L U K U 1 0

COM-asiakkaat

Tässä luvussa

Tässä luvussa opit lisää siitä, kuinka sovellukset ja komponentit toimivat COM-palvelinkomponenttien asiakkaina ja kuinka ne hyödyntävät komponenttien tarjoamia palveluja. Opit, kuinka Microsoft Visual C++:n kääntäjä yksinkertaistaa COM-asiakkaan koodin kirjoittamista. Opit myös, kuinka tehdään COM-objekteja, jotka voivat sisältää toisia COM-objekteja.

Ennen kuin aloitat

Ennen kuin aloitat tämän luvun läpikäymisen sinun tulisi lukea luvut 2-9 ja suorittaa niihin liittyvät tehtävät.

Oppitunti 1: COM-asiakassovellukset

Tällä oppitunnilla luodaan yksinkertainen sovellus, joka käyttää luvun 9 harjoituk-sissa tehtyä Encoder COM-komponenttia. Sovellus käyttää MIDL-kääntäjän EncodeServer.idl-tiedostosta luomia header-tiedostoja. Oppitunnin jälkipuoliskolla kerrotaan Visual C++ -kääntäjän ominaisuukista, jotka tukevat COM-asiakassovellusten ja komponenttien luomista ja luodaan EncodeClient-sovellus uudelleen näiden menetelmien avulla.

Tämän oppitunnin jälkeen:

- Tiedät, kuinka asiakassovellus voi käyttää MIDL-kääntäjän generoimia header-tiedostoja.
- Tunnet Visual C++ COM -kääntäjä tuen ominaisuudet.
- Tiedät, kuinka tyyppikirjasto tuodaan projektiin.

Oppitunnin arvioitu kesto: 30 minuuttia

COM-palvelimen Header-tiedostot

Luvussa 9 opit, kuinka MIDL-kääntäjä muodostaa header-tiedostot, joiden kautta COM-palvelimen rajapinta ja GUID-määrittelyt saadaan C ja C++ -asiakkaiden käyttöön. Nämä tiedostot on generoitu Encoder-komponentille ja ne ovat nimiltään EncodeServer.h ja EncodeServer_i.c.

EncodeServer_i.c-tiedosto sisältää seuraavan koodin, jossa määritellään Encoder-komponentin määrittämät GUID:t helposti luettavina (ja kirjoitettavina) vakioina:

```
const IID IID_IEncoder =
{0x69F4B926,0x6641,0x11D3,{0x93,0x4B,0x00,0x80,0xC7,0xFA,0x0C,0x3E}};

const IID LIBID_ENCODESERVERLib =
{0x69F4B91A,0x6641,0x11D3,{0x93,0x4B,0x00,0x80,0xC7,0xFA,0x0C,0x3E}};

const CLSID CLSID_Encoder =
{0x69F4B917,0x6641,0x11D3,{0x93,0x4B,0x00,0x80,0xC7,0xFA,0x0C,0x3E}};
```

The IID and CLSID types are defined earlier in the file as C structures that represent a GUID.

At the heart of the EncodeServer.h file is the following C++ declaration (comments have been removed for clarity):

```
MIDL_INTERFACE("69F4B926-6641-11D3-934B-0080C7FA0C3E")
IEncoder : public Iunknown
```

```
{
    public:
    virtual HRESULT STDMETHODCALLTYPE EncodeString(
        const BSTR instring, BSTR __RPC_FAR *outstring) = 0;

    virtual HRESULT STDMETHODCALLTYPE get_Key(
        short __RPC_FAR *pVal) = 0;

    virtual HRESULT STDMETHODCALLTYPE put_Key(
        short newVal) = 0;
};
```

Tämä koodi määrittelee abstraktin luokan **IEncoder** (joka on periytetty abstraktista luokasta **IUnknown**), joka sisältää **IEncoder** COM-rajapinnan julkistamia metodeja vastaavat jäsenmetodit. **MIDL_INTERFACE-**makro yhdistää Microsoftin erityisen **__declspec(uuid())**-määrittäjän avulla rajapinnan GUID:n luokan kanssa. Avainsanaa **__uuidof()** voidaan tällä tavoin soveltaa luokkaan liitetyn vakio-GUID:n esille saamiseen.

Asiakasohjelmat käyttävät näihin header-tiedostoihin talletettuja määrittelyjä apuna luodessaan **IEncoder**-luokkaan osoittimia, jotka voidaan välittää funktiolle **CoCreateInstance**(), seuraavassa esimerkissä esitetyllä tavalla:

```
IEncoder * pServer;
HRESULT hr = ::CoCreateInstance(CLSID_Encoder, NULL,
    CLSCTX_INPROC_SERVER,
    IID_IEncoder,
    (void **) &pServer);
```

Jos tämä fuktion kutsu onnistuu luomaan **Encoder** COM-objektin, pServer-muuttuja ottaa vastaan osoittimen COM-objektin vtableen. **pServer**-osoitinta voidaan tämän jälkeen käyttää kutsuttaessa **IEncoder**-rajapinnan metodeja.

Jotta saisimme paremman käsityksen EncodeServer.h ja EncodeServer_i.c header-tiedostojen käytöstä, teemme yksinkertaisen konsoliohjelman, EncodeHello, joka esittää **Encoder** COM-objektin avulla koodatun “Hello World”- tekstin.

► **EncodeHello-sovelluksen luominen**

1. Valitse Visual C++:n **File**-valikosta **New**. Valitse projektin tyypiksi **Win32 Console Application** ja kirjoita projektin nimeksi **EncodeHello**. Napauta **OK**.
2. Valitse Win32 Console Application Wizardin vaiheessa 1 **A simple application** ja napauta **Finish**. Luo projekti napauttamalla **OK**.

3. Kopioi Resurssienhallinnassa tiedostot EncodeServer.h ja EncodeServer_i.c EncodeServer-projektin kansiosta EncodeHello-projektin kansioon.
4. Etsi ja avaa FileViewissä EncodeHello.cpp-tiedosto. Lisää tiedoston alkuun seuraavat rivit:

```
#include <iostream.h>
#include "EncodeServer.h"
#include "EncodeServer_i.c"
```

5. Lisää **Main()**-funktion runkoon koodia niin, että se näyttää seuraavalta:

   (Tämä koodi on asennettu oheisrompulta tiedostoon CH10_01.cpp.)

```
int main(int argc, char* argv[])
{
    ::CoInitialize(NULL);

    IEncoder * pServer;

    HRESULT hr = ::CoCreateInstance(CLSID_Encoder, NULL,
        CLSCTX_INPROC_SERVER,
        IID_IEncoder,
        (void **) &pServer);

    if(SUCCEEDED(hr))
    {
        short nKey = 1;
        cout << "Enter a code key between -5 and 5: ";
        cin >> nKey;

        wchar_t wstrHello[16] = L"Hello World!";
        BSTR bstrHello = ::SysAllocString(wstrHello);
        BSTR bstrCodedHello;

        HRESULT hr = pServer->put_Key(nKey);
        if(FAILED(hr)) goto ComError;

        hr = pServer->EncodeString(bstrHello, &bstrCodedHello);
        if(FAILED(hr)) goto ComError;

        char strOut[16];
        wcstombs(strOut, bstrCodedHello, 16);
        cout << "\n" << strOut << "\n\n";

        ComError:
            if(FAILED(hr)) cout << "COM Error" << "\n\n";

            ::SysFreeString(bstrHello);
            ::SysFreeString(bstrCodedHello);
```

```
        pServer->Release();
    }
    ::CoUninitialize();

    return 0;
}
```

CoInitialize()-funktio kutsuu **CoInitializeEx()**-funktiota parametrillä *COINIT_APARTMENTTHREADED*. Luvun 8 oppitunnilla 4 opit, kuinka **CoInitializeEx()**-funktiota täytyy kutsua COM-kirjastojen alustamiseksi nykyisessä säikeessä ja luotavan COM-objektin säiemallin määrittämiseksi. Jokaista **CoInitialize()** tai **CoInitializeEx()** -funktion kutsua kohden täytyy suorittaa **CoUninitialize()**-funktion kutsu, joka sulkee nykyisen säikeen COM-kirjaston.

EncodeHello.cpp-tiedoston koodi on hyvin suoraviivainen — luodaan **Encoder** COM -objektin ilmentymä, **Key**-ominaisuudelle asetetaan käyttäjän antama ar-vo, ja “Hello World!”-merkkijono koodataan kutsumalla **EncodeString()**-metodia.

Huomaa funktion loppupuolella oleva **IUnknown::Release()**-kutsu. Luvun 8 oppitunnilla 1 opit, että käytettäessä **CoCreateInstance()**-funktiota rajapinnan osoittimen hankkimiseen, palvelimen viittauslaskimen arvoa kasvatetaan. Viittauslaskimen arvon pienentäminen rajapinnan osoittimen käytön loputtua on asiakkaan vastuulla ja se suoritetaan **Release()**-kutsulla.

6. Varmista EncodeHello-sovelluksen toimivuus kääntämällä ja käynnistämällä.

Visual C++ -kääntäjän tuki COM:lle

Microsoft esitteli Visual C++ 5.0:ssa joukon luokkia ja C++-kielen laajennuksia, jotka yksinkertaistavat COM-asiakasohjelmien tekemistä. Näihin kuuluvat **_com_ptr_t**, joka on COM-rajapinnan osoittimen kapseloiva osoitinluokka ja **_bstr_t-** ja **_variant_t**-luokat, jotka kapseloivat **BSTR-** ja **VARIANT**-tietotyypit. Mukana on myös COM-poikkeusluokka **_com_error**. Saat nämä luokat käyttöösi sisällyttämällä (#include-komennolla) koodiisi comdef.h-tiedoston. Toinen uusi Visual C++ 5.0:ssa tullut uusi piirre on #import-direktiivi, joka generoi C++-headertiedostoja COM-palvelinobjektin tyyppikirjaston pohjalta. Näissä headertiedostoissa käytetään laajasti **_com_ptr_t**-luokkaa ja käytetään **_bstr_t** ja **_variant_t** -luokkia siellä, missä **BSTR-** ja **VARIANT**-tietotyypit ovat käytössä.

Visual C++:n COM-kääntäjän ominaisuudet on kuvattu tarkemmin seuraavissa jaksoissa.

_com_ptr_t

_com_ptr_t-luokka on malliluokka, joka kapseloi COM-rajapinnan osoittimen. Luokka sisältää jonkin verran lisäkoodia, joka yksinkertaistaa viittausten laskemista. **_com_ptr_t**-objekti kutsuu kapseloidun rajapinnan **IUnknown::AddRef()** ja **IUnknown::Release()** -metodeja puolestasi varmistaakseen, että rajapinnan sisältämien COM-objektien elinaikaa käsitellään oikein. **AddRef()**-funktiota kutsu-taan automaattisesti, kun **_com_ptr_t**-objekti luodaan olemassaolevan rajapinnan osoittimen kopiona, ja **Release()**-funktiota kutsutaan automaattiseti, kun **_com_ptr_t**-objektin käyttö loppuu.

Vaikka tämä "älykäs" toiminto tekeekin lähdekoodista luettavampaa, älä anna **_com_ptr_t**-luokan vaivuttaa itseäsi väärään turvallisuuden tunteeseen. Sinun tulisi olla koko ajan selvillä koodissa käytetyistä COM-objekteista, ja tietää koska **AddRef()** ja **Release()** -funktioita tulee kutsua — vaikka älykäs osoitin tekisikin sen puolestasi.

Yksinkertaisin tapa **_com_ptr_t**-mallin alustamiseen tiettyyn rajapintatyyppiin on luoda **_com_ptr_t**-objekti **_COM_SMARTPTR_TYPEDEF**-makron avulla. Tämä makro ottaa määrätyn rajapinnan nimen ja yksilöllisen GUID:n ja määrittelee **_com_ptr_t**, jonka nimeksi tulee rajapinnan nimi varustettuna etuliitteellä *Ptr*. Esi-merkiksi tiedostossa, joka sisältää EncodeServer.h ja EncodeServer_i.c header-tiedostot, seuraava rivi määrittelee**_com_ptr_t** sovituksen **IEncoderPtr**:lle:

```
_COM_SMARTPTR_TYPEDEF(IEncoder,   __uuidof(IEncoder));
```

Alustetun tyypin ilmentymät voivat tämän jälkeen kutsua **com_ptr_t**:n jäsenfunktiota **CreateInstance()**, jonka avulla ne saavat osoittimen COM-palvelimelle seuraavasti:

```
IEncoderPtr pEnc ;
pEnc.CreateInstance(CLSID_Encoder);
```

Osoitinta voidaan tämän jälkeen käyttää rajapinnan metodien kutsumiseen hyödyntämällä **com_ptr_t**:n ylikuormittamaa ->-operaattoria seuraavasti:

```
int n = 3;
HRESULT hr = pEnc->put_Key(n);
```

Huomaa, että **_com_ptr_t**:n jäsenfunktioita kutsutaan käyttämällä **piste**-operaattoria (kuten kutsuttaessa **CreateInstance()**) ja että rajapinnan metodeja kutsutaan käyttämällä ylikuormitettua ->-operaattoria (kuten **put_Key()**-metodia kutsuttaessa juuri nähtiin).

On myös mahdollista luoda **_com_ptr_t**-objekteja olemassaolevasta COM-rajapinnan osoittimesta tai kopioimalla toisesta **_com_ptr_t**-objektista. Visual C++:n ohje antaa tarkemmat ohjeet **AddRef()** ja **Release()** -metodien käytöstä näissä tapauksissa.

_bstr_t

_bstr_t-objekti kapseloi **BSTR**-tietotyypin. Luokka huolehtii resurssien allokoinnista ja deallokoinnista käyttämällä **SysAllocString()** ja **SysFreeString()** -funktioiden sisäisiä kutsuja ja käyttää viittausten laskentaa käyttääkseen muistia tehokkaasti. Luokka sisältää joukon operaattoreita, joiden avulla voit käyttää **_bstr_t**-objektia yhtä helposti kuin **CString**-objektia. Yksi operaattori, jota ei kuitenkaan tarjota käyttöön, on **& "address of"** -operaattori, joten et voi välittää **_bstr_t**-objektin osoitetta funktioille, jotka odottavat **BSTR***-objektia.

_variant_t

_variant_t-objekti on ohut kääre **VARIANT**-tietotyypille. Luokka hoitaa kapseloidun **VARIANT**-tyypin luomisen ja tuhoamisen käyttämällä sisäisiä API-funktioiden **VariantInit()** ja **VariantClear()** -kutsuja. **_variant_t**-objekti sisältää muodostimen ja operaattorit, joiden avulla **VARIANT**-tyypin luominen ja käsitteleminen on helppoa.

_com_error

_com_error-objekti edustaa COM:n poikkeuksia ja kapseloi lähes kaikkien COM-rajapintojen metodien palauttamat HRESULT-koodit. **_com_error**-objektin voidaan nostaa **_com_raise_error()**-funktiolla. Lisätietoja **_com_error**-objektista saat luvun 13 oppitunnilla 2.

Import-direktiivi

Esikäsittelijän #import-direktiivin avulla voidaan generoida C++ header-tiedosto COM-objektista ja sen rajapinnoista tyyppikirjaston pohjalta. Tämä on hyödyllistä, jos et pääse käsiksi MIDL-luomaan header-tiedostoon; tyyppikirjasto on lähes aina käytettävissä, koska se on tavallisesti sidottu palvelimen DLL tai EXE -tiedostoon. #import-esikäsittelijä luo myös joukon älykkäitä osoittimia, joiden avulla voidaan käsitellä COM-objektin rajapintoja.

Kun käytät #import-direktiiviä, Visual Studion C-kääntäjän esikäsittelijä luo kaksi otsikkotiedostoa, joissa tyyppikirjaston sisältö on rakennettu uudelleen C++-lähdekoodiksi. Pääotsikkotiedosto vastaa MIDL:n muodostamaa tiedostoa, jossa se määrittelee C++-funktiot, joita käytetään COM-objektin julkistaman rajapinnan metodien kutsumiseen. Se määrittelee myös lisäfunktioita, jotka käärivät sisäänsä suorat rajapintojen metodit tarjoten käyttöön ominaisuudet ja metodit samaan tapaan kuin Visual Basic -asiakkaat. Ominaisuuksia voidaan käsitellä luokan jäsenmuuttujien kautta, ja metodit on kiedottu niin, että paluuarvoina annetaan *[out, retval]*-parametrit. HRESULT-arvo käännetään — jos se ilmoittaa virheestä, nostetaan **_com_error**-poikkeus. **_bstr_t** ja **_variant_t** luokkia käytetään näiden kiedottujen funktioiden argumenttien ja paluuarvojen tyyppeinä aina kun se muu-ten on mahdollista.

Pääotsikkotiedoston nimi on sama kuin vastaavan tyyppikirjaston tai dynaamisesti linkitetyn tiedoston nimi, lukuun ottamatta tarkenninta .tlh. Toisella otsikkotiedostolla on sama nimi kuin tyyppikirjastolla lukuun ottamatta .tli-tarkenninta. Toinen otsikkotiedosto sisältää kääntäjän muodostamat käärefunktiot ja se sisällytetään (#include-komnnon avulla) pääotsikkotiedostoon.

Molemmat otsikkotiedostot sijoitetaan output-hakemistoon. Kääntäjä lukee ja kääntää ne, mikäli pääotsikkotiedosto on osoitettu #include-direktiivillä.

Pääotsikkotiedosto sisältää COM-objektien julkistamien rajapintojen älykkäiden osoittimien määritelmät. **_COM_SMARTPTR_TYPEDEF**-makroa käytetään kaikkien tyyppikirjastossa määriteltyjen **_com_ptr_t**-mallien muodostamiseen. comdef.h-tiedosto, joka sisältää kääntäjän tukiluokkien määrittelyt, sisällytetään .tlh-tiedostoon.

Seuraavassa harjoituksessa näet, kuinka #import-direktiiviä käytetään tyyppikirjaston tuomiseen ja kuinka asiakasohjelmassa käytetään muodostettuja osoittimia ja käärefunktioita. Muokkaamme aiemmin tässä luvussa tehtyä EncodeHello-sovellusta.

► **EncodeServer-tyyppikirjaston tuominen**

1. Palaa EncodeHello-projektiin. Etsi FileViewissä StdAfx.h-tiedosto.
2. Lisää //{{AFX_INSERT_LOCATION}}-kommentin eteen seuraavan kaltainen rivi:

   ```
   #import "C:\EncodeServer\Debug\EncodeServer.dll" no_namespace
   ```

   Varmista, että kirjoitat EncodeServer.dll-tiedostolle oikean polun. Se on todennäköisesti koneellasi erilainen kuin mallissa. **no_namespace**-attribuutti täytyy määritellä. Näin varmistetaan, että tyyppikirjastosta generoitu luokka määritellään globaaliin nimialueeseen.
3. Tallenna ja sulje StdAfx.h-tiedosto. Napauta FileViewissä hiiren kakkospainikkeella **StdAfx.cpp**-tiedostoa ja valitse pikavalikosta **Compile StdAfx.cpp**.
4. Kun käännös on valmis, etsi tiedostot EncodeServer.tlh ja EncodeServer.tli EncodeServer\Debug -kansiosta. Tarkista .tlh-tiedoston sisältö. Huomaa, että **IEncoderPtr**-osoittimen määrittely on tehty seuraavalla rivillä:

   ```
   _COM_SMARTPTR_TYPEDEF(IEncoder,  __uuidof(IEncoder));
   ```

   Huomaa myös seuraavat jäsenfunktioiden määrittelyt **IEncoder**-struktuurin määrittelyssä:

   ```
   // Property data
   __declspec(property(get=GetKey,put=PutKey))
   short Key;
   ```

```
// Wrapper methods for error-handling
_bstr_t EncodeString (_bstr_t instring);
short GetKey ();
void PutKey (short pVal);

// Raw methods provided by interface
virtual HRESULT __stdcall raw_EncodeString (BSTR instring, BSTR *
      outstring) = 0;
virtual HRESULT __stdcall get_Key (short * pVal) = 0;
virtual HRESULT __stdcall put_Key (short pVal) = 0;
```

Lopussa olevat “raaka”-metodit ovat vastaavia kuin MIDL-kääntäjän EncodeServer.h-tiedostoon muodostamat. Kääremetodit näyttävät kuitenkin paljon enemmän tavallisen C++-luokan sisältämiltä metodeilta. Huomaa, että Key-jäsenmuuttujan määrittelyssä käytetään *__declspec(property)*-määrittelijää. Sen avulla luokan käyttäjät pystyvät käsittelemään **Get** ja **Put** -funktioita käyttämällä Key-muuttujaa sijoituslauseen vasemmalla ja oikealla puolella. Huomaa myös, että nämä funktiot käyttävät **_bstr_t**-luokkaa argumentteina ja paluuarvoina.

Käärefunktioiden toteutukset ovat tiedostossa EncodeServer.tli. **GetKey()**-funktion toteutus, joka seuraa tuonnempana, havainnollistaa, kuinka *[out, retval]* parametrit välitetään takaisin paluuarvoina ja kuinka HRESULT-arvo tulkataan ja mahdollisesti aiheutetaan poikkeus.

```
inline short IEncoder::GetKey ()
{
    short _result;
    HRESULT _hr = get_Key(&_result);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
    return _result;
}
```

Nyt kun tyyppikirjasto on tuotu, voidaan sovelluksen koodia muuttaa niin, että siinä hyödynnetään tyyppikirjaston tuomisen seurauksena syntyneitä luokkia.

► EncodeHello-sovelluksen koodin muokkaaminen

1. Poista seuraavat **#include**-direktiivit EncodeHello.cpp-tiedoston alusta:

```
#include “EncodeServer.h”
#include “EncodeServer_i.c”
```

2. Muutetaan EncodeHello-sovelluksen **Main()**-funktio seuraavanlaiseksi: (Tämä koodi on asennettu oheisrompulta tiedostoon CH10_02.cpp.)

```
int main(int argc, char* argv[])
{
    CoInitialize(NULL);
    {
        IEncoderPtr pServer;

        HRESULT hr = pServer.CreateInstance(__uuidof(Encoder));

        if(SUCCEEDED(hr))
        {
            short nKey = 1;
            cout << "Enter a code key between -5 and 5: ";
            cin >> nKey;

            _bstr_t bstrHello = "Hello World!";
            _bstr_t bstrCodedHello;

            try
            {
                pServer->Key = nKey;
                bstrCodedHello = pServer->EncodeString(bstrHello);

                cout << "\n" << (const char *)
                bstrCodedHello << "\n\n";
            }
            catch(_com_error e)
            {
                cout << e.ErrorMessage() << "\n\n";
            }
        }
    }
    ::CoUninitialize();

    return 0;
}
```

Huomaa tässä esimerkissä, että sovelluksen koodi sijoitetaan omaan koodilohkoonsa **CoInitialize()** ja **CoUninitialize()** kutsujen väliin. Näin varmistetaan, että *pServer*-muuttuja poistuu käytöstä ennen **CoUninitialize()**-funktion kutsua, joka sulkee COM-kirjastot. Kun *pServer* on tarpeeton, **_com_ptr_t**-tuhoaja kutsuu kapseloidun **IEncoder**-osoittimen **Release()**-metodia. Jos tämä tapahtuu sen *jälkeen*, kun COM-kirjastot on suljettu, katastrofi on valmis.

Huomaat, että tämä koodi on paljon helpompilukuinen kuin aiempi versio, ja että se muistuttaa läheisesti tavallista C++-koodia, jossa ei käytetä COM:a. Älä anna ulkonäön pettää — on joka tapauksessa mahdotonta kirjoittaa tehokasta ja virheetöntä COM-koodia tuntematta allaolevaa tekniikkaa.

Oppitunnin yhteenveto

MIDL-kääntäjä muodostaa otsikkotiedostoja, joiden avulla COM-palvelimen rajapinnat ja GUID-määrittelyt saadaan käyttöön C ja C++ -asiakkaiden lähdekoodissa. Näissä tiedostoissa määritellään rakenteet, joita voidaan käyttää COM-objektin muodostamien rajapintojen käsittelmiseen. GUID määrittelyjen luettavat versiot määritellään myös.

Visual C++ COM -kääntäjä sisältää **#import**-lauseen, jonka avulla voit luoda COM-objektin ja sen rajapintojen otsikkotiedostot tyyppikirjastossa olevien tietojen perusteella. Muodostetuissa tiedostoissa käytetään paljon **_com_ptr_t**-osoitinluokkaa, **_com_error-**poikkeusluokkaa ja muita Microsoftin omia C++-kielen laajennuksia, joiden avulla COM-rajapinta kääritään ja helpotetaan asiakkaan ohjelmointia. **_bstr_t** ja **_variant_t** -luokat helpottavat **BSTR** ja **VARIANT**-tietotyyppien käyttämistä.

Oppitunti 2: COM-objektien uudelleenkäyttö

Yksi olio-ohjelmoinnin tärkeimmistä eduista on mahdollisuus koodin uudelleenkäyttöön. Kun olet luonut objektin, joka suorittaa tarkasti määritellyn palvelun tehokkaasti ja virheettömästi, haluat todennäköisesti käyttää kyseistä objektia useita kertoja. Tällä oppitunnilla opit eri tapoja, joilla COM-objekti voi käyttää muiden COM-objektien palveluja. Erityisesti käsitellään kahta tavallisinta uudelleenkäyttötekniikkaa, sisällytystä ja koostamista. Nämä tekniikat mahdollistavat olemassaolevan COM-objektin upottamisen kehitettävään COM-objektiin, paljastamalla upotetun objektin rajapinnat sen sisältävän objektin rajapintoina.

Tämän oppitunnin jälkeen:

- Tiedät eron toteutuksen ja rajapintojen periyttämisen välillä.
- Tiedät sisällyttämisen ja koostamisen välisen eron objekteja uudellen käytettäessä.
- Tiedät, kuinka koostettavan objektin **IUnknown**-rajapinta toteutetaan.
- Tiedät, kuinka koostaminen toteutetaan ATL:ssä.

Oppitunnin arvioitu kesto: 30 minuuttia

Objektien uudelleenkäyttö C++:ssa ja COM:ssa

C++-ohjelmoijat käyttävät yleensä valmiita luokkia käyttämällä joko *sisällytys* (containment) tai *periyttämis* (inheritance) -tekniikkaa. Sisällyttäminen tarkoittaa objektin määrittämistä luokan sisällä, kuten seuraavassa esimerkissä nähdään:

```
#include “Acme.h”   // Contains AcmeViewer class definition
            // Defines the member functions SetFile() and Display()
class MyViewer
{
protected:
      AcmeViewer m_obj;

public:
      MyViewer() {m_obj.SetFile(“C:\images\scully.gif”, AV_TYPE_GIF);}
      DisplayGif() {m_obj.Display();}
}
```

Tässä koodissa määritellään yksinkertainen luokka **MyViewer**, joka sisältää valmiiksi olemassaolevan **AcmeViewer-**luokan ilmentymän. **MyViewer** uudelleenkäyttää **AcmeViewer**-luokan koodin. **MyViewer**-muodostin luo sisäl-

lytetyn **AcmeViewer**-objektin, ja **MyViewer::DisplayGif()**-funktio käyttää **AcmeViewer**-objektin tarjoamia palveluja. **MyViewer**-luokka voi hallita **AcmeViewer**-objektin käyttöä ja tapaa, jolla objektin palveluja käytetään.

Periyttäminen on tehokas uudelleenkäyttömenetelmä, joka on keskeinen osa C++-kieltä. **AcmeViewer**-luokasta aikaisemmin saamiesi tietojen perusteella pitäisi olla itsestään selvää, että seuraava määrittely:

```
#include “Acme.h”

class MyViewer : public AcmeViewer
{
public:
    MyViewer() {SetFile(“C:\images\scully.gif”, AV_TYPE_GIF);}
}
```

antaa mahdollisuuden kutsua **AcmeViewer**-luokan suojattuja ja suojaamattomia jäsenfunktioita **MyViewer**-objektin jäsenfunktioina, seuraavasti:

```
MyViewer aViewer;
aViewer.Display();
```

COM tukee sisällyttämistä, mutta ei tue periyttämistä samalla tavalla kuin C++. Edellisessä esimerkissä **MyViewer**-luokka perii **AcmeViewer**-luokan tarjoamat toiminnot. **AcmeViewer**-luokan suojatut ja suojaamattomat jäsenfunktiot ovat **MyViewer**-luokan käytettävissä. Tällaisessa tapauksessa puhutaan *toteutuksen* periyttämisestä.

COM ei tue toteutuksen periyttämistä. COM erottaa COM-objektin tarjoaman rajapinnan ja sen varsinaisen toteutuksen loogisesti toisistaan. Julkistettu COM-rajapinta, joka identifioidaan maailmalla GUID:nsa perusteella on *muuttumaton* —sen on taattu pysyvän muuttumattomana. Vaikka olisi olemassa monia COM-objekteja, jotka toteuttavat rajapinnan määrittämän toiminnallisuuden useilla eri tavoilla, asiakas tietää aina, kuinka näiden objektien kanssa kommunikoidaan. Ra-japinta edustaa COM-palvelimen ja asiakkaan välistä sopimusta, joka määrittelee, millaista tietoa objekti odottaa saavansa ja millaista tietoa se palauttaa.

Toteutuksen periyttäminen tekee periytetystä luokasta riippuvaisen kantaluokan toteutuksesta. Jos kantaluokan toteutus muuttuu, periytyminen voi lakata toimimasta oikein ja tarvitsisi näin ollen toteutuksensa muuttamista. Tämä aiheuttaa ongelmia suurissa ohjelmointitöissä, erityisesti, jos kantaluokan lähdekoodi ei ole käytettävissä. COM-objektien rajapinnan ja toteutuksen erottaminen toisistaan pienentää näitä ongelmia, mutta tarkoittaa samalla, että et voi uudelleenkäyttää COM-komponentteja periyttämällä niitä toisistaan kuten C++-luokkia.

COM tukee periyttämisen muotoa, jota kutsutaan *rajapinnan* periyttämiseksi. Rajapinnat tehdään C++:ssa vain virtuaalifunktioita sisältävinä abstrakteina luokkina, jotka määrittelevät, mutta eivät toteuta, rajapinnan metodit. Periyttämällä rajapinnan toisesta määrittelet vtablen rakenteen. Vtable sisältää toteutettujen metodien osoittimet. Seuraava määrittely esimerkiksi luo oikein rakennetun vtablen, joka sisältää osoittimet kolmelle **IUnknown**-metodille, joita seuraavat **IEncoder**-metodien osoittimet:

```
IEncoder : public IUnknown
{
    // IEncoder methods declared here
}
```

Rajapintojen periyttämisen tekee mahdolliseksi COM-rajapintojen muuttumattomuus. Voit periyttää COM-rajapinnan mistä tahansa toisesta COM-rajapinnasta ja voit olla varma, ettei kukaan muuta kantarajapintaa tietämättäsi ja näin sekoita vtableasi.

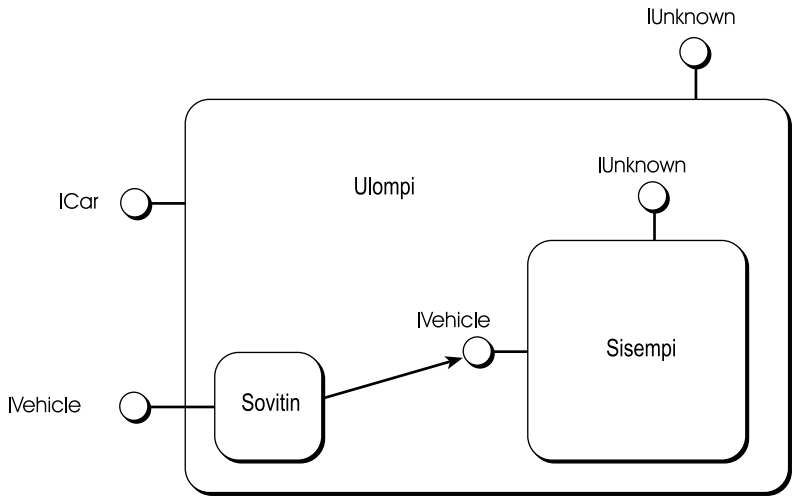
Sisällyttäminen ja koostaminen

COM-objektien tekijät voivat uudelleenkäyttää olemassa olevia COM-objekteja, joko *sisällyttämällä* (containment) *tai koostamalla* (aggregation). Termit sisällyttäminen ja koostaminen viittaavat tapaan, jolla toinen objekti (*ulompi objekti*, outer object) uudelleen käyttää toista objektia (*sisempi objekti*, inner object).

Sisällyttäminen

COM:n sisällyttäminen toimii samaan tapaan kuin C++:n sisällyttämistekniikka, josta kerrottiin aiemmin tällä oppitunnilla. Ulompi objekti luo sisemmän objektin (tavallisesti kutusmalla **CoCreateInstance()**-funktiota) ja varastoi viittauksen sisempään objektin tietojäseneensä. Ulompi objekti toteuttaa sisemmän objektin rajapinnat sovitinrajapintojen avulla. Sovitinrajapinta ohjaa metodien kutsut sisemmälle objektille.

Kuvassa 10.1 nähdään, kuinka **ICar**-rajapinnan sisältävä COM-objekti sisältää COM-objektin, jolla on **IVehicle**-rajapinta ja kuinka ulompi objekti julkistaa molemmat rajapinnat.



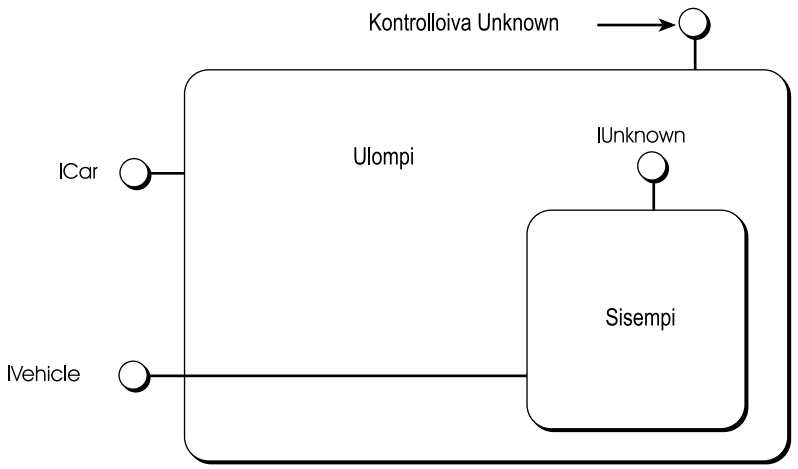
Kuva 10.1 COM:n sisällyttäminen

Ulomman objektin ei tarvitse julkistaa kaikkia sisemmän objektin rajapintoja. Kuten C++ sisällyttämisessäkin, ulomman objektin rajapinta kontrolloi sisemmän objektin rajapintojen käsittelyä, ja voi myös kontrolloida tapaa, jolla sisemmän kontrollin rajapintoja käytetään. Ulompi objekti ei ohjaa mitään sisemmän objektin **IUnknown**-rajapinnalle meneviä kutsuja eteenpäin, koska sisemmällä objektilla ei ole mitään tietoa ulomman objektin julkistamista rajapinnoista. Jos esimerkiksi kuvassa 10.1 kuvatun objektin asiakas pyytää **IUnknown**-osoittimen, se olettaa voivansa käyttää tämän osoittimen avulla **ICar**-rajapintaa siinä kuin **IVehicle**-rajapintaakin. Sisemmän objektin **IUnknown**-rajapinnan osoitin ei pysty käsittelemään **ICar**-osoittimelle meneviä palvelupyyntöjä.

Koostaminen

Kuten sisällytettäessäkin, ulompi objekti varastoi sisemmän objektin **IUnknown**-rajapintaan kohdistuvat viittaukset. Kuitenkin sisällyttämisestä poiketen ulompi objekti julkistaa sisemmän objektin suoraan asiakkaan käsiteltäväksi sen sijaan, että loisi sitä vastaavan rajapinnan edustajan. Näin ollen koostettaessa vältytään metodikutsujen edelleenohjaamisen sisällyttämisessä aiheuttamalta lisäkuormalta.

Seuraavan sivun kuvassa 10.2 nähdään, kuinka sisemmän objektin rajapinta julkistetaan koostettaessa.



Kuva 10.2 Koostaminen COM:ssa

Jotta koostaminen onnistuisi, sisemmän objektin täytyy olla *koostettava* (aggregatable) — se on pitänyt ohjelmoida niin, että se tukee koostamista. Koska ulomman objektin asiakkaat voivat nyt saada sisemmän objektin julkistamisen rajapintojen osoittimet, ne voivat kutsua sisemmän objektin **IUnknown**-rajapinnan metodeja. Koska sisempi objekti ei voi käsitellä **QueryInterface()**, **AddRef()** ja **Release()** -palvelupyyntöjä ulomman objektin puolesta, asiakkaan sisemmän objektin **IUnknown**-rajapinnalle tekemät kutsut täytyy delegoida ulomman objektin **IUnknown**-rajapinnalle (*controlling unknown*).

Huomio Pidä mielessä, että koostetun objektin asiakas ei ole millään lailla tietoinen koostamisesta, joka on asiakkaan näkökulmasta toteutukseen liittyvä yksityiskohta. Asiakas näkee yhden COM-objektin, jolla on yksi **IUnknown**-osoitin.
Jos asiakas saa sisemmän objektin rajapinnan osoittimen, se olettaa sen olevan ulomman objektin julkistaman rajapinnan osoitin.

Kun ulompi objekti kutsuu sisempää objektia, se käyttää **CoCreateInstance()**-funktion toista argumenttia kontrolloivan unknown-osoittimen välittämiseen objektin luokkatehtaalle. Jos tämä osoite ei ole NULL, sisempi objekti tietää, että se on koostettu ja delegoi ulkoisilta asiakkailta tulevat **IUnknown**-metodien kutsut kontrolloivalle rajapinnalle.

Sisemmän objektin täytyy kuitenkin kyetä käsittelemään ulkoisten asiakkaiden **IUnknown**-metodien kutsut (jotka ohjataan eteenpäin) erilailla kuin ne **IUnknown**-metodien kutsut, jotka ovat peräisin kontrolloivalta unknowlta (ja joita ulompi objekti käyttää saadakseen yhteyden rajapintoihin ja kontrolloidakseen koostettujen objektien elinaikaa). Tästä syystä koostettavan COM-objektin täytyy sisältää kaksi versiota **IUnknown**-rajapinnasta: *delegoiva* ja *ei-delegoiva*

versio. Ulompi komponentti kutsuu ei-delegoivia **IUnknown**-metodeja; ulkoiset asiakkaat kutsuvat delegoivia **IUnknown**-metodeja. Delegoivat metodit uudelleenohjaavat kutsut kontrolloivalle unknown:lle (jos objekti on koostettu), tai ei-delegoivalle unknown:lle (jos objektia ei ole koostettu).

Koostaminen ATL:ssä

ATL sisältää joukon makroja ja kantametodeja, jotka helpottavat komponenttien koostamista. Koostettava (sisempi) objekti luodaan käyttämällä ATL Object Wizardia uuden COM-objektin lisäämiseen ATL-projektiin ja määrittämällä, että komponentti tukee koostamista. Sopivat valinnat tehdään **Attributes**-sivulta, joka nähdään kuvassa 10.3.

Kuva 10.3 ATL Object Wizardin koostamisasetukset

Oletuksena ATL COM-objektit ovat koostettavia. Ei-koostettava objekti luodaan valitsemalla vaihtoehto **No** aggregation-ryhmästä. **Only**-valinnan seurauksena luodaan objekti, jota voidaan käyttää vain koostettuna.

► **Ulomman objektin toteuttaminen**

1. Lisää **IUnknown**-osoitin luokan objektiin ja alusta se muodostimessa arvolla NULL. Tässä harjoituksessa viittaamme tähän osoittimeen nimellä **m_pUnkInner**.
2. Määrittele luokan määrittelyn sisällä makro **DECLARE_GET_CONTROLLING_UNKNOWN**. Tässä makrossa määritellään funktio **GetControllingUnknown()**.
3. Ylikuormita metodit **FinalConstruct()** ja **FinalRelease()**. Ne ovat **CComObjectRootBase**-kantaluokan metodeja, joita kutsutaan viimeisimmässä vaiheessa COM-objekteja luotaessa ja tuhottaessa.
4. **FinalConstruct()**-metodissa kutsutaan **CoCreateInstance()**-funktiota ja välitetään sille luotavan sisemmän objektin CLSID ensimmäisenä argument-

tina. Välitä **GetControllingUnknown()**-funktion palauttama arvo toisena argumenttina, jolloin sisempi objekti saa kontrolloivan **IUnknown**-osoittimen. Välitä **m_pUnkInner**-osoittimen osoitin viidentenä argumettina, sisemmän objektin **IUnknown**-rajapinnan osoitinta varten. Seuraavassa näet esimerkin ylikuormitetusta **FinalConstruct()**-funktiosta:

```
HRESULT FinalConstruct()
{
    return CoCreateInstance(CLSID_InnerObject,
        GetControllingUnknown(), CLSCTX_ALL,
        IID_IInnerObject, &m_pUnkInner);
}
```

5. Kutsu **FinalRelease**-metodissa sisemmän objketin **IUnknown::Release()**-metodia.
6. Lisää **COM_INTERFACE_ENTRY_AGGREGATE**-makroa käyttämällä mer-kintä sisemmän objektin rajapinnoista ulomman objektin COM-karttaan. Makro on **COM_INTERFACE_ENTRY**-makron erikoistunut versio, joka käsittelee koostettuja objekteja.
   COM_INTERFACE_ENTRY_AGGREGATE ottaa ylimääräisen **IUnknown *** -argumentin, joka osoittaa sisemmän objektin unknown-rajapintaan.
7. Lisää sisemmän objektin rajapinnan määritykset ulomman objektin .idl-tiedostoon ja viittaus tähän rajapintaan [coclass]-osaan.

Oppitunnin yhteenveto

C++-ohjelmoijat ovat tottuneet käyttämään periyttämistä koodia uudelleen-käytettäessä. COM tekee selkeän eron rajapinnan ja sen toteutuksen välille ja vaatii, että rajapinta pysyy maailmanlaajuisesti muuttumattomana. Tästä syystä COM voi tukea rajapintojen periyttämistä (rajapintojen määrittelyjen uudelleen-käyttämistä), mutta ei toteutuksen periyttämistä.

COM tukee kahta koodin uudelleenkäyttötapaa: koostamista ja sisällyttämistä. Nämä tekniikat riippuvat suhteesta, jolla ulompi objekti uudelleenkäyttää sisempää objektia. Kun sisempi objekti on sisällytetty, sen rajapintoja ei julkisteta suoraan asiakkaille, vaan asiakkaat käyttävät niitä ulomman objektin sovittimien kautta. Koostettu objekti julkistaa rajapintansa suoraan asiakkaalle. Kun objekteja koostetaan, täytyy sisempi ja ulompi objekti toteuttaa niin, että asiakas voi käsitellä vain kontrolloivaa rajapintaa — ulomman objektin **IUnknown**-rajapinnan osoitinta. ATL sisältää joukon makroja ja kantaluokkia, jotka helpottavat komponenttien koostamista.

Laboratorio 10: UploadStockData-komponentin toteutus

Tässä laboratoriossa tehdään laboratoriossa 9 luodun UploadStockData-komponentin toteutus. Teemme IUploadStockData-rajapinnan **ConnectToDatabase()**, **Disconnect()** ja **UploadRecord()** -metodien toteutukset. Nämä metodit käyttävät ADO-kirjastoa Stocks-tietokantaan yhdistymiseen. Kertaa luvun 7 oppitunnin 3 tiedot ADO:sta.

Tässä laboratoriossa kirjoitetaan STUpload-sovelluksen **Data**-valikon **Upload**-komennon toteuttamiseen tarvittava koodi. Käyttäjä voi tämän komennon avulla kopioida avattuna olevan dokumentin tiedot Stocks-tietokantaan.

Laboratorion suorittaminen edellyttää, että olet asentanut SQL Serverin ja olet pystyttänyt Stocks-tietokannan johdanto-osassa kuvatulla tavalla.

Yhteystietotiedoston luominen

UploadStockData-komponentti luo ADO **Connection**-objektin Stocks-tietokantayhteyttä varten. **Connection**-objektille täytyy välittää tiedot palvelun toimittajasta, tietokannasta ja turvallisuusasetuksista yhteysmerkkijonossa. Esimerkiksi oman STUpload-sovelluksesi ADO data -kontrollin käyttämä versio yhteysmerkkijonosta on seuraava:

```
Provider=SQLOLEDB.1;Integrated Security=SSPI;Persist Security Info=False;Initial
Catalog=Stocks;Data Source=(local)
```

Kun sovellusta jaetaan myös muille käyttäjille, ei ole viisasta kovakoodata yhteystietoja sovelluksen lähdekoodiin, koska silloin muutokset tietokanta ja verkkoasetuksiin vaatisivat toimiakseen sovelluksen uudelleenkääntämistä ja -jakelua. Tämän tapaisten ongelmien välttämiseksi, yhteystiedot määritellään erityisessä *yhteystietotiedostossa*. Yhteystietotiedoston tarkennin on .udl. Kun OLE DB on asennettu koneellesi, voit konfiguroida yhteystietotiedoston käyttämällä yksinkertaista käyttöliittymää. Kun yhteystietotiedosto on konfiguroitu, asetetaan yhteysmerkkijono viittaamaan tiedostoon seuraavasti:

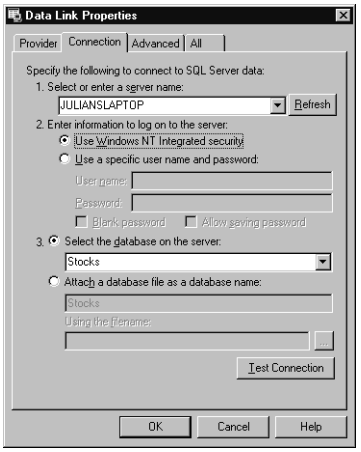
```
File Name=C:\DataLinks\STLink.udl
```

Yhteystietotiedoston konfiguroiminen tai uudelleenjakaminen onnistuu helposti asiakkaan koneella, jos asetukset muuttuvat.

► **Yhteystietotiedoston luominen**

1. Luo Resurssienhallinnassa uusi kansio kiintolevyn juurihakemistoon. Anna kansion nimeksi **DataLinks**.
2. Napauta **DataLinks**-kansiota resurssienhallinnan vasemmassa osassa. Napauta hiiren kakkospainikkeella oikeassa osassa ja valitse **New**-valikosta **Microsoft Data Link**.

3. Anna tiedostolle uusi nimi **STLink.udl**.
4. Aloita tietojen muokkaaminen kaksoisnapauttamalla **STLink.udl**-tiedostoa.
5. Valitse **Data Link Properties** -dialogin **Provider**-sivulta **Microsoft OLE DB Provider for SQL Server**.
6. Valitse **Connection**-sivulta (kuvassa 10.4) palvelimeksi paikallinen tietokone. Valitse **Use Windows NT Integrated security**, tai kirjoita sopiva SQL Server -tilin nimi ja salasana.



Kuva 10.4 Yhteystietotiedoston muokkaaminen

7. Valitse palvelimelta **Stocks**-tietokanta. Kokeile yhteyttä napauttamalla **Test Connection**.
8. Sulje ikkuna napauttamalla **OK** ja tallenna asetukset tiedostoon napauttamalla uudelleen **OK**.

IUploadStockData-rajapinnan metodien toteuttaminen

Laboratorion tässä osassa palataan STLoadData-projektiin ja toteutetaan UploadStockData-komponentin julkistaman **IUploadStockData**-rajapinnan **ConnectToDatabase()**, **Disconnect()** ja **UploadRecord()** metodit. Näiden harjoitusten aikana opit, kuinka asiakas voi käyttää ADO:n tarjoamia COM-rajapintoja työskennellessään OLE DB -tietolähteen kanssa.

Ensin täytyy tuoda ADO tyyppikirjasto, jotta GUID määrittelyt saadaan projektissa käyttöön ja jotta saadaan muodostettua älykkäät osoittimet ADO rajapintoihin.

► **ADO-tyyppikirjaston tuominen**

1. Avaa **STUpload.dsw**-työtilatiedosto Visual Studioon. Varmista, että koko **Build**-työkalurivi on näkyvissä ja valitse sitten **STLoadData**-projekti alasvetovalikosta.
2. Avaa FileViewissä Header Files -kansio, joka on **STLoadData Files**:in alla. Kaksoisnapauta **StdAfx.h**-tiedostoa.
3. Lisää seuraava rivi tiedoston loppuun ennen kommenttia // {{AFX_INSERT_LOCATION}} :

```
#import "C:\Program Files\Common Files\System\ado\msado15.dll" \
    no_namespace rename("EOF", "adoEOF")
```

   Tiedoston msado15.dll polku saattaa olla koneellasi erilainen. Varmista, että koodiin tulee oikea polku. Huomaa myös, että esimerkissä on merkkiä \ käytetty jakamaan lause kahdelle riville.

4. Tallenna ja sulje StdAfx.h-tiedosto.
5. Napauta FileViewissä hiiren kakkospainikkeella **StdAfx.cpp**-tiedostoa Source Files -kansiossa. Valitse pikavalikosta **Compile StdAfx.cpp**.

Kääntäjä käsittelee StdAfx.h-tiedostossa olevan #import-komennon ja luo tiedostot msado15.tlh ja msado15.tli Debug-kansioon. Nämä tiedostot sisältävät ADO-rajapintojen GUID-määrittelyt ja älykkäät osoittimet. Rajapinnat ovat nyt käytettävissä STLoadData-projektissa.

UploadStockData-komponentti ottaa yhteyden tietokantaan yhden ADO **Connection**-objektin kautta. Objektia käytetään kutsumalla **UploadRecord()**-metodia. Tämä **Connection**-objekti toteutetaan komponenttiluokan jäsenmuuttujana ja se avataan ja suljetaan käyttämällä **ConnectToDatabase()** ja **Disconnect()** -metodeja.

► **Connection-objektin toteutus**

1. Lisää seuraava **protected**-tietojäsen **CUploadStockData**-luokan määrittelyyn UploadStockData.h-tiedostoon:

```
_ConnectionPtr m_pConnection;
```

2. Lisää samassa tiedostossa olevaan **CUploadStockData**-luokan muodostimeen seuraava rivi:

```
m_pConnection = NULL;
```

3. Etsi UploadStockData.cpp-tiedostosta **CUploadStockData::ConnectToDatabase()**-funktio. Lisää funktioon koodia niin, että se näyttää seuraavalta:

```
STDMETHODIMP CUploadStockData::ConnectToDatabase()
{
    // Test to see if we're connected already
    if(m_pConnection) return S_OK;

    HRESULT hr = m_pConnection.CreateInstance(__uuidof(Connection));
    if(FAILED(hr)) return hr;

    hr = m_pConnection->Open(L"File Name=C:\\STLink.UDL",
        L"", L"", -1);

    if(FAILED(hr)) return hr;

    return S_OK;
}
```

   Muista, että m_pConnection-muuttuja on **_com_ptr_t**-osoitin, joka osoittaa ADO **Connection**-objektin julkistamaan rajapintaan.

4. Etsi **CUploadStockData::Disconnect()**-funktio. Lisää funktioon koodia niin, että se näyttää seuraavalta:

```
STDMETHODIMP CUploadStockData::Disconnect()
{
    if(m_pConnection)
    {
        m_pConnection->Close();
        m_pConnection = NULL;
    }
    return S_OK;
}
```

Voit nyt lisätä **IUploadStockData::UploadRecord()**-metodin toteuttamiseen tarvittavan koodin.

► **UploadRecord()-metodin toteutus**

1. Valitse **Project**-valikosta **Settings**. Varmista, että STLoadData-projekti on valittuna vasemmalta. Valitse **Project Settings** -dialogin **C/C++**-sivun **Category**-valikosta **C++ Language**. Napauta **Enable exception handling** -valintaa ja napauta **OK**.

2. Etsi UploadStockData.cpp-tiedostosta **CUploadStockData::UploadRecord()**-funktio. Lisää funktioon koodia niin, että se näyttää seuraavalta:

(Tämä koodi on asennettu oheisrompulta tiedostoon CH10_03.cpp.)

```
STDMETHODIMP CUploadStockData::UploadRecord(BSTR fund, DATE date,
    double price, BSTR uplBy, DATE uplDate)
{
    // Test for live connection to data source
    if(m_pConnection == NULL)
        return E_FAIL;

    // Create recordset
    _RecordsetPtr pRecordset;

    HRESULT hr = pRecordset.CreateInstance(__uuidof(Recordset));
    if(FAILED(hr)) return hr;

    try
    {
        // Open recordset
        _variant_t vConnection = m_pConnection.GetInterfacePtr();
        hr = pRecordset->Open(L"pricehistory", vConnection,
            adOpenForwardOnly, adLockOptimistic, adCmdTableDirect);

        if(FAILED(hr)) return hr;

        // Add new record, set fields to new values and update
        hr = pRecordset->AddNew();
        if(FAILED(hr)) throw_com_error(hr);

        pRecordset->Fields->GetItem(L"ph_fund")->Value = fund;
        pRecordset->Fields->GetItem(L"ph_date")->Value = date;
        pRecordset->Fields->GetItem(L"ph_price")->Value = price;
        pRecordset->Fields->GetItem(L"ph_uploadedby")->Value = uplBy;
        pRecordset->Fields->GetItem(L"ph_uploaddate")->
            Value = uplDate;

        hr = pRecordset->Update();
        if(FAILED(hr)) throw_com_error(hr);
    }
        catch(_com_error e)
        {
            // very unsophisticated error handling
            try
        {
            pRecordset->Close();
        }
```

```
            catch(...) // Close() may throw another exception
            {
            }
            return E_FAIL;
        }
        pRecordset->Close();

        return S_OK;
    }
```

Huomaa, että olemassaoleva Connection-objekti **m_pConnection** välitetään argumenttina Recordset **Open()**-metodille. Yksityiskohtaisempaa tietoa **Open()**-metodin muista argumenteista saat hakemalla Visual C++:n ohjeesta merkkijonolla "*ADO*".

Huomaa myös, että ADO **Recordset**-objektin **AddNew()** ja **Update()** metodeja käytetään yhdessä lisättäessä uutta tietoa. Uusi tietue lisätään tietojoukkoon **AddNew()**-metodilla, kentien arvot (käsitellään **Recordset** -objektin *Fields*-kokoelman kautta) muutetaan, ja tietokanta päivitetään kutsumalla **Update()**-metodia.

3. Voit nyt kääntää STLoadData-projektin, jolloin STLoadData.dll COM-palvelin luodaan ja rekisteröidään uudelleen.

STUpload-sovelluksen Upload Data -komennon toteuttaminen

Laboratorion seuraavassa osassa toteutetaan STUpload-sovelluksen **Data**-valikon **Upload**-komento. Sovellukseen lisätään myös käyttöliittymäkomentokäsittelijä, jonka avulla valikon ja työkalurivin komento saadaan käyttöön dokumentin ollessa ladattunan sovellukseen.

► **Upload Data -komennon toteuttaminen**

1. Sulje kaikki Visual Studion editorissa avoinna olevat dokumentit.
2. Valitse **Build** työkalurivin alasvetovalikosta **STUpload**-projekti.
3. Avaa ClassWizard painamalla CTRL+W. Valitse **Message Maps** -välilehden **Class name** -alasvetovalikosta **CSTUploadDoc**-luokka.
4. Luo COMMAND ja UPDATE_COMMAND_UI -käsittelijät **OnDataUpload()** ja **OnUpdateDataUpload()** tunnisteelle **ID_DATA_UPLOAD**. Valitse **OnUpdateDataUpload()**-funktio **Member functions** -luettelosta ja siirry funktion toteutukseen napauttamalla **Edit Code**.

5. Toteuta **CSTUploadDoc::OnUpdateDataUpload()**-funktio seuraavasti: (Tämä koodi on asennettu oheisrompulta tiedostoon CH10_04.cpp.)

```
void CSTUploadDoc::OnUpdateDataUpload(CCmdUI* pCmdUI)
{
    // Enable the UploadData command only if there is
    // data on file and a fund currently selected for viewing

    BOOL bEnable = GetCurrentFund().IsEmpty() ? FALSE : TRUE;

    pCmdUI->Enable(bEnable);
}
```

6. Lisää STUploadDoc.cpp-tiedoston alkuun seuraavat direktiivit:

```
#include <comdef.h>     // for Compiler COM support
#include <lmcons.h>     // for the UNLEN constant
#include ".\STLoadData\STLoadData.h"
#include ".\STLoadData\STLoadData_i.c"
```

7. Etsi **CSTUploadDoc::OnDataUpload()**-funktio ja lisää seuraava funktion toteuttava koodi:

```
void CSTUploadDoc::OnDataUpload()
{
    if(AfxMessageBox("Upload current file to database?",
        MB_OKCANCEL) == IDCANCEL)
        return;

    ::CoInitialize(NULL);

    _COM_SMARTPTR_TYPEDEF(IUploadStockData,
        __uuidof(IUploadStockData));

    IUploadStockDataPtr pServer;

    HRESULT hr = pServer.CreateInstance(CLSID_UploadStockData);

    if(SUCCEEDED(hr))
        hr = pServer->ConnectToDatabase();

    if(SUCCEEDED(hr))
    {
        try
        {
            POSITION pos = m_DocList.GetHeadPosition();

            while(pos)
            {
                CStockData sd = m_DocList.GetNext(pos);
```

```
                    BSTR fund = sd.GetFund().AllocSysString();
                    DATE date = sd.GetDate().m_dt;
                    double price = sd.GetPrice();

                    DWORD dwLen = UNLEN + 1;
                    TCHAR cUser[UNLEN + 1];
                    ::GetUserName(cUser, &dwLen);
                    CString strUser(cUser);

                    BSTR uplBy = (strUser.Left(10)).AllocSysString();
                    COleDateTime dtToday =
                        COleDateTime::GetCurrentTime();
                    DATE uplDate = dtToday.m_dt;

                    HRESULT hr = pServer->UploadRecord(fund,
                        date, price, uplBy, uplDate);

                    ::SysFreeString(fund);
                    ::SysFreeString(uplBy);

                    if(FAILED(hr))
                    {
                        CString strPrompt = "Upload of:\n";
                        strPrompt += sd.GetAsString();
                        strPrompt += "\nfailed";

                        if(AfxMessageBox(strPrompt,
                            MB_OKCANCEL) == IDCANCEL)
                            break;
                    }
                }
                if(!pos) // We got to the end of the loop
                    AfxMessageBox("Upload completed successfully");
            }
            catch(_com_error e)
            {
                ::MessageBox(NULL, e.ErrorMessage(), NULL, MB_OK);
            }
            pServer->Disconnect();
        }
        ::CoUninitialize();
}
```

Käy koodi läpi ja varmista, että ymmärrät, kuinka koodissa luodaan ja käytetään UploadStockData-komponentin ilmentymää dokumentin **m_DocList**-jäsenenen sisältämien tietueiden lisäämiseen. Huomaa, että Windowsin API-funktion **GetUserName()** avulla haetaan kirjautuneena olevan käyttäjän nimi. Tämän nimen 10 ensimmäistä merkkiä sijoitetaan Stocks-tietokannan hintahistoriataulukon **ph_uploadedby**-kenttään.

Voit nyt kääntää STUpload-sovelluksen.

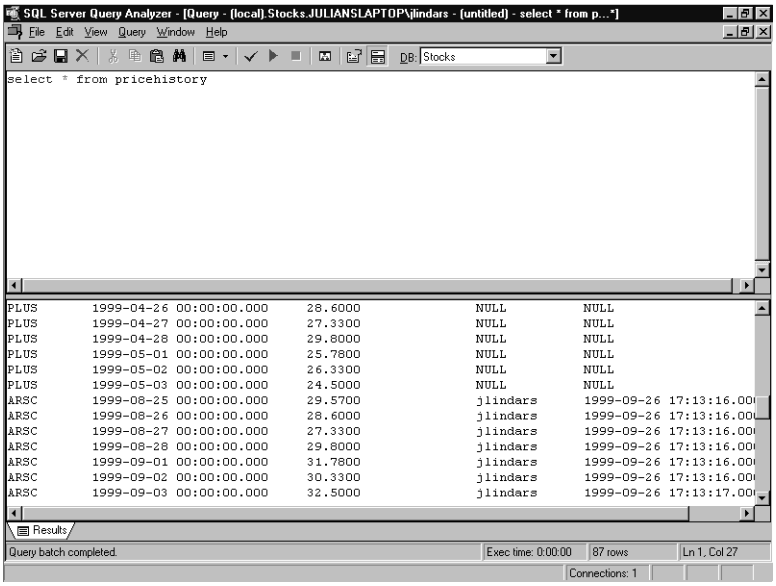
► **Upload Data -komennon kokeileminen**

1. Käynnistä STUpload-sovellus. Valitse **Data**-valikosta **Import**, ja lataa Ch10Test.dat-tiedosto kansiosta Chapter 10\Data.
2. Valitse mikä tahansa osake Select Fund -ikkunasta, jolloin **Upload Data** -komento tulee käyttöön.
3. Valitse **Data**-valikosta **Upload**. Kopioi tiedostossa olevat tiedot Stocks-tietokannan hintahistoriatauluun napautamalla **OK**. Kun kopiointi on valmis, napauta **OK**.
4. Käynnistä uudelleen **Upload**-komento **Data**-valikosta. Huomaat, että tiedoston kaikki tietueet aiheuttavat virheen. Tämä johtuu siitä, että hintahistoriataulun Primary Key -rajoitus estää kaksoisarvojen lisäämisen. Poistu tallennuksesta napauttamalla **Cancel**. Sulje STUpload-sovellus.
5. Napauta **Käynnistä**-valikkoa, valitse **Ohjelmat**, **Microsoft SQL Server 7.0**, ja avaa Query Analyzer -sovellus. Ota yhteys **(local)** tietokantaan.
6. Valitse **Stocks** työkalurivin **DB**-alasvetovalikosta.
7. Kirjoita Query Analyzerin pääikkunaan seuraava rivi:

```
select * from pricehistory
```

8. Suorita SQL-kysely painamalla CTRL+E.

Kyselyn tulokset näkyvät Query Analyzer -ikkunan alaosassa kuten kuvassa 10.5. Varmista, että lisäämäsi rivit näkyvät taulukon lopussa ja että käyttäjänimesi ja päivämäärä näkyvät oikein kentissä ph_uploadedby ja ph_uploaddate.



Kuva 10.5 Hintahistoria avattuna Query Analyzeriin

Kertaus

1. Millainen luokka edustaa COM-rajapintaa C++:ssa?

2. Kuinka **_com_ptr_t-**objekti helpottaa COM-palvelimen elinajan kontrollointia?

3. Mitkä ovat COM:n sisällyttämisen hyvät ja huonot puolet?

4. Mikä on **CoCreateInstance()**-funktion toisen parametrin merkitys?