i).GetName* beispielsweise für den Aufzählungstypen *(red, green, blue*) nacheinander die Strings »red« , »green« und »blue« .

Die Formular-Unit

Dies waren bereits die entscheidenden Teile des Quelltextes. Es genügt tatsächlich für jede neu zu erzeugende Komponente, sie einfach mit *Create(ParentForm)* zu initialisieren, um sie korrekt in Delphis Formulareditor einzubinden. Dank der VCL-internen Abläufe beim Einfügen von Komponenten in den Besitz eines Formulars erfährt die Delphi-IDE von jeder Einfüge-Operation, erzeugt automatisch die passenden Variablendeklarationen in der zugehörigen Object-Pascal-Unit und merkt sich natürlich auch, dass das Formular verändert wurde, so dass auch die Sicherheitsabfrage vor dem Schließen des Formulars weiterhin funktioniert.

Sollten Sie durch einen Experten einmal andere Änderungen an einem Formular vornehmen, ohne dass Delphi etwas davon merkt (z.B. indem Sie nur einfache Properties verändern), können Sie die bereits in Kapitel 6.4.4 erwähnte Methode *Modified* der *IFormDesigner*-Schnittstelle verwenden, um die IDE über die Änderung zu unterrichten.

A.4 Eine CodeExplorer-Imitation

Als Letztes sei hier noch der CodeExplorer des Packages *IteaExperten* erwähnt (Abbildung A.8). Wenn das Package erfolgreich installiert ist, bringt Sie ein Druck von Strg + Shift + I oder die Auswahl des Menüpunktes BEARBEITEN | ITÉA-CODEEXPLORER in das Fenster des Explorers, das in der obersten Ebene eines TreeViews zunächst alle Klassen und Interfaces der Unit darstellt. Um auch Variablen, Funktionen und Typen zu sehen, müssen Sie zuerst die entsprechenden Ordner-Knoten expandieren. Der Inhalt der Klassen und Interfaces wird (ohne die von Delphis CodeExplorer gewohnten Möglichkeiten zur Konfiguration) in die Unterkategorien *Funktionen* und *Variablen* aufgeteilt. Ein Doppelklick auf einen Eintrag bringt Sie bei Funktionen und Methoden direkt zur jeweiligen Implementation, bei Variablen und Typen zur jeweiligen Deklaration.

In der derzeitigen Version wird der Explorer übrigens nicht aktualisiert, wenn Sie eine andere Seite in Delphis Editor aufschlagen. Dies verhindert, dass Sie nach dem Umschlagen von Seiten eventuell einige Sekunden auf Itéa warten müssen. Mit Hilfe der *IOTAEditorNotifier*-Schnittstelle ließe sich jedoch auch diese Funktion leicht nachrüsten.

Der unvollendete Status der Itéa-DLL bringt es außerdem mit sich, dass nicht immer der gesamte Inhalt einer Unit im Itéa-CodeExplorer dargestellt werden kann. Kommt es beispielsweise während des Parsens zu einem schweren Fehler, beendet Itéa einfach das Sammeln der Symbolinformationen. Es kann auch möglich sein, dass ein Doppelklick Sie nicht zur Implementation, sondern zur Deklaration der Funktion führt. In diesem Fall ist der Parse-Fehler zwischen der Deklaration und der Implementation aufgetreten, so dass Itéa nur die Position der Deklaration kennt, aber nicht die der Implementation. Mit einem Doppelklick auf die Statuszeile des Explorers erhalten Sie eine Liste der von Itéa ausgegebenen Fehlermeldungen. Sie gibt mit ihrer letzten Fehlermeldung zumindest einen Hinweis darauf, bis wohin Itéa die Unit parsen konnte.

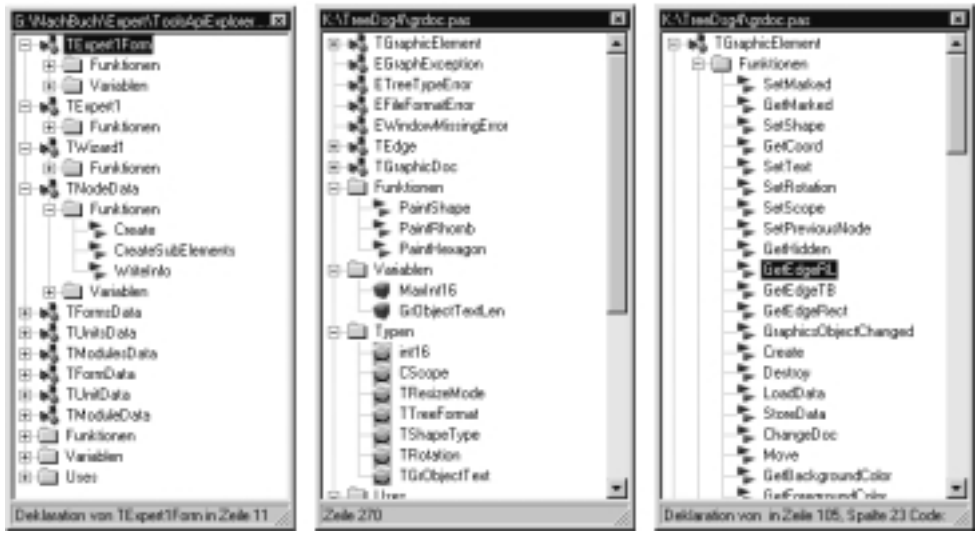


Abbildung A.8: Dieser CodeExplorer läuft auch mit der Standard-Version von Delphi und birgt noch einige Erweiterungsmöglichkeiten.

Für Besitzer von Delphi-Versionen ohne eigenen CodeExplorer ist es vielleicht der interessanteste Teil des Packages – da seine Implementation aber noch einfacher als die in A.2 beschriebenen Editorfunktionen ist, soll sie hier nicht genauer beschrieben werden. Sie besteht im Wesentlichen aus einer massiven Verwendung der *ScopeHandler*-Methoden *GetClass*, *GetFunc*, *GetVar*, *GetType* zur Aufzählung der in der Unit enthalte-

nen Bezeichner und den entsprechenden Aufrufen von *TTreeNodes.AddChildObject* zur Konstruktion der TreeView-Knoten. Wird einer der Knoten doppelt angeklickt, so lässt sich die anzuzeigende Zeile mit der *ScopeHandler*-Methode *GetStartLine* in Windeseile herausfinden. Damit der Cursor nicht nur auf diese Zeile gesetzt wird, sondern auch in jedem Fall sichtbar ist, muss auch das *TopPos*-Property von Delphis *TIEditView*-Schnittstelle gesetzt werden. Hierzu liefert *ScopeHandler.GetDeclPos* die Zeile, in der sich der Kopf der angesprungenen Methode befindet, denn diese soll im Idealfall als erste Zeile im Editor sichtbar sein (und nicht etwa das *begin* des Methodenrumpfes, auf das schon die *CursorPos* gesetzt wird).

B VCL-Hierarchiegrafiken des TreeDesigners

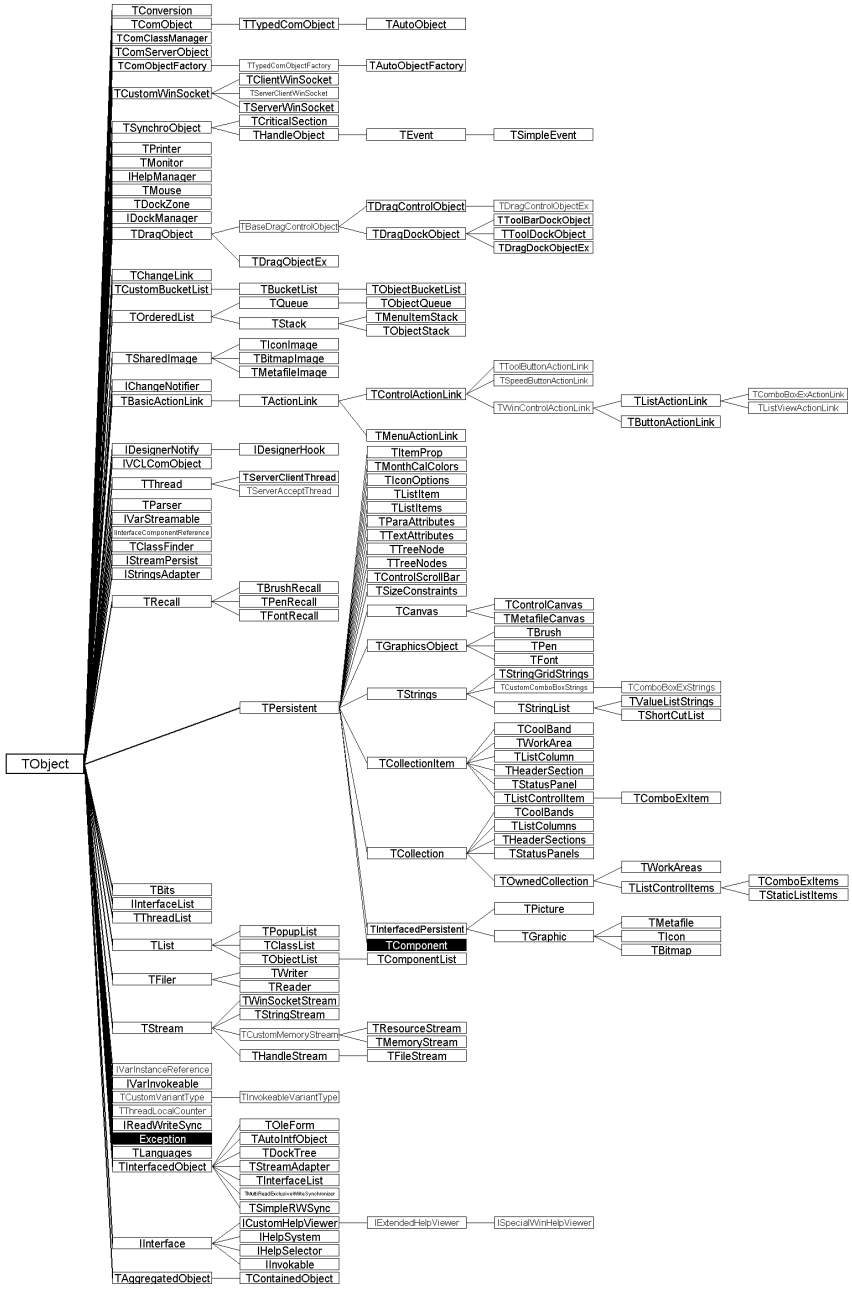


Abbildung B.1: Die »Wurzelregion« der VCL-Klassenhierarchie, aus Platzgründen hier beschränkt auf eine Auswahl von Klassen der Personal-Ausgabe (für eine umfassende Übersicht bedarf es eines großen Posters, wie es von Borland mitgeliefert wird).

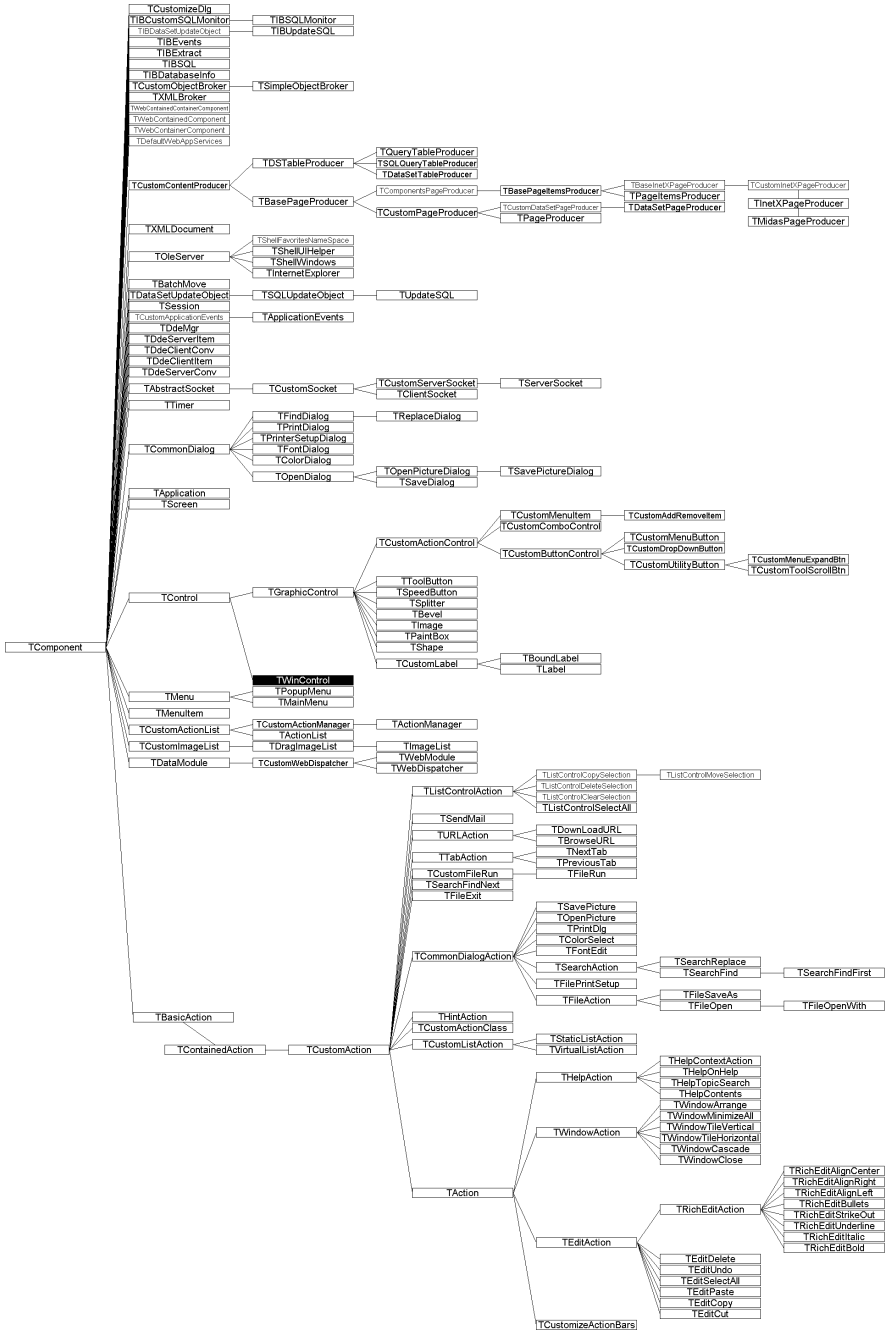


Abbildung B.2: TComponent bildet die Grundlage für alle Komponenten, sowohl visuelle als auch nicht-visuelle (hier auch mit Klassen aus Professional- und Enterprise-Version).

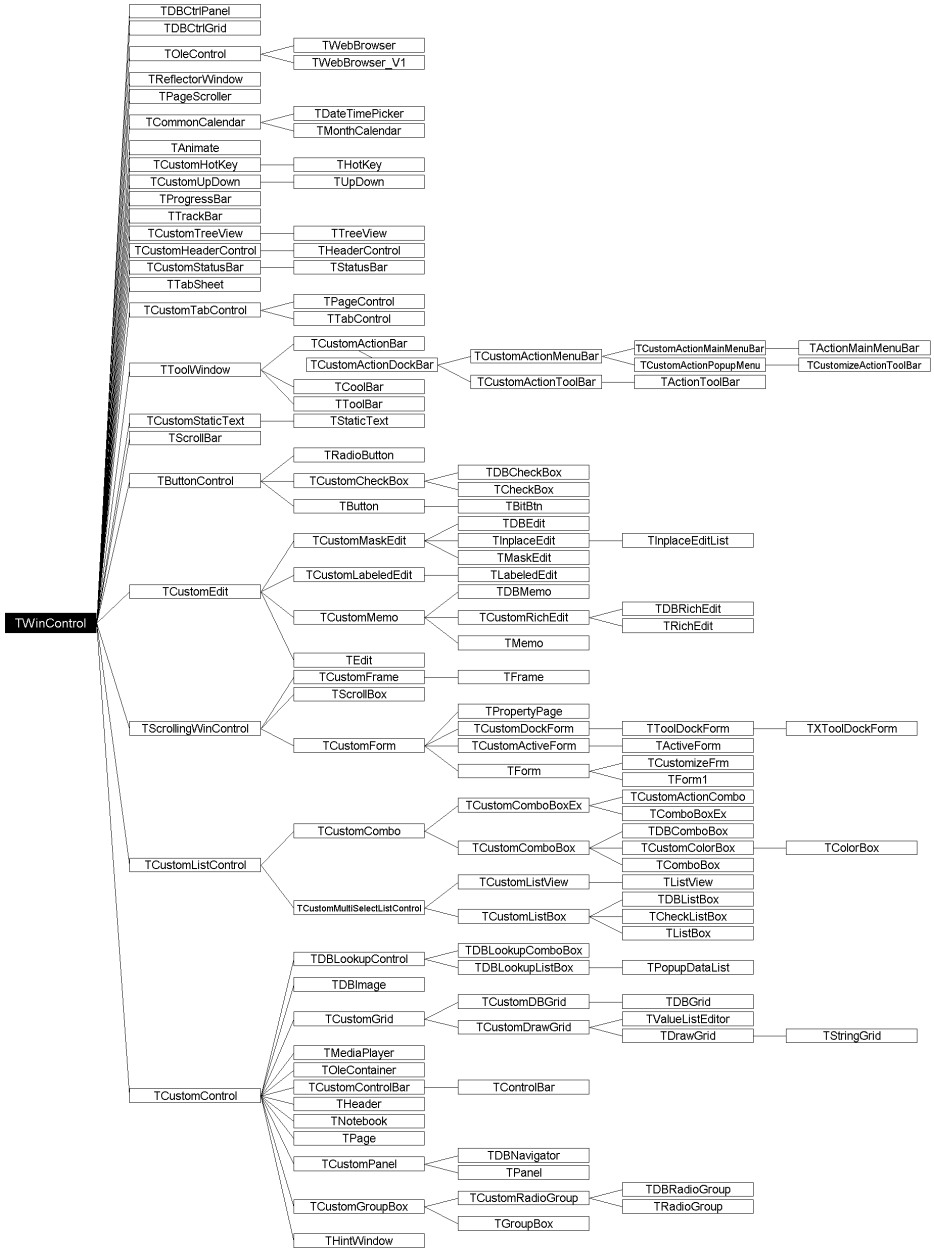


Abbildung B.3: Der in Abb. B.2 ausgeblendete Teilbaum der TWinControl-Komponenten (alle Delphi-Ausgaben; für eine Beschränkung auf die Personal-Ausgabe siehe Abbildung 3.3).

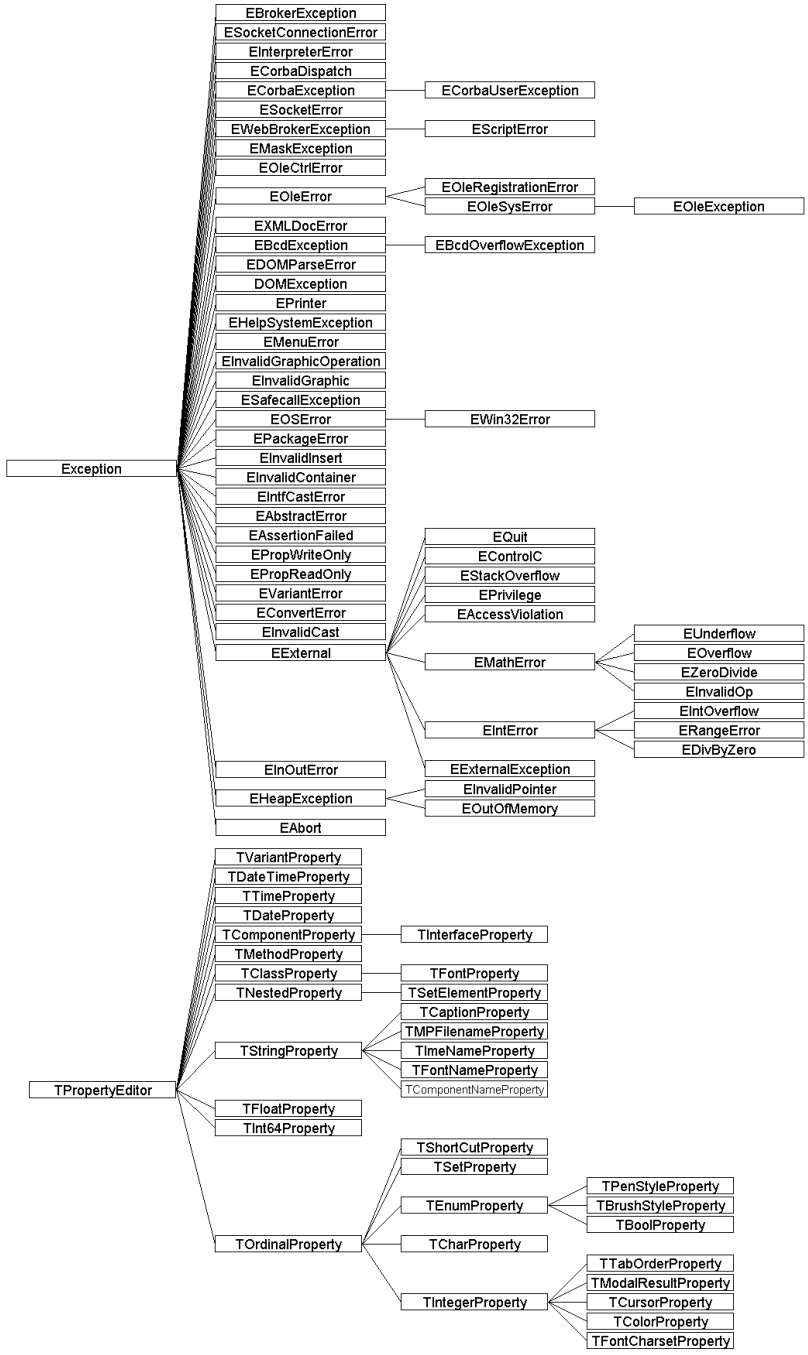


Abbildung B.4: Die Teilbäume der Klassen Exception (in Abbildung B.1 ausgeblendet) und TPropertyEditor (weitere ToolsApi-Klassen siehe Abbildung A.1).

Anhang C: Rezeptverzeichnis

1 Benutzerschnittstelle: Entwurf

2 Formulare zur Laufzeit

3 Formulare: Management mehrerer Formulare

4 Benutzerschnittstelle: Komponenten

5 Features

6 Benutzereingabe mit Tastatur und Maus

Tastatur und Maus

Tastatur

Maus

7 Grafikausgabe und Drucken

Komponenten und Klassen

Steuerung des Ablaufs

Zeichenoperationen

Drucken

8 Ein- und Ausgabe: Dateien

Dateien allgemein

Konfigurationsdateien

9 Programmiertechnik

Datenstrukturen

OOP

Erweitertes Pascal

Debugging

Programmfluss

10 System

Integration in die Windows-Umgebung

Nutzung verschiedener System-Merkmale

COM-Themen

siehe auch

11 Datenbanken – ausgewählte Themen

12 Komponenten – ausgewählte Themen

Stichwortverzeichnis

Das Stichwortverzeichnis enthält einige spezielle Punkte, auf die hier besonders hingewiesen werden soll:

- Klassenelemente sind grundsätzlich als Unterpunkt der Klassen aufgeführt, durch die sie definiert werden. Da jedoch die Elemente der grundlegenden Klassen wie beispielsweise *TControl* und *TWinControl* beim Einstieg in Delphischwer zu unterscheiden sind und weitere Elemente in verschiedenen Klassen unabhängig eingeführt werden, wurde das Stichwort *VCL - allgemeine Klassenelemente* gewählt, um diesen Elementen ein gemeinsames Dach zu geben. Sollten Sie weder dort noch bei einer speziellen Klasse fündig werden, empfiehlt sich das Nachschlagen unter *TControl* oder *TWinControl*oder der Aufruf von Delphis Online-Hilfe, die meistens die Klassen auflistet, in denen das gesuchte Klassenelement vorkommt.
- Drei Stichwörter fassen andere Stichwörter zusammen, die auch als eigenständige Punkte des Verzeichnisses aufgeführt werden: *VCL* fasst die Klassen zusammen, die an einer bestimmten Stelle ausführlich beschrieben sind, während Sie die Klassen, die eher kurz erwähnt werden, im Verzeichnis nur als eigenständige Stichwörter finden. In *Beispielprogramme* und *Windows-API* finden Sie schließlich eine Übersicht über die Stellen, an denen namentlich genannte Beispielprogramme besprochen sind, bzw. über die verwendeten Windows-API-Funktionen.
- Alle Stichwörter, die speziell dem Begriff *Steuerelemente* zuzuordnen wären, wurden der Eindeutigkeit halber dem Punkt *Komponenten* untergeordnet. Weitere zusammenfassende Stichwörter sind *IDE* und *Datenbanken*.
- Falls Sie sich intensiver mit der Beispielanwendung *TreeDesigner* beschäftigen und Informationen zu einer der darin definierten Methoden suchen, sollten Sie unter den Punkten *TGraphicDoc*, *TGraphicElement* und *TDocumentForm* nachschlagen.

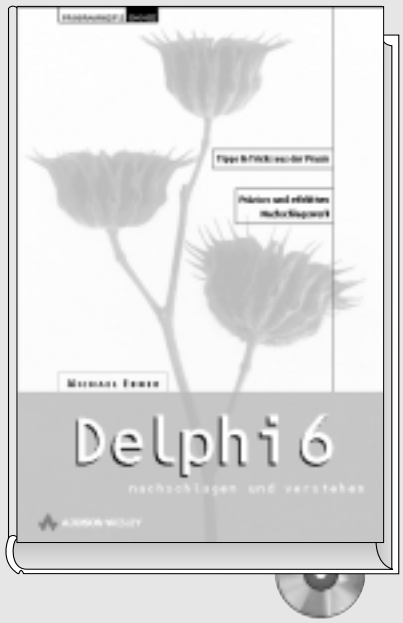
C

R

U

V

T H E S I G N O F E X C E L L E N C E



Delphi 6

nachschlagen und verstehen

Michael Ebner

Als präzises und effektives Nachschlagewerk konzentriert sich dieses Buch auf die wesentlichen, bei der täglichen Arbeit mit Delphi 6 auftretenden Fragen und Probleme, so dass Sie innerhalb kürzester Zeit die Lösung dafür finden und weiterarbeiten können. Viele Tips und Tricks aus der Praxis sorgen dafür, dass das Wissen konkret, anschaulich und realitätsbezogen vermittelt wird.

Programmer's Choice

ca. 950 Seiten, 1 CD-ROM, ET 07-2001
DM 119,90/öS 875,00/sFr 108,00
ISBN 3-8273-1786-X

www.addison-wesley.de

ADDISON-WESLEY

THE SIGN OF EXCELLENCE

Delphi 6

Datenbankprogrammierung

Michael Ebner

Dieses Buch vermittelt kompetent das nötige Wissen für eine erfolgreiche Datenbankprogrammierung mit Delphi 6. Alle Datenbank-Komponenten werden anhand kurzer Beispiele vorgestellt. Es folgt eine Referenz aller Eigenschaften, Methoden und Ereignisse. Anschaulich wird die Erstellung professioneller Datenbank-Applikationen vorgeführt. Daran anschließend zeigen ausführliche Beispielprojekte exemplarisch auf, wie man ein komplexes Gesamtprojekt erfolgreich planen und ausführen kann.

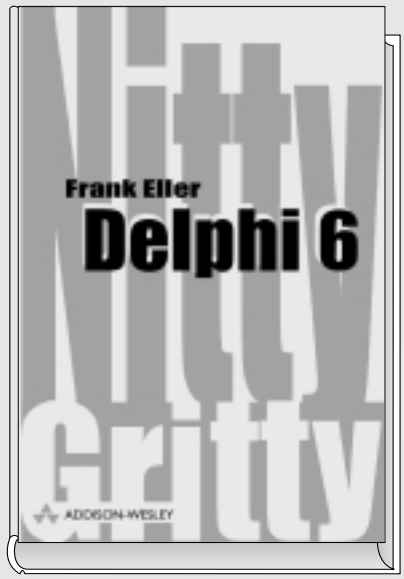
Programmer's Choice

ca. 780 Seiten, 1 CD-ROM, ET 07-2001
DM 119,90/öS 875,00/sFr 108,00
ISBN 3-8273-1785-1

www.addison-wesley.de

ADDISON-WESLEY

THE SIGN OF EXCELLENCE



Delphi 6

Frank Eller

Delphi 6 auf den Punkt gebracht. Sie erhalten einen Überblick über die Komponenten und alle wichtigen Neuerungen in Delphi 6 sowie über verschiedene nützliche Programmiertechniken und die wichtigsten Funktionen und Prozeduren. Danach wenden Sie sich fortgeschritteneren Themen wie z.B. der Datenbankprogrammierung, der Programmierung von Drag&Drop-Operationen sowie dem Arbeiten mit Texten und Grafiken, Threads und Dateien zu. Tipps und Tricks zur Programmierung mit Delphi runden das Ganze ab.

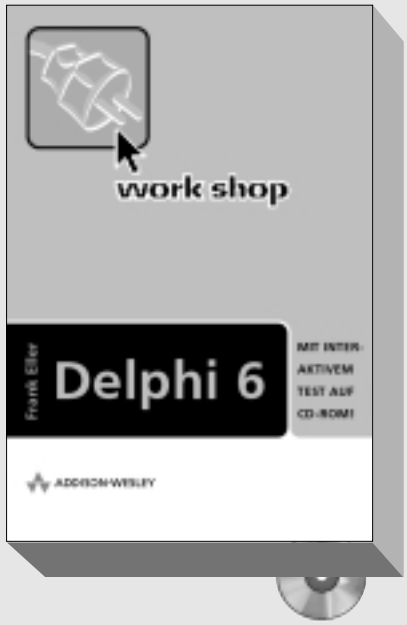
Nitty Gritty

ca. 413 Seiten, ET 07-2001
DM 25,00/sFr 183,00/sFr 23,00
ISBN 3-8273-1775-4

www.addison-wesley.de

ADDISON-WESLEY

THE SIGN OF EXCELLENCE



Workshop Delphi 6

Frank Eller

Möchten Sie Ihr Delphi-Wissen vertiefen? In diesem Workshop finden Sie viele Übungen zu wichtigen Aspekten von Object Pascal und der Programmierung mit Delphi 6. Unter anderem werden Klassen, Komponententenentwicklung, Rekursionen, Zeiger, Felder und verschiedene Algorithmen angespochen. Eine Fülle von Tipps und Tricks zu den einzelnen Aufgaben und ausführliche Beispiellösungen helfen bei der Bewältigung von Programmieraufgaben. Mit Online-Test auf CD.

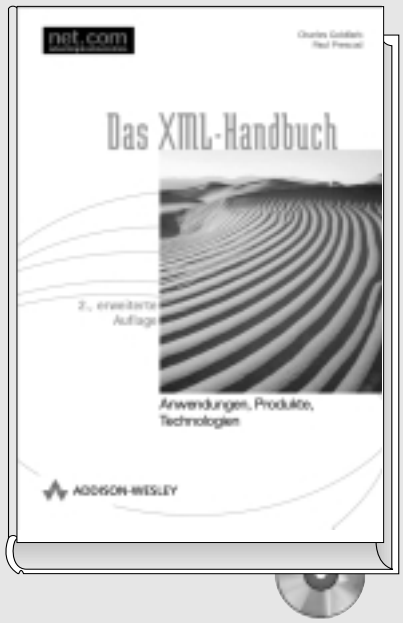
workshop

ca. 350 Seiten, 1 CD-ROM, ET 07-2001
DM 69,90/öS 510,00/sFr 63,00
ISBN 3-8273-1799-1

www.addison-wesley.de

ADDISON-WESLEY

THE SIGN OF EXCELLENCE



Das XML-Handbuch

Anwendungen, Produkte, Technologien

Charles F. Goldfarb, Paul Prescod

Vom SGML-Erfinder! Dieses Buch führt Sie umfassend in das Thema XML ein. Anwendungen, Produkte und Technologien stehen im Mittelpunkt. Nach einer Einführung in das Konzept von XML lernen Sie im zweiten Teil anhand zahlreicher Beispielprojekte die Einsatzmöglichkeiten z.B. in den Bereichen E-Commerce und Content Management kennen. Der dritte Teil bietet weiterführendes Wissen und behandelt auch verwandte Technologien wie XSL, XSLT, XLink oder topic maps. Auf der CD: u.a. IBM alphaWorks suite.

net.com

912 Seiten, 1 CD-ROM
DM 119,90/öS 875,00/sFr 108,00
ISBN 3-8273-1712-6

www.addison-wesley.de

ADDISON-WESLEY

THE SIGN OF EXCELLENCE

COM-Komponenten-Handbuch

Automatisierte Administration und Systemprogrammierung unter Windows

Holger Schwichtenberg

Dieses neue Buch dokumentiert zahlreiche COM-Komponenten (ADO, ADSI, CDO, WMI, DMO, FSO, MOM, u.a.), die den programmgesteuerten Zugriff auf Windows (einschl. 2000, XP sowie .NET) und BackOffice-Anwendungen ermöglichen. Die über 500 Code-Beispiele sind sowohl in VBScript als auch unter VB 6.0 lauffähig. Für Administratoren und Visual Basic-Entwickler gleichermaßen geeignet.

win.tec

ca. 450 Seiten, 1 CD-ROM
DM 89,90/öS 656,00/sFr 78,00
ISBN 3-8273-1936-6

www.addison-wesley.de

ADDISON-WESLEY