4

Approaches to reusability

“Follow the lead of hardware design! It is not right that every new development should start from scratch. There should be catalogs of software modules, as there are catalogs of VLSI devices: when we build a new system, we should be ordering components from these catalogs and combining them, rather than reinventing the wheel every time. We would write less software, and perhaps do a better job at that which we do get to write. Wouldn’t then some of the problems that everybody complains about — the high costs, the overruns, the lack of reliability — just go away? Why is it not so?”

You have probably heard remarks of this kind; perhaps you have uttered them yourself. As early as 1968, at the now famous NATO conference on software engineering, Doug McIlroy was advocating “*mass-produced software components*”. Reusability, as a dream, is not new.

It would be absurd to deny that some reuse occurs in software development. In fact one of the most impressive developments in the industry since the first edition of this book was published in 1988 has been the gradual emergence of reusable components, often modest individually but regularly gaining ground; they range from small modules meant to work with Microsoft’s Visual Basic (VBX) and OLE 2 (OCX, now ActiveX) to full libraries, also known as “frameworks”, for object-oriented environments.

Another exciting development is the growth of the Internet: the advent of a wired society has eased or in some cases removed some of the logistic obstacles to reuse which, only a few years ago, might have appeared almost insurmountable.

But this is only a beginning. We are far from McIlroy’s vision of turning software development into a component-based industry. The techniques of object-oriented software construction make it possible for the first time to envision a state of the discipline, in the not too distant future, in which this vision will have become the reality, for the greatest benefit not just of software developers but, more importantly, of those who need their products — quickly, and at a high level of quality.

In this chapter we will explore some of the issues that must be addressed for reusability to succeed on such a large scale. The resulting concepts will guide the discussion of object-oriented techniques throughout the rest of this book.

4.1 THE GOALS OF REUSABILITY

We should first understand why it is so important to improve software reusability. No need here for "motherhood and apple pie" arguments: as we will see, the most commonly touted benefits are not necessarily the most significant; by going beyond the obvious we can make sure that our quest for reuse will pursue the right targets, avoid mirages, and yield the highest return on our investment.

Expected benefits

From more reusable software you may expect improvements on the following fronts:

This section is based on the more extensive discussion of management aspects of reuse in the book "Object Success" [M 1995].

- **Timeliness** (in the sense defined in the discussion of software quality factors: speed of bringing projects to completion and products to market). By relying on existing components we have *less* software to develop and hence can build it faster.
- **Decreased maintenance effort**. If someone else is responsible for the software, that someone is also responsible for its future evolutions. This avoids the *competent developer's paradox*: the more you work, the more work you create for yourself as users of your products start asking you for new functionalities, ports to new platforms etc. (Other than relying on someone else to do the job, or retiring, the only solution to the competent software developer's paradox is to become an *in*competent developer so that no one is interested in your products any more — not a solution promoted by this book.)
- **Reliability**. By relying on components from a reputed source, you have the guarantee, or at least the expectation, that their authors will have applied all the required care, including extensive testing and other validation techniques; not to mention the expectation, in most cases, that many other application developers will have had the opportunity to try these components before you, and to come across any remaining bugs. The assumption here is not necessarily that the component developers are any smarter than you are; simply that the components they build — be they graphics modules, database interfaces, sorting algorithms … — are *their* official assignment, whereas for you they might just be a necessary but secondary chore for the attainment of *your* official goal of building an application system in your own area of development.
- **Efficiency**. The same factors that favor reusability incite the component developers to use the best possible algorithms and data structures known in their field of specialization, whereas in a large application project you can hardly expect to have an expert on board for *every* field touched on by the development. (Most people, when they think of the connection between reusability and efficiency, tend to see the reverse effect: the loss of fine-tuned optimizations that results from using general solutions. But this is a narrow view of efficiency: in a large project, you cannot realistically perform such optimizations on every piece of the development. You can, however, aim at the best possible solutions in your group's areas of excellence, and for the rest rely on someone else's expertise.)

- **Consistency**. There is no good library without a strict emphasis on regular, coherent design. If you start using such a library — in particular some of the best current object-oriented libraries — its style will start to influence, through a natural process of osmosis, the style of the software that you develop. This is a great boost to the quality of the software produced by an application group.
- **Investment**. Making software reusable is a way to preserve the know-how and inventions of the best developers; to turn a fragile resource into a permanent asset.

Many people, when they accept reusability as desirable, think only of the first argument on this list, improving productivity. But it is not necessarily the most important contribution of a reuse-based software process. The reliability benefit, for example, is just as significant. It is extremely difficult to build guaranteeably reusable software if every new development must independently validate every single piece of a possibly huge construction. By relying on components produced, in each area, by the best experts around, we can at last hope to build systems that we trust, because instead of redoing what thousands have done before us — and, most likely, running again into the mistakes that they made — we will concentrate on enforcing the reliability of our truly new contributions.

This argument does not just apply to reliability. The comment on efficiency was based on the same reasoning. In this respect we can see reusability as standing apart from the other quality factors studied in chapter 1: by enhancing it you have the potential of enhancing **almost all** of the other qualities. The reason is economic: if, instead of being developed for just one project, a software element has the potential of serving again and again for many projects, it becomes economically attractive to submit it to the best possible quality-enhancing techniques — such as formal verification, usually too demanding to be cost-effective for most projects but the most mission-critical ones, or extensive optimization, which in ordinary circumstances can often be dismissed as undue perfectionism. For reusable components, the reasoning changes dramatically; improve just one element, and thousands of developments may benefit.

This reasoning is of course not completely new; it is in part the transposition to software of ideas that have fundamentally affected other disciplines when they turned from individual craftsmanship to mass-production industry. A VLSI chip is more expensive to build than a run-of-the-mill special-purpose circuit, but if well done it will show up in countless systems and benefit their quality because of all the design work that went into it once and for all.

Reuse consumers, reuse producers

If you examined carefully the preceding list of arguments for reusability, you may have noted that it involves benefits of two kinds. The first four are benefits you will derive from basing your application developments on existing reusable components; the last one, from making your *own* software reusable. The next-to-last (consistency) is a little of both.

This distinction reflects the two aspects of reusability: the **consumer view**, enjoyed by application developers who can rely on components; and the **producer view**, available to groups that build reusability into their own developments.

In discussing reusability and reusability policies you should always make sure which one of these two views you have in mind. In particular, if your organization is new to reuse, remember that it is essentially impossible to start as a reuse producer. One often meets managers who think they can make development reusable overnight, and decree that no development shall henceforth be specific. (Often the injunction is to start developing "business objects" capturing the company's application expertise, and ignore general-purpose components — algorithms, data structures, graphics, windowing and the like — since they are considered too "low-level" to yield the real benefits of reuse.) This is absurd: developing reusable components is a challenging discipline; the only known way to learn is to start by using, studying and imitating good existing components. Such an approach will yield immediate benefits as your developments will take advantage of these components, and it will start you, should you persist in your decision to become a producer too, on the right learning path.

Reuse Path principle

Be a reuse consumer before you try to be a reuse producer.

Here too "Object Success" explores the policy issues further.

4.2 WHAT SHOULD WE REUSE?

Convincing ourselves that Reusability Is Good was the easy part (although we needed to clarify *what* is really good about it). Now for the real challenge: how in the world are we going to get it?

The first question to ask is what exactly we should expect to reuse among the various levels that have been proposed and applied: reuse of personnel, of specifications, of designs, of "patterns", of source code, of specified components, of abstracted modules.

Reuse of personnel

The most common source of reusability is the developers themselves. This form of reuse is widely practiced in the industry: by transferring software engineers from project to project, companies avoid losing know-how and ensure that previous experience benefits new developments.

This non-technical approach to reusability is obviously limited in scope, if only because of the high turnover in the software profession.

Reuse of designs and specifications

Occasionally you will encounter the argument that we should be reusing designs rather than actual software. The idea is that an organization should accumulate a repository of blueprints describing accepted design structures for the most common applications it develops. For example, a company that produces aircraft guidance systems will have a set of model designs summarizing its experience in this area; such documents describe module templates rather than actual modules.

This approach is essentially a more organized version of the previous one — reuse of know-how and experience. As the discussion of documentation has already suggested, the very notion of a design as an independent software product, having its own life separate from that of the corresponding implementation, seems dubious, since it is hard to guarantee that the design and the implementation will remain compatible throughout the evolution of a software system. So if you only reuse the design you run the risk of reusing incorrect or obsolete elements.

These comments are also applicable to a related form of reuse: reuse of specifications.

To a certain extent, one can view the progress of reusability in recent years, aided by progress in the spread of object technology and aiding it in return, as resulting in part from the downfall of the old idea, long popular in software engineering circles, that the only reuse worthy of interest is reuse of design and specification. A narrow form of that idea was the most effective obstacle to progress, since it meant that all attempts to build actual components could be dismissed as only addressing trivial needs and not touching the truly difficult aspects. It used to be the dominant view; then a combination of theoretical arguments (the arguments of object technology) and practical achievements (the appearance of successful reusable components) essentially managed to defeat it.

"Defeat" is perhaps too strong a term because, as often happens in such disputes, the result takes a little from both sides. The idea of reusing designs becomes much more interesting with an approach (such as the view of object technology developed in this book) which removes much of the gap between design and implementation. Then the difference between a module and a design for a module is one of degree, not of nature: a module design is simply a module of which some parts are not fully implemented; and a fully implemented module can also serve, thanks to abstraction tools, as a module design. With this approach the distinction between reusing modules (as discussed below) and reusing designs tends to fade away.

Design patterns

In the mid-nineteen-nineties the idea of *design patterns* started to attract considerable attention in object-oriented circles. Design patterns are architectural ideas applicable across a broad range of application domains; each pattern makes it possible to build a solution to a certain design issue.

Chapter 21 discusses the undoing pattern.

Here is a typical example, discussed in detail in a later chapter. The *issue*: how to provide an interactive system with a mechanism enabling its users to undo a previously executed command if they decide it was not appropriate, and to reexecute an undone command if they change their mind again. The *pattern*: use a class *COMMAND* with a precise structure (which we will study) and an associated "history list". We will encounter many other design patterns.

[Gamma 1995]; *see also* [Pree 1994].

One of the reasons for the success of the design pattern idea is that it was more than an idea: the book that introduced the concept, and others that have followed, came with a catalog of directly applicable patterns which readers could learn and apply.

Design patterns have already made an important contribution to the development of object technology, and as new ones continue to be published they will help developers to benefit from the experience of their elders and peers. How can the general idea contribute to reuse? Design patterns should not encourage a throwback to the "*all that counts is design reuse*" attitude mentioned earlier. A pattern that is *only* a book pattern, however elegant and general, is a pedagogical tool, not a reuse tool; after all, computing science students have for three decades been learning from their textbooks about relational query optimization, Gouraud shading, AVL trees, Hoare's Quicksort and Dijkstra's shortest path algorithm without anyone claiming that these techniques were breakthroughs in reusability. In a sense, the patterns developed in the past few years are only incremental additions to the software professional's bag of standard tricks. In this view the new contribution is the patterns themselves, not the idea of pattern.

As most people who have looked carefully at the pattern work have recognized, such a view is too limited. There seems to be in the very notion of pattern a truly new contribution, even if it has not been fully understood yet. To go beyond their mere pedagogical value, patterns must go further. A successful pattern cannot just be a book description: it must be a **software component**, or a set of components. This goal may seem remote at first because many of the patterns are so general and abstract as to seem impossible to capture in actual software modules; but here the object-oriented method provides a radical contribution. Unlike earlier approaches, it will enable us to build reusable modules that still have replaceable, not completely frozen elements: modules that serve as general schemes (*patterns* is indeed the appropriate word) and can be adapted to various specific situations. This is the notion of *behavior class* (a more picturesque term is *programs with holes*); it is based on O-O techniques that we will study in later chapters, in particular the notion of deferred class. Combine this with the idea of groups of components intended to work together — often known as *frameworks* or more simply as *libraries* — and you get a remarkable way of reconciling reusability with adaptability. These techniques hold, for the pattern movement, the promise of exerting, beyond the new-bag-of-important-tricks effect, an in-depth influence on reusability practices.

See "Programs with holes", page 506.

Reusability through the source code

Personnel, design and specification forms of reuse, useful as they may be, ignore a key goal of reusability. If we are to come up with the software equivalent of the reusable parts of older engineering disciplines, what we need to reuse is the actual stuff of which our products are made: executable software. None of the targets of reuse seen so far — people, designs, specifications — can qualify as the off-the-shelf components ready to be included in a new software product under development.

If what we need to reuse is software, in what form should we reuse it? The most natural answer is to use the software in its original form: source text. This approach has worked very well in some cases. Much of the Unix culture, for example, originally spread in universities and laboratories thanks to the on-line availability of the source code, enabling users to study, imitate and extend the system. This is also true of the Lisp world.

The economic and psychological impediments to source code dissemination limit the effect that this form of reuse can have in more traditional industrial environments. But a more serious limitation comes from two technical obstacles:

See also "Formats for reusable component distribution", page 79 below.

- Identifying reusable software with reusable source removes information hiding. Yet no large-scale reuse is possible without a systematic effort to protect reusers from having to know the myriad details of reused elements.
- Developers of software distributed in source form may be tempted to violate modularity rules. Some parts may depend on others in a non-obvious way, violating the careful limitations which the discussion of modularity in the previous chapter imposed on inter-module communication. This often makes it difficult to reuse some elements of a complex system without having to reuse everything else.

A satisfactory form of reuse must remove these obstacles by supporting abstraction and providing a finer grain of reuse.

Reuse of abstracted modules

All the preceding approaches, although of limited applicability, highlight important aspects of the reusability problem:

- Personnel reusability is necessary if not sufficient. The best reusable components are useless without well-trained developers, who have acquired sufficient experience to recognize a situation in which existing components may provide help.
- Design reusability emphasizes the need for reusable components to be of sufficiently high conceptual level and generality — not just ready-made solutions to special problems. The classes which we will encounter in object technology may be viewed as design modules as well as implementation modules.
- Source code reusability serves as a reminder that software is in the end defined by program texts. A successful reusability policy must produce reusable program elements.

The discussion of source code reusability also helps narrow down our search for the proper units of reuse. A basic reusable component should be a software element. (From there we can of course go to *collections* of software elements.) That element should be a *module* of reasonable size, satisfying the modularity requirements of the previous chapter; in particular, its relations to other software, if any, should be severely limited to facilitate independent reuse. The information describing the module's capabilities, and serving as primary documentation for reusers or prospective reusers, should be *abstract*: rather than describing all the details of the module (as with source code), it should, in accordance with the principle of Information Hiding, highlight the properties relevant to clients.

The term **abstracted module** will serve as a name for such units of reuse, consisting of directly usable software, available to the outside world through a description which contains only a subset of each unit's properties.

The rest of part B of this book is devoted to devising the precise form of such abstracted modules; part C will then explore their properties.

More on distribution formats below.

The emphasis on abstraction, and the rejection of source code as the vehicle for reuse, do not necessarily prohibit *distributing* modules in source form. The contradiction is only apparent: what is at stake in the present discussion is not how we will deliver modules to their reusers, but what they will use as the primary source of information about them. It may be acceptable for a module to be distributed in source form but reused on the basis of an abstract interface description.

4.3 REPETITION IN SOFTWARE DEVELOPMENT

To progress in our search for the ideal abstracted module, we should take a closer look at the nature of software construction, to understand what in software is most subject to reuse.

Anyone who observes software development cannot but be impressed by its repetitive nature. Over and again, programmers weave a number of basic patterns: sorting, searching, reading, writing, comparing, traversing, allocating, synchronizing… Experienced developers know this feeling of *déjà vu*, so characteristic of their trade.

A good way to assess this situation (assuming you develop software, or direct people who do) is to answer the following question:

> *How many times over the past six months did you, or people working for you, write some program fragment for table searching?*

Table searching is defined here as the problem of finding out whether a certain element *x* appears in a table *t* of similar elements. The problem has many variants, depending on the element types, the data structure representation for *t*, the choice of searching algorithm.

Chances are you or your colleagues will indeed have tackled this problem one or more times. But what is truly remarkable is that — if you are like others in the profession — the program fragment handling the search operation will have been written at the lowest reasonable level of abstraction: by writing code in some programming language, rather than calling existing routines.

See bibliographic references on page 99.

To an observer from outside our field, however, table searching would seem an obvious target for widely available reusable components. It is one of the most researched areas of computing science, the subject of hundreds of articles, and many books starting with volume 3 of Knuth's famous treatise. The undergraduate curriculum of all computing science departments covers the most important algorithms and data structures. Certainly not a mysterious topic. In addition:

- It is hardly possible, as noted, to write a useful software system which does not include one or (usually) several cases of table searching. The investment needed to produce reusable modules is not hard to justify.
- As will be seen in more detail below, most searching algorithms follow a common pattern, providing what would seem to be an ideal basis for a reusable solution.

4.4 NON-TECHNICAL OBSTACLES

Why then is reuse not more common?

Most of the serious impediments to reuse are technical; removing them will be the subject of the following sections of this chapter (and of much of the rest of this book). But of course there are also some organizational, economical and political obstacles.

The NIH syndrome

An often quoted psychological obstacle to reuse is the famous Not Invented Here ("NIH") syndrome. Software developers, it is said, are individualists, who prefer to redo everything by themselves rather than rely on someone else's work.

This contention (commonly heard in managerial circles) is not borne out by experience. Software developers do not like useless work more than anyone else. When a good, well-publicized and easily accessible reusable solution is available, it gets reused.

Consider the typical case of lexical and syntactic analysis. Using parser generators such as the Lex-Yacc combination, it is much easier to produce a parser for a command language or a simple programming language than if you must program it from scratch. The result is clear: where such tools are available, competent software developers routinely reuse them. Writing your own tailor-made parser still makes sense in some cases, since the tools mentioned have their limitations. But the developers' reaction is usually to go by default to one of these tools; it is when you want to use a solution not based on the reusable mechanisms that you have to argue for it. This may in fact cause a new syndrome, the **reverse** of NIH, which we may call HIN (Habit Inhibiting Novelty): a useful but limited reusable solution, so entrenched that it narrows the developers' outlook and stifles innovation, becomes counter-productive. Try to convince some Unix developers to use a parser generator other than Yacc, and you may encounter HIN first-hand.

Something which may externally look like NIH does exist, but often it is simply the developers' understandably cautious reaction to new and unknown components. They may fear that bugs or other problems will be more difficult to correct than with a solution over which they have full control. Often such fears are justified by unfortunate earlier attempts at reusing components, especially if they followed from a management mandate to reuse at all costs, not accompanied by proper quality checks. If the new components are of good quality and provide a real service, fears will soon disappear.

What this means for the producer of reusable components is that quality is even more important here than for more ordinary forms of software. If the cost of a non-reusable, one-of-a-kind solution is N, the cost R of a solution relying on reusable components is never zero: there is a learning cost, at least the first time; developers may have to bend their software to accommodate the components; and they must write some interfacing software, however small, to call them. So even if the reusability savings

$$r = \frac{R}{N}$$

and other benefits of reuse are potentially great, you must also convince the candidate reusers that the reusable solution's quality is good enough to justify relinquishing control.

See [M 1995].

This explains why it is a mistake to target a company's reusability policy to the potential reusers (the *consumers*, that is to say the application developers). Instead you should put the heat on the *producers*, including people in charge of acquiring external components, to ensure the quality and usefulness of their offering. Preaching reuse to application developers, as some companies do by way of reusability policy, is futile: because application developers are ultimately judged by how effectively they produce their applications, they should and will reuse not because you tell them to but because you have done a good enough job with the reusable components (developed or acquired) that it will be *profitable* for their applications to rely on these components.

The economics of procurement

A potential obstacle to reuse comes from the procurement policy of many large corporations and government organizations, which tends to impede reusability efforts by focusing on short-term costs. US regulations, for example, make it hard for a government agency to pay a contractor for work that was not explicitly commissioned (normally as part of a Request For Proposals). Such rules come from a legitimate concern to protect taxpayers or shareholders, but can also discourage software builders from applying the crucial effort of *generalization* to transform good software into reusable components.

"GENERALIZATION", 28.5, page 928

On closer examination this obstacle does not look so insurmountable. As the concern for reusability spreads, there is nothing to prevent the commissioning agency from including in the RFP itself the requirement that the solution must be general-purpose and reusable, and the description of how candidate solutions will be evaluated against these criteria. Then the software developers can devote the proper attention to the generalization task and be paid for it.

Software companies and their strategies

Even if customers play their part in removing obstacles to reuse, a potential problem remains on the side of the contractors themselves. For a software company, there is a constant temptation to provide solutions that are purposely *not* reusable, for fear of not getting the next job from the customer — because if the result of the current job is too widely applicable the customer may not need a next job!

I once heard a remarkably candid exposé of this view after giving a talk on reuse and object technology. A high-level executive from a major software house came to tell me that, although intellectually he admired the ideas, he would never implement them in his own company, because that would be killing the goose that laid the golden egg: more than 90% of the company's business derived from renting manpower — providing analysts and programmers on assignment to customers — and the management's objective was to bring the figure to 100%. With such an outlook on software engineering, one is not likely to greet with enthusiasm the prospect of widely available libraries of reusable components.

The comment was notable for its frankness, but it triggered the obvious retort: if it is at all possible to build reusable components to replace some of the expensive services of a software house's consultants, sooner or later someone will build them. At that time a company that has refused to take this route, and is left with nothing to sell but its consultants' services, may feel sorry for having kept its head buried in the sand.

It is hard not to think here of the many engineering disciplines that used to be heavily labor-intensive but became industrialized, that is to say tool-based — with painful economic consequences for companies and countries that did not understand early enough what was happening. To a certain extent, object technology is bringing a similar change to the software trade. The choice between people and tools need not, however, be an exclusive one. The engineering part of software engineering is not identical to that of mass-production industries; humans will likely continue to play the key role in the software construction process. The aim of reuse is not to replace humans by tools (which is often, in spite of all claims, what has happened in other disciplines) but to change the distribution of what we entrust to humans and to tools. So the news is not all bad for a software company that has made its name through its consultants. In particular:

- In many cases developers using sophisticated reusable components may still benefit from the help of experts, who can advise them on how best to use the components. This leaves a meaningful role for software houses and their consultants.
- As will be discussed below, reusability is inseparable from extendibility: good reusable components will still be open for adaptation to specific cases. Consultants from a company that developed a library are in an ideal position to perform such tuning for individual customers. So selling components and selling services are not necessarily exclusive activities; a components business can serve as a basis for a service business.
- More generally, a good reusable library can play a strategic role in the policy of a successful software company, even if the company sells specific solutions rather than the library itself, and uses the library for internal purposes only. If the library covers the most common needs and provides an extendible basis for the more advanced cases, it can enable the company to gain a competitive edge in certain application areas by developing tailored solutions to customers' needs, faster and at lower cost than competitors who cannot rely on such a ready-made basis.

Accessing components

Another argument used to justify skepticism about reuse is the difficulty of the component management task: progress in the production of reusable software, it is said, would result in developers being swamped by so many components as to make their life worse than if the components were not available.

Cast in a more positive style, this comment should be understood as a warning to developers of reusable software that the best reusable components in the world are useless if nobody knows they exist, or if it takes too much time and effort to obtain them. The practical success of reusability techniques requires the development of adequate databases of components, which interested developers may search by appropriate keywords to find out quickly whether some existing component satisfies a particular need. Network services must also be available, allowing electronic ordering and immediate downloading of selected components.

These goals do raise technical and organizational problems. But we must keep things in proportion. Indexing, retrieving and delivering reusable components are engineering issues, to which we can apply known tools, in particular database technology; there is no reason why software components should be more difficult to manage than customer records, flight information or library books.

Reusability discussions used to delve forever into the grave question "how in the world are we going to make the components available to developers?". After the advances in networking of the past few years, such debates no longer appear so momentous. With the World-Wide Web, in particular, have appeared powerful search tools (AltaVista, Yahoo…) which have made it far easier to locate useful information, either on the Internet or on a company's Intranet. Even more advanced solutions (produced, one may expect, with the help of object technology) will undoubtedly follow. All this makes it increasingly clear that the really hard part of progress in reusability lies not in organizing reusable components, but in building the wretched things in the first place.

A note about component indexing

On the matter of indexing and retrieving components, a question presents itself, at the borderline between technical and organizational issues: how should we associate indexing information, such as keywords, with software components?

The Self-Documentation principle suggests that, as much as possible, information about a module — indexing information as well as other forms of module documentation — should appear in the module itself rather than externally. This leads to an important requirement on the notation that will be developed in part C of this book to write software components, called classes. Regardless of the exact form of these classes, we must equip ourselves with a mechanism to attach indexing information to each component.

"Self-Documentation", page 54.

The syntax is straightforward. At the beginning of a module text, you will be invited to write an **indexing clause** of the form

```
indexing
    index_word1: value, value, value…
    index_word2: value, value, value…
    …
    … Normal module definition (see part C) …
```

More details in "Indexing clauses", page 890.

Each *index_word* is an identifier; each *value* is a constant (integer, real etc.), an identifier, or some other basic lexical element.

There is no particular constraint on index words and values, but an industry, a standards group, an organization or a project may wish to define their own conventions. Indexing and retrieval tools can then extract this information to help software developers find components satisfying certain criteria.

As we saw in the discussion of Self-Documentation, storing such information in the module itself — rather than in an outside document or database — decreases the likelihood of including wrong information, and in particular of forgetting to update the information when updating the module (or conversely). Indexing clauses, modest as they may seem, play a major role in helping developers keep their software organized and register its properties so that others can find out about it.

Formats for reusable component distribution

Another question straddling the technical-organizational line is the form under which we should distribute reusable components: source or binary? This is a touchy issue, so we will limit ourselves to examining a few of the arguments on both sides.

"Using assertions for documentation: the short form of a class", page 390.

For a professional, for-profit software developer, it often seems desirable to provide buyers of reusable components with an interface description (the *short form* discussed in a later chapter) and the binary code for their platform of choice, but not the source form. This protects the developer's investment and trade secrets.

Binary is indeed the preferred form of distribution for commercial application programs, operating systems and other tools, including compilers, interpreters and development environments for object-oriented languages. In spite of recurring attacks on the very idea, emanating in particular from an advocacy group called the League for Programming Freedom, this mode of commercial software distribution is unlikely to recede much in the near future. But the present discussion is not about ordinary tools or application programs: it is about libraries of reusable software components. In that case one can also find some arguments in favor of source distribution.

For the component producer, an advantage of source distribution is that it eases porting efforts. You stay away from the tedious and unrewarding task of adapting software to the many incompatible platforms that exist in today's computer world, relying instead on the developers of object-oriented compilers and environments to do the job for you. (For the *consumer* this is of course a counter-argument, as installation from source will require more work and may cause unforeseen errors.)

Some compilers for object-oriented languages may let you retain some of the portability benefit without committing to full source availability: if the compiler uses C as intermediate generated code, as is often the case today, you can usually substitute portable C code for binary code. It is then not difficult to devise a tool that obscures the C form, making it almost as difficult to reverse-engineer as a binary form.

T. B. Steel: "A First Version of UNCOL", Joint Computer Conf., vol. 19, Winter 1961, pages 371-378.

Also note that at various stages in the history of software, dating back to UNCOL (UNiversal COmputing Language) in the late fifties, people have been defining low-level instruction formats that could be interpreted on any platform, and hence could provide a portable target for compilers. The ACE consortium of hardware and software companies was formed in 1988 for that purpose. Together with the Java language has come the notion of Java bytecode, for which interpreters are being developed on a number of platforms. But for the component producer such efforts at first represent more work, not less: until you have the double guarantee that the new format is available on every platform of interest *and* that it executes target code as fast as platform-specific solutions, you cannot forsake the old technology, and must simply add the new target code format to those you already support. So a solution that is advertized as an end-all to all portability problems actually creates, in the short term, more portability problems.

ISE's compilers use both C generation and bytecode generation.

Perhaps more significant, as an argument for source code distribution, is the observation that attempts to protect invention and trade secrets by removing the source form of the implementation may be of limited benefit anyway. Much of the hard work in the construction of a good reusable library lies not in the implementation but in the design of the components' interfaces; and that is the part that you are bound to release anyway. This is particularly clear in the world of data structures and algorithms, where most of the necessary techniques are available in the computing science literature. To design a successful library, you must embed these techniques in modules whose interface will make them useful to the developers of many different applications. This interface design is part of what you must release to the world.

Also note that, in the case of object-oriented modules, there are two forms of component reuse: as a client or, as studied in later chapters, through inheritance. The second form combines reuse with adaptation. Interface descriptions (short forms) are sufficient for client reuse, but not always for inheritance reuse.

Finally, the educational side: distributing the source of library modules is a good way to provide models of the producer's best engineering, useful to encourage consumers to develop their own software in a consistent style. We saw earlier that the resulting standardization is one of the benefits of reusability. Some of it will remain even if client developers only have access to the interfaces; but nothing beats having the full text.

The chapter on teaching object technology develops this point in "Apprenticeship", page 944.

Be sure to note that even if source is available it should not serve as the primary documentation tool: for that role, we continue to use the module interface.

This discussion has touched on some delicate economic issues, which condition in part the advent of an industry of software components and, more generally, the progress of the software field. How do we provide developers with a fair reward for their efforts and an acceptable degree of protection for their inventions, without hampering the legitimate interests of users? Here are two opposite views:

- At one end of the spectrum you will find the positions of the League for Programming Freedom: all software should be free and available in source form.

See the bibliographical notes.

- At the other end you have the idea of *superdistribution*, advocated by Brad Cox in several articles and a book. Superdistribution would allow users to duplicate software freely, charging them not for the purchase but instead for each use. Imagine a little counter attached to each software component, which rings up a few pennies every time you make use of the component, and sends you a bill at the end of the month. This seems to preclude distribution in source form, since it would be too easy to remove the counting instructions. Although JEIDA, a Japanese consortium of electronics companies, is said to be working on hardware and software mechanisms to support the concept, and although Cox has recently been emphasizing enforcement mechanisms built on regulations (like copyright) rather than technological devices, superdistribution still raises many technical, logistic, economic and psychological questions.

An assessment

Any comprehensive approach to reusability must, along with the technical aspects, deal with the organizational and economical issues: making reusability part of the software development culture, finding the right cost structure and the right format for component distribution, providing the appropriate tools for indexing and retrieving components. Not surprisingly, these issues have been the focus of some of the main reusability initiatives from governments and large corporations\, such as the STARS program of the US Department of Defense (*Software Technology for Adaptable*, *Reliable Systems*) and the "software factories" installed by some large Japanese companies.

Important as these questions are in the long term, they should not detract our attention from the main roadblocks, which are still technical. Success in reuse requires the right modular structures and the construction of quality libraries containing the tens of thousands of components that the industry needs.

The rest of this chapter concentrates on the first of these questions; it examines why common notions of module are not appropriate for large-scale reusability, and defines the requirements that a better solution — developed in the following chapters — must satisfy.

4.5 THE TECHNICAL PROBLEM

What should a reusable module look like?

Change and constancy

Software development, it was mentioned above, involves much repetition. To understand the technical difficulties of reusability we must understand the nature of that repetition.

Such an analysis reveals that although programmers do tend to do the same kinds of things time and time again, these are not *exactly* the same things. If they were, the solution would be easy, at least on paper; but in practice so many details may change as to defeat any simple-minded attempt at capturing the commonality.

A telling analogy is provided by the works of the Norwegian painter Edvard Munch, the majority of which may be seen in the museum dedicated to him in Oslo, the birthplace of Simula. Munch was obsessed with a small number of profound, essential themes: love, anguish, jealousy, dance, death… He drew and painted them endlessly, using the same pattern each time, but continually changing the technical medium, the colors, the emphasis, the size, the light, the mood.

Such is the software engineer's plight: time and again composing a new variation that elaborates on the same basic themes.

Take the example mentioned at the beginning of this chapter: table searching. True, the general form of a table searching algorithm is going to look similar each time: start at some position in the table *t*; then begin exploring the table from that position, each time checking whether the element found at the current position is the one being sought, and, if not, moving to another position. The process terminates when it has either found the element or probed all the candidate positions unsuccessfully. Such a general pattern is applicable to many possible cases of data representation and algorithms for table searching, including arrays (sorted or not), linked lists (sorted or not), sequential files, binary trees, B-trees and hash tables of various kinds.

It is not difficult to turn this informal description into an incompletely refined routine:

```
has (t: TABLE, x: ELEMENT): BOOLEAN is
        -- Is there an occurrence of x in t?
    local
        pos: POSITION
    do
        from
            pos := INITIAL_POSITION (x, t)
        until
            EXHAUSTED (pos, t) or else FOUND ( pos, x, t)
        loop
            pos := NEXT (pos, x, t)
        end

        Result := not EXHAUSTED (pos, t)
    end
```

(A few clarifications on the notation: **from** … **until** … **loop** … **end** describes a loop, initialized in the **from** clause, executing the **loop** clause zero or more times, and terminating as soon as the condition in the **until** clause is satisfied. *Result* denotes the value to be returned by the function. If you are not familiar with the **or else** operator, just accept it as if it were a boolean **or**.)

or else *is explained in "Non-strict boolean operators", page 454.*

Although the above text describes (through its lower-case elements) a general pattern of algorithmic behavior, it is not a directly executable routine since it contains (in upper case) some incompletely refined parts, corresponding to aspects of the table searching problem that depend on the implementation chosen: the type of table elements (*ELEMENT*), what position to examine first (*INITIAL_POSITION*), how to go from a candidate position to the next (*NEXT*), how to test for the presence of an element at a certain position (*FOUND*), how to determine that all interesting positions have been examined (*EXHAUSTED*).

Rather than a routine, then, the above text is a routine pattern, which you can only turn into an actual routine by supplying refinements for the upper-case parts.

The reuse-redo dilemma

All this variation highlights the problems raised by any attempt to come up with general-purpose modules in a given application area: how can we take advantage of the common pattern while accommodating the need for so much variation? This is not just an implementation problem: it is almost as hard to *specify* the module so that client modules can rely on it without knowing its implementation.

These observations point to the central problem of software reusability, which dooms simplistic approaches. Because of the versatility of software — its very softness — candidate reusable modules will not suffice if they are inflexible.

A frozen module forces you into the **reuse or redo** dilemma: reuse the module exactly as it is, or redo the job completely. This is often too limiting. In a typical situation, you discover a module that may provide you with a solution for some part of your current job, but not necessarily the exact solution. Your specific needs may require some adaptation of the module's original behavior. So what you will want to do in such a case is to reuse *and* redo: reuse some, redo some — or, you hope, reuse a lot and redo a little. Without this ability to combine reuse and adaptation, reusability techniques cannot provide a solution that satisfies the realities of practical software development.

So it is not by accident that almost every discussion of reusability in this book also considers extendibility (leading to the definition of the term "modularity", which covers both notions and provided the topic of the previous chapter). Whenever you start looking for answers to one of these quality requirements, you quickly encounter the other.

"The Open-Closed principle", page 57.

This duality between reuse and adaptation was also present in the earlier discussion of the Open-Closed principle, which pointed out that a successful software component must be usable as it stands (closed) while still adaptable (open).

The search for the right notion of module, which occupies the rest of this chapter and the next few, may be characterized as a constant attempt to reconcile reusability and extendibility, closure and openness, constancy and change, satisfying today's needs and trying to guess what tomorrow holds in store.

4.6 FIVE REQUIREMENTS ON MODULE STRUCTURES

How do we find module structures that will yield directly reusable components while preserving the possibility of adaptation?

The table searching issue and the *has* routine pattern obtained for it on the previous page illustrate the stringent requirements that any solution will have to meet. We can use this example to analyze what it takes to go from a relatively vague recognition of commonality between software variants to an actual set of reusable modules. Such a study will reveal five general issues:

- Type Variation.
- Routine Grouping.
- Implementation Variation.
- Representation Independence.
- Factoring Out Common Behaviors.

Type Variation

The *has* routine pattern assumes a table containing objects of a type *ELEMENT*. A particular refinement might use a specific type, such as *INTEGER* or *BANK_ACCOUNT*, to apply the pattern to a table of integers or bank accounts.

But this is not satisfactory. A reusable searching module should be applicable to many different types of element, without requiring reusers to perform manual changes to the software text. In other words, we need a facility for describing type-parameterized modules, also known more concisely as **generic** modules. Genericity (the ability for modules to be generic) will turn out to be an important part of the object-oriented method; an overview of the idea appears later in this chapter.

"Genericity", page 96.

Routine Grouping

Even if it had been completely refined and parameterized by types, the *has* routine pattern would not be quite satisfactory as a reusable component. How you search a table depends on how it was created, how elements are inserted, how they are deleted. So a searching routine is not enough by itself as a unit or reuse. A self-sufficient reusable module would need to include a set of routines, one for each of the operations cited — creation, insertion, deletion, searching.

This idea forms the basis for a form of module, the "package", found in what may be called the encapsulation languages: Ada, Modula-2 and relatives. More on this below.

Implementation Variation

The *has* pattern is very general; there is in practice, as we have seen, a wide variety of applicable data structures and algorithms. Such variety indeed that we cannot expect a single module to take care of all possibilities; it would be enormous. We will need a family of modules to cover all the different implementations.

A general technique for producing and using reusable modules will have to support this notion of module family.

Representation Independence

A general form of reusable module should enable clients to specify an operation without knowing how it is implemented. This requirement is called Representation Independence.

Assume that a client module *C* from a certain application system — an asset management program, a compiler, a geographical information system… — needs to determine whether a certain element *x* appears in a certain table *t* (of investments, of language keywords, of cities). Representation independence means here the ability for *C* to obtain this information through a call such as

present := *has* (*t*, *x*)

without knowing what kind of table *t* is at the time of the call. *C*'s author should only need to know that *t* is a table of elements of a certain type, and that *x* denotes an object of that type. Whether *t* is a binary search tree, a hash table or a linked list is irrelevant for him; he should be able to limit his concerns to asset management, compilation or geography. Selecting the appropriate search algorithm based on *t*'s implementation is the business of the table management module, and of no one else.

This requirement does not preclude letting clients choose a specific implementation when they create a data structure. But only one client will have to make this initial choice; after that, none of the clients that perform searches on *t* should ever have to ask what exact kind of table it is. In particular, the client *C* containing the above call may have received *t* from one of its own clients (as an argument to a routine call); then for *C* the name *t* is just an abstract handle on a data structure whose details it may not be able to access.

"Information Hiding", page 51.

You may view Representation Independence as an extension of the rule of Information Hiding, essential for smooth development of large systems: implementation decisions will often change, and clients should be protected. But Representation Independence goes further. Taken to its full consequences, it means protecting a module's clients against changes not only during the *project lifecycle* but also *during execution* — a much smaller time frame! In the example, we want *has* to adapt itself automatically to the run-time form of table *t*, even if that form has changed since the last call.

"Single Choice", page 61.

Satisfying Representation Independence will also help us towards a related principle encountered in the discussion of modularity: Single Choice, which directed us to stay away from multi-branch control structures that discriminate among many variants, as in

```
if "t is an array managed by open hashing" then
    "Apply open hashing search algorithm"
elseif "t is a binary search tree" then
    "Apply binary search tree traversal"
elseif
    (etc.)
end
```

"DYNAMIC BINDING", 14.4, page 480.

It would be equally unpleasant to have such a decision structure in the module itself (we cannot reasonably expect a table management module to know about all present and future variants) as to replicate it in every client. The solution is to hide the multi-branch choice completely from software developers, and have it performed automatically by the underlying run-time system. This will be the role of **dynamic binding**, a key component of the object-oriented approach, to be studied in the discussion of inheritance.

Factoring Out Common Behaviors

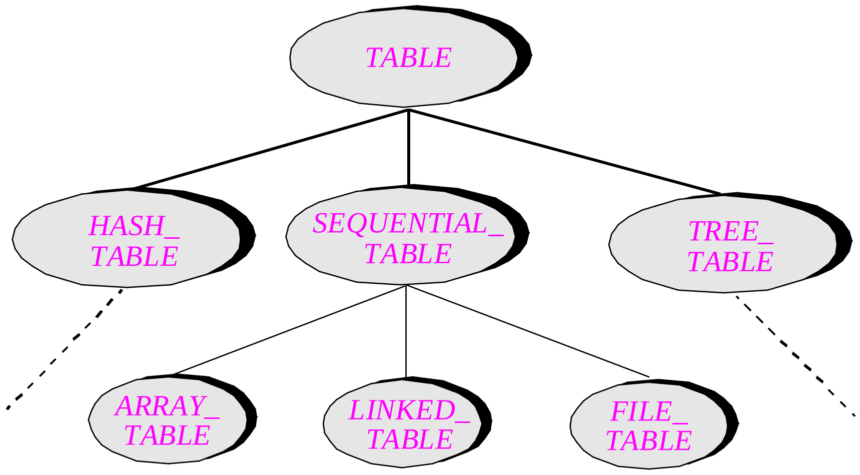
If Representation Independence reflects the client's view of reusability — the ability to ignore internal implementation details and variants –, the last requirement, Factoring Out Common Behaviors, reflects the view of the supplier and, more generally, the view of developers of reusable classes. Their goal will be to take advantage of any commonality that may exist within a family or sub-family of implementations.

The variety of implementations available in certain problem areas will usually demand, as noted, a solution based on a family of modules. Often the family is so large that it is natural to look for sub-families. In the table searching case a first attempt at classification might yield three broad sub-families:

- Tables managed by some form of hash-coding scheme.
- Tables organized as trees of some kind.
- Tables managed sequentially.

Each of these categories covers many variants, but it is usually possible to find significant commonality between these variants. Consider for example the family of sequential implementations — those in which items are kept and searched in the order of their original insertion.

Some possible table implementations



Possible representations for a sequential table include an array, a linked list and a file. But regardless of these differences, clients should be able, for any sequentially managed table, to examine the elements in sequence by moving a (fictitious) **cursor** indicating the position of the currently examined element. In this approach we may rewrite the searching routine for sequential tables as:

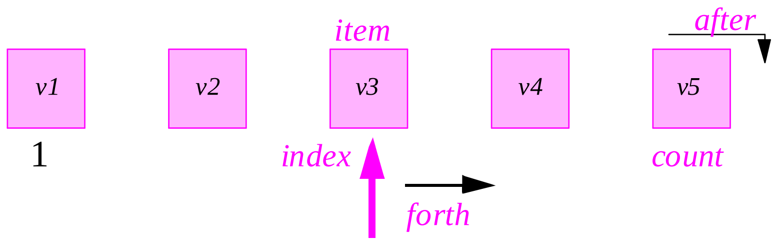
"ACTIVE DATA STRUCTURES", 23.4, page 774, will explore details of the cursor technique.

```
has (t: SEQUENTIAL_TABLE; x: ELEMENT): BOOLEAN is
        -- Is there an occurrence of x in t?
    do
        from start until
            after or else found (x)
        loop
            forth
        end
        Result := not after
    end
```

This form relies on four routines which any sequential table implementation will be able to provide:

- *start*, a command to move the cursor to the first element if any.
- *forth*, a command to advance the cursor by one position. (Support for *forth* is of course one of the prime characteristics of a sequential table implementation.)
- *after*, a boolean-valued query to determine if the cursor has moved past the last element; this will be true after a *start* if the table was empty.
- *found* (*x*), a boolean-valued query to determine if the element at cursor position has value *x*.

Sequential structure with cursor



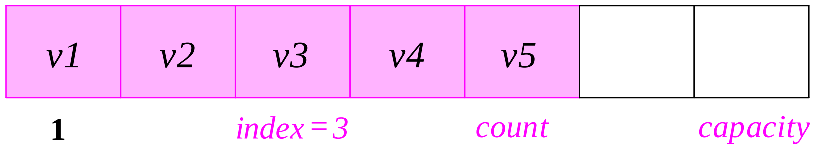
The general routine pattern was on page 82.

At first sight, the routine text for *has* at the bottom of the preceding page resembles the general routine pattern used at the beginning of this discussion, which covered searching in any table (not just sequential). But the new form is not a routine pattern any more; it is a true routine, expressed in a directly executable notation (the notation used to illustrate object-oriented concepts in part C of this book). Given appropriate implementations for the four operations *start*, *forth*, *after* and *found* which it calls, you can compile and execute the latest form of *has*.

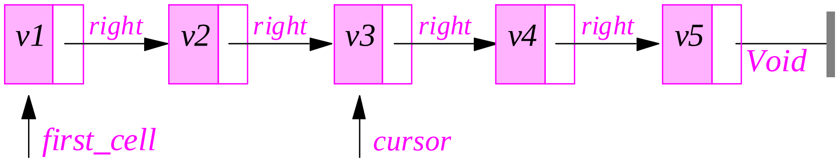
For each possible sequential table representation you will need a representation for the cursor. Three example representations are by an array, a linked list and a file.

The first uses an array of *capacity* items, the table occupying positions 1 to *count*. Then you may represent the cursor simply as an integer *index* ranging from 1 to *count + 1*. (The last value is needed to represent a cursor that has moved "*after*" the last item.)

Array representation of sequential table with cursor

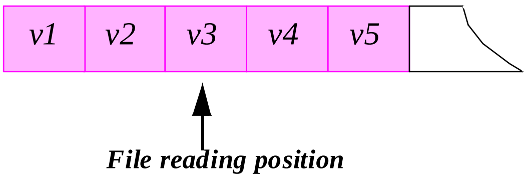


The second representation uses a linked list, where the first cell is accessible through a reference *first_cell* and each cell is linked to the next one through a reference *right*. Then you may represent the cursor as a reference *cursor*.



Linked list representation of sequential table with cursor

The third representation uses a sequential file, in which the cursor simply represents the current reading position.



Sequential file representation of a sequential table with cursor

The implementation of the four low-level operations *start*, *forth*, *after* and *found* will be different for each variant. The following table gives the implementation in each case. (The notation *t @ i* denotes the *i*-th element of array *t*, which would be written *t* [*i*] in Pascal or C; *Void* denotes a void reference; the Pascal notation *f*↑, for a file *f*, denotes the element at the current file reading position.)

	start	*forth*	*after*	*found* (*x*)
Array	*i* := 1	*i* := *i* + 1	*i* > *count*	*t* @ *i* = *x*
Linked list	*c* := *first_cell*	*c* := *c* . *right*	*c* = *Void*	*c* . *item* = *x*
File	*rewind*	*read*	*end_of_file*	*f*↑ = *x*

In this table index is abbreviated as i and cursor as c.

The challenge of reusability here is to avoid unneeded duplication of software by taking advantage of the commonality between variants. If identical or near-identical fragments appear in different modules, it will be difficult to guarantee their integrity and to ensure that changes or corrections get propagated to all the needed places; once again, configuration management problems may follow.

All sequential table variants share the *has* function, differing only by their implementation of the four lower-level operations. A satisfactory solution to the reusability problem must include the text of *has* in only one place, somehow associated with the general notion of sequential table independently of any choice of representation. To describe a new variant, you should not have to worry about *has* any more; all you will need to do is to provide the appropriae versions of *start*, *forth*, *after* and *found*.

4.7 TRADITIONAL MODULAR STRUCTURES

Together with the modularity requirements of the previous chapter, the five requirements of Type Variation, Routine Grouping, Implementation Variation, Representation Independence and Factoring Out Common Behaviors define what we may expect from our reusable components — abstracted modules.

Let us study the pre-O-O solutions to understand why they are not sufficient — but also what we should learn and keep from them in the object-oriented world.

Routines

The classical approach to reusability is to build libraries of routines. Here the term *routine* denotes a software unit that other units may call to execute a certain algorithm, using certain inputs, producing certain outputs and possibly modifying some other data elements. A calling unit will pass its inputs (and sometimes outputs and modified elements) in the form of *actual arguments*. A routine may also return output in the form of a *result*; in this case it is known as a *function*.

The terms *subroutine, subprogram* and *procedure* are also used instead of *routine*. The first two will not appear in this book except in the discussion of specific languages (the Ada literature talks about subprograms, and the Fortran literature about subroutines.) "Procedure" will be used in the sense of a routine which does not return a result, so that we have two disjoint categories of routine: procedures and functions. (In discussions of the C language the term "function" itself is sometimes used for the general notion of routine, but here it will always denote a routine that returns a result.)

Routine libraries have been successful in several application domains, in particular numerical computation, where excellent libraries have created some of the earliest success stories of reusability. Decomposition of systems into routines is also what one obtains through the method of top-down, functional decomposition. The routine library approach indeed seems to work well when you can identify a (possibly large) set of individual problems, subject to the following limitations:

R1 • Each problem admits a simple specification. More precisely, it is possible to characterize every problem instance by a small set of input and output arguments.

R2 • The problems are clearly distinct from each other, as the routine approach does not allow putting to good use any significant commonality that might exist — except by reusing some of the design.

R3 • No complex data structures are involved: you would have to distribute them among the routines using them, losing the conceptual autonomy of each module.

The table searching problem provides a good example of the limitations of subroutines. We saw earlier that a searching routine by itself does not have enough context to serve as a stand-alone reusable module. Even if we dismissed this objection, however, we would be faced with two equally unpleasant solutions:

- A single searching routine, which would try to cover so many different cases that it would require a long argument list and would be very complex internally.

- A large number of searching routines, each covering a specific case and differing from some others by only a few details in violation of the Factoring Out Common Behaviors requirement; candidate reusers could easily lose their way in such a maze.

More generally, routines are not flexible enough to satisfy the needs of reuse. We have seen the intimate connection between reusability and extendibility. A reusable module should be open to adaptation, but with a routine the only means of adaptation is to pass different arguments. This makes you a prisoner of the Reuse or Redo dilemma: either you like the routine as it is, or you write your own.

Packages

In the nineteen-seventies, with the progress of ideas on information hiding and data abstraction, a need emerged for a form of module more advanced than the routine. The result may be found in several design and programming languages of the period; the best known are CLU, Modula-2 and Ada. They all offer a similar form of module, known in Ada as the package. (CLU calls its variant the cluster, and Modula the module. This discussion will retain the Ada term.)

This approach is studied in detail, through the Ada notion of package, in chapter 33. Note again that by default "Ada" means Ada 83. (Ada 95 retains packages with a few additions.)

Packages are units of software decomposition with the following properties:

P1 • In accordance with the Linguistic Modular Units principle, "package" is a construct of the language, so that every package has a name and a clear syntactic scope.

P2 • Each package definition contains a number of declarations of related elements, such as routines and variables, hereafter called the **features** of the package.

P3 • Every package can specify precise access rights governing the use of its features by other packages. In other words, the package mechanism supports information hiding.

P4 • In a compilable language (one that can be used for implementation, not just specification and design) it is possible to compile packages separately.

Thanks to P3, packages deserve to be seen as abstracted modules. Their major contribution is P2, answering the Routine Grouping requirement. A package may contain any number of related operations, such as table creation, insertion, searching and deletion. It is indeed not hard to see how a package solution would work for our example problem. Here — in a notation adapted from the one used in the rest of this book for object-oriented software — is the sketch of a package *INTEGER_TABLE_HANDLING* describing a particular implementation of tables of integers, through binary trees:

```
package INTEGER_TABLE_HANDLING feature
    type INTBINTREE is
        record
                    -- Description of representation of a binary tree, for example:
            info: INTEGER
            left, right: INTBINTREE
        end
        new: INTBINTREE is
                -- Return a new INTBINTREE, properly initialized.
            do … end
        has (t: INTBINTREE; x: INTEGER): BOOLEAN is
                -- Does x appear in t?
            do … Implementation of searching operation … end
        put (t: INTBINTREE; x: INTEGER) is
                -- Insert x into t.
            do … end
        remove (t: INTBINTREE; x: INTEGER) is
                -- Remove x from t.
            do … end
    end -- package INTEGER_TABLE_HANDLING
```

This package includes the declaration of a type (*INTBINTREE*), and a number of routines representing operations on objects of that type. In this case there is no need for variable declarations in the package (although the routines may have local variables).

Client packages will now be able to manipulate tables by using the various features of *INTEGER_TABLE_HANDLING*. This assumes a syntactic convention allowing a client to use feature *f* from package *P*; let us borrow the CLU notation: *P*$*f*. Typical extracts from a client of *INTEGER_TABLE_HANDLING* may be of the form:

```
        -- Auxiliary declarations:
    x: INTEGER; b: BOOLEAN

        -- Declaration of t using a type defined in INTEGER_TABLE_HANDLING:
    t: INTEGER_TABLE_HANDLING$INTBINTREE

        -- Initialize t as a new table, created by function new of the package:
    t := INTEGER_TABLE_HANDLING$new

        -- Insert value of x into table, using procedure put from the package:
    INTEGER_TABLE_HANDLING$put (t, x)

        -- Assign True or False to b, depending on whether or not x appears in t
        -- for the search, use function has from the package:
    b := INTEGER_TABLE_HANDLING$has (t, x)
```

Note the need to invent two related names: one for the module, here *INTEGER_TABLE_HANDLING*, and one for its main data type, here *INTBINTREE*. One of the key steps towards object orientation will be to merge the two notions. But let us not anticipate.

A less important problem is the tediousness of having to write the package name (here *INTEGER_TABLE_HANDLING*) repeatedly. Languages supporting packages solve this problem by providing various syntactic shortcuts, such as the following Ada-like form:

```
        with INTEGER_TABLE_HANDLING then
            … Here has means INTEGER_TABLE_HANDLING$has, etc. …
        end
```

Another obvious limitation of packages of the above form is their failure to deal with the Type Variation issue: the module as given is only useful for tables of integers. We will shortly see, however, how to correct this deficiency by making packages generic.

The package mechanism provides information hiding by limiting clients' rights on features. The client shown on the preceding page was able to declare one of its own variables using the type *INTBINTREE* from its supplier, and to call routines declared in that supplier; but it has access neither to the internals of the type declaration (the **record** structure defining the implementation of tables) nor to the routine bodies (their **do** clauses). In addition, you can hide some features of the package (variables, types, routines) from clients, making them usable only within the text of the package.

"Supplier" is the inverse of "client". Here the supplier is INTEGER_ TABLE_HANDLING.

Languages supporting the package notion differ somewhat in the details of their information hiding mechanism. In Ada, for example, the internal properties of a type such as *INTBINTREE* will be accessible to clients unless you declare the type as **private**.

Often, to enforce information hiding, encapsulation languages will invite you to declare a package in two parts, interface and implementation, relegating such secret elements as the details of a type declaration or the body of a routine to the implementation part. Such a policy, however, results in extra work for the authors of supplier modules, forcing them to duplicate feature header declarations. With a better understanding of Information Hiding we do not need any of this. More in later chapters.

See "Using assertions for documentation: the short form of a class", page 390 and "Showing the interface", page 805.

Packages: an assessment

Compared to routines, the package mechanism brings a significant improvement to the modularization of software systems into abstracted modules. The possibility of gathering a number of features under one roof is useful for both supplier and client authors:

- The author of a supplier module can keep in one place and compile together all the software elements relating to a given concept. This facilitates debugging and change. In contrast, with separate subroutines there is always a risk of forgetting to update some of the routines when you make a design or implementation change; you might for example update *new*, *put* and *has* but forget *remove*.
- For client authors, it is obviously easier to find and use a set of related facilities if they are all in one place.

The advantage of packages over routines is particularly clear in cases such as our table example, where a package groups all the operations applying to a certain data structure.

But packages still do not provide a full solution to the issues of reusability. As noted, they address the Routine Grouping requirement; but they leave the others unanswered. In particular they offer no provision for factoring out commonality. You will have noted that *INTEGER_TABLE_HANDLING*, as sketched, relies on one specific choice of implementation, binary search trees. True, clients do not need to be concerned with this choice, thanks to information hiding. But a library of reusable components will need to provide modules for many different implementations. The resulting situation is easy to foresee: a typical package library will offer dozens of similar but never identical modules in a given area such as table management, with no way to take advantage of the commonality. To provide reusability to the clients, this technique sacrifices reusability on the suppliers' side.

Even on the clients' side, the situation is not completely satisfactory. Every use of a table by a client requires a declaration such as the above:

t: *INTEGER_TABLE_HANDLING$INTBINTREE*

forcing the client to choose a specific implementation. This defeats the Representation Independence requirement: client authors will have to know more about implementations of supplier notions than is conceptually necessary.

4.8 OVERLOADING AND GENERICITY

Two techniques, overloading and genericity, offer candidate solutions in the effort to bring more flexibility to the mechanisms just described. Let us study what they can contribute.

Syntactic overloading

Overloading is the ability to attach more than one meaning to a name appearing in a program.

The most common source of overloading is for variable names: in almost all languages, different variables may have the same name if they belong to different modules (or, in the Algol style of languages, different blocks within a module).

More relevant to this discussion is **routine overloading**, also known as operator overloading, which allows several routines to share the same name. This possibility is almost always available for arithmetic operators (hence the second name): the same notation, *a* + *b*, denotes various forms of addition depending on the types of *a* and *b* (integer, single-precision real, double-precision real). But most languages do not treat an operation such as "+" as a routine, and reserve it for predefined basic types — integer, real and the like. Starting with Algol 68, which allowed overloading the basic operators, several languages have extended the overloading facility beyond language built-ins to user-defined operations and ordinary routines.

In Ada, for example, a package may contain several routines with the same name, as long as the signatures of these routines are different, where the signature of a routine is defined here by the number and types of its arguments. (The general notion of signature also includes the type of the results, if any, but Ada resolves overloading on the basis of the arguments only.) For example, a package could contain several square functions:

The notation, compatible with the one in the rest of this book, is Ada-like rather than exact Ada. The REAL type is called FLOAT in Ada; semicolons have been removed.

square (*x*: *INTEGER*): *INTEGER* **is do** … **end**
square (*x*: *REAL*): *REAL* **is do** … **end**
square (*x*: *DOUBLE*): *DOUBLE* **is do** … **end**
square (*x*: *COMPLEX*): *COMPLEX* **is do** … **end**

Then, in a particular call of the form *square* (*y*), the type of *y* will determine which version of the routine you mean.

A package could similarly declare a number of search functions, all of the form

has (*t*: "SOME_TABLE_TYPE"; *x*: *ELEMENT*) **is do** … **end**

supporting various table implementations and differing by the actual type used in lieu of "SOME_TABLE_TYPE". The type of the first actual argument, in any client's call to *has*, suffices to determine which routine is intended.

These observations suggest a general characterization of routine overloading, which will be useful when we later want to contrast this facility with genericity:

Role of overloading

Routine overloading is a facility for clients. It makes it possible to write the same client text when using different implementations of a certain concept.

See the corresponding definition of genericity on page 97.

What does routine overloading really bring to our quest for reusability? Not much. It is a syntactic facility, relieving developers from having to invent different names for various implementations of an operation and, in essence, placing that burden on the compiler. But this does not solve any of the key issues of reusability. In particular, overloading does nothing to address Representation Independence. When you write the call

has (*t*, *x*)

you must have declared *t* and so (even if information hiding protects you from worrying about the details of each variant of the search algorithm) you must know exactly what kind of table *t* is! The only contribution of overloading is that you can use the same name in all cases. Without overloading each implementation would require a different name, as in

has_binary_tree (*t*, *x*)
has_hash (*t*, *x*)
has_linked (*t*, *x*)

Is the possibility of avoiding different names a benefit after all? Perhaps not. A basic rule of software construction, object-oriented or not, is the **principle of non-deception**: differences in semantics should be reflected by differences in the text of the software. This is essential to improve the understandability of software and minimize the risk of errors. If the *has* routines are different, giving them the same name may mislead a reader of the software into believing that they are the same. Better force a little more wordiness on the client (as with the above specific names) and remove any danger of confusion.

The further one looks into this style of overloading, the more limited it appears. The criterion used to disambiguate calls — the signature of argument lists — has no particular merit. It works in the above examples, where the various overloads of *square* and *has* are all of different signatures, but it is not difficult to think of many cases where the signatures would be the same. One of the simplest examples for overloading would seem to be, in a graphics system, a set of functions used to create new points, for example under the form

p1 := *new_point* (*u*, *v*)

More on syntactic overloading in "Multiple creation and overloading", page 239 and *"O-O development and overloading", page 564*.

There are two basic ways to specify a new point: through its cartesian coordinates *x* and *y* (the projections on the horizontal axis), and through its polar coordinates ρ and θ (the distance to the origin, and the angle with the horizontal axis). But if we overload function *new_point* we are in trouble, since both versions will have the signature

new_point (*p*, *q*: *REAL*): *POINT*

This example and many similar ones show that type signature, the criterion for disambiguating overloaded versions, is irrelevant. But no better one has been proposed.

The recent Java language regrettably includes the form of syntactic overloading just described, in particular to provide alternative ways to create objects.

Semantic overloading (a preview)

"DYNAMIC BINDING", 14.4, page 480.

The form of routine overloading described so far may be called **syntactic overloading**. The object-oriented method will bring a much more interesting technique, dynamic binding, which addresses the goal of Representation Independence. Dynamic binding may be called **semantic overloading**. With this technique, you will be able to write the equivalent of *has* (*t*, *x*), under a suitably adapted syntax, as a request to the machine that executes your software. The full meaning of the request is something like this:

How remarkably concise software languages are in comparison!

Dear Hardware-Software Machine:

Please look at what t is; I know that it must be a table, but not what table implementation its original creator chose — and to be honest about it I'd much rather remain in the dark. After all, my job is not table management but investment banking [*or compiling, or computer-aided-design etc.*]. *The chief table manager here is someone else. So find out for yourself about it and, once you have the answer, look up the proper algorithm for has for that particular kind of table. Then apply that algorithm to determine whether x appears in t, and tell me the result. I am eagerly waiting for your answer.*

I regret to inform you that, beyond the information that t is a table of some kind and x a potential element, you will not get any more help from me.

With my sincerest wishes,

Your friendly application developer.

Unlike syntactic overloading, such semantic overloading is a direct answer to the Representation Independence requirement. It still raises the specter of violating the principle of non-deception; the answer will be to use **assertions** to characterize the common semantics of a routine that has many different variants (for example, the common properties which characterize *has* under all possible table implementations).

Because semantic overloading, to work properly, requires the full baggage of object orientation, in particular inheritance, it is understandable that non-O-O languages such as Ada offer syntactic overloading as a partial substitute in spite of the problems mentioned above. In an object-oriented language, however, providing syntactic overloading on top of dynamic binding can be confusing, as is illustrated by the case of C++ and Java which both allow a class to introduce several routines with the same name, leaving it to the compiler and the human reader to disambiguate calls.

Genericity

Genericity is a mechanism for defining parameterized module patterns, whose parameters represent types.

This facility is a direct answer to the Type Variation issue. It avoids the need for many modules such as

INTEGER_TABLE_HANDLING
ELECTRON_TABLE_HANDLING
ACCOUNT_TABLE_HANDLING

by enabling you instead to write a single module pattern of the form

TABLE_HANDLING [*G*]

where *G* is a name meant to represent an arbitrary type and known as a **formal generic parameter**. (We may later encounter the need for two or more generic parameters, but for the present discussion we may limit ourselves to one.)

Such a parameterized module pattern is known as a **generic module**, although it is not really a module, only a blueprint for many possible modules. To obtain one of these actual modules, you must provide a type, known as an **actual generic parameter**, to replace *G*; the resulting (non-generic) modules are written for example

TABLE_HANDLING [*INTEGER*]
TABLE_HANDLING [*ELECTRON*]
TABLE_HANDLING [*ACCOUNT*]

using types *INTEGER*, *ELECTRON* and *ACCOUNT* respectively as actual generic parameters. This process of obtaining an actual module from a generic module (that is to say, from a module pattern) by providing a type as actual generic parameter will be known as **generic derivation**; the module itself will be said to be generically derived.

Two small points of terminology. First, generic derivation is sometimes called generic instantiation, a generically derived module then being called a generic instance. This terminology can cause confusion in an O-O context, since "instance" also denotes the run-time creation of objects (*instances*) from the corresponding types. So for genericity we will stick to the "derivation" terminology.

Another possible source of confusion is "parameter". A routine may have formal arguments, representing values which the routine's clients will provide in each call. The literature commonly uses the term parameter (formal, actual) as a synonym for argument (formal, actual). There is nothing wrong in principle with either term, but if we have both routines and genericity we need a clear convention to avoid any misunderstanding. The convention will be to use "argument" for routines only, and "parameter" (usually in the form "generic parameter" for further clarification) for generic modules only.

Internally, the declaration of the generic module *TABLE_HANDLING* will resemble that of *INTEGER_TABLE_HANDLING* above, except that it uses *G* instead of *INTEGER* wherever it refers to the type of table elements. For example:

To be compared with INTEGER_TABLE_HANDLING, page 90.

```
package TABLE_HANDLING [G] feature
    type BINARY_TREE is
        record
            info: G
            left, right: BINARY_TREE
        end
    has (t: BINARY_TREE; x: G): BOOLEAN
            -- Does x appear in t?
        do … end
    put (t: BINARY_TREE; x: G) is
        -- Insert x into t.
        do … end
    (Etc.)
end -- package TABLE_HANDLING
```

Chapter 10 discusses O-O genericity.

It is somewhat disturbing to see the type being declared as *BINARY_TREE*, and tempting to make it generic as well (something like *BINARY_TREE* [*G*]). There is no obvious way to achieve this in a package approach. Object technology, however, will merge the notions of module and type, so the temptation will be automatically fulfilled. We will see this when we study how to integrate genericity into the object-oriented world.

It is interesting to define genericity in direct contrast with the definition given earlier for overloading:

See the corresponding definition of overloading on page 94.

Role of genericity

Genericity is a facility for the authors of supplier modules. It makes it possible to write the same supplier text when using the same implementation of a certain concept, applied to different kinds of object.

What help does genericity bring us towards realizing the goals of this chapter? Unlike syntactic overloading, genericity has a real contribution to make since as noted above it solves one of the main issues, Type Variation. The presentation of object technology in part C of this book will indeed devote a significant role to genericity.

Basic modularity techniques: an assessment

We have obtained two main results. One is the idea of providing a single syntactic home, such as the package construct, for a set of routines that all manipulate similar objects. The other is genericity, which yields a more flexible form of module.

All this, however, only covers two of the reusability issues, Routine Grouping and Type Variation, and provides little help for the other three — Implementation Variation, Representation Independence and Factoring Out Common Behaviors. Genericity, in particular, does not suffice as a solution to the Factoring issue, since making a module generic defines two levels only: generic module patterns, parameterized and hence open to variation, but not directly usable; and individual generic derivations, usable directly but closed to further variation. This does not allow us to capture the fine differences that may exist between competing representations of a given general concept.

On Representation Independence, we have made almost no progress. None of the techniques seen so far — except for the short glimpse that we had of semantic overloading — will allow a client to use various implementations of a general notion without knowing which implementation each case will select.

To answer these concerns, we will have to turn to the full power of object-oriented concepts.

4.9 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Software development is a highly repetitive activity, involving frequent use of common patterns. But there is considerable variation in how these patterns are used and combined, defeating simplistic attempts to work from off-the-shelf components.
- Putting reusability into practice raises economical, psychological and organizational problems; the last category involves in particular building mechanisms to index, store and retrieve large numbers of reusable components. Even more important, however, are the underlying technical problems: commonly accepted notions of module are not adequate to support serious reusability.
- The major difficulty of reuse is the need to combine reuse with adaptation. The “reuse or redo” dilemma is not acceptable: a good solution must make it possible to retain some aspects of a reused module and adapt others.
- Simple approaches, such as reuse of personnel, reuse of designs, source code reuse, and subroutine libraries, have experienced some degree of success in specific contexts, but all fall short of providing the full potential benefits of reusability.
- The appropriate unit of reuse is some form of abstracted module, providing an encapsulation of a certain functionality through a well-defined interface.
- Packages provide a better encapsulation technique than routines, as they gather a data structure and the associated operations.
- Two techniques extend the flexibility of packages: routine overloading, or the reuse of the same name for more than one operation; genericity, or the availability of modules parameterized by types.
- Routine overloading is a syntactic facility which does not solve the important issues of reuse, and harms the readability of software texts.
- Genericity helps, but only deals with the issue of type variation.
- What we need: techniques for capturing commonalities within groups of related data structure implementations; and techniques for isolating clients from having to know the choice of supplier variants.

4.10 BIBLIOGRAPHICAL NOTES

The first published discussion of reusability in software appears to have been McIlroy's 1968 *Mass-Produced Software Components*, mentioned at the beginning of this chapter. His paper [McIlroy 1976] was presented in 1968 at the first conference on software engineering, convened by the NATO Science Affairs Committee. (1976 is the date of the proceedings, [Buxton 1976], whose publication was delayed by several years.) McIlroy advocated the development of an industry of software components. Here is an extract:

> *Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software…*
>
> *When we undertake to write a compiler, we begin by saying "What table mechanism shall we build?". Not "What mechanism shall we use?"…*
>
> *My thesis is that the software industry is weakly founded* [*in part because of*] *the absence of a software components subindustry… Such a components industry could be immensely successful.*

One of the important points argued in the paper was the necessity of module families, discussed above as one of the requirements on any comprehensive solution to reuse.

> *The most important characteristic of a software components industry is that it will offer families of* [*modules*] *for a given job.*

Rather than the word "module", McIlroy's text used "routine"; in light of this chapter's discussion, this is — with the hindsight of thirty years of further software engineering development — too restrictive.

A special issue of the IEEE *Transactions on Software Engineering* edited by Biggerstaff and Perlis [Biggerstaff 1984] was influential in bringing reusability to the attention of the software engineering community; see in particular, from that issue, [Jones 1984], [Horowitz 1984], [Curry 1984], [Standish 1984] and [Goguen 1984]. The same editors included all these articles (except the first mentioned) in an expanded two-volume collection [Biggerstaff 1989]. Another collection of articles on reuse is [Tracz 1988]. More recently Tracz collected a number of his *IEEE Computer* columns into a useful book [Tracz 1995] emphasizing the management aspects.

One approach to reuse, based on concepts from artificial intelligence, is embodied in the MIT Programmer's Apprentice project; see [Waters 1984] and [Rich 1989], reproduced in the first and second Biggerstaff-Perlis collections respectively. Rather than actual reusable modules, this system uses patterns (called *clichés* and *plans*) representing common program design strategies.

Ada is covered in chapter 33; see its "BIBLIOGRAPHICAL NOTES", 33.9, page 1097.

Three "encapsulation languages" were cited in the discussion of packages: Ada, Modula-2 and CLU. Ada is discussed in a later chapter, whose bibliography section gives references to Modula-2, CLU, as well as Mesa and Alphard, two other encapsulation languages of the "modular generation" of the seventies and early eighties. The equivalent of a package in Alphard was called a form.

An influential project of the nineteen-eighties, the US Department of Defense's STARS, emphasized reusability with a special concern for the organizational aspects of the problem, and using Ada as the language for software components. A number of contributions on this approach may be found in the proceedings of the 1985 STARS DoD-Industry conference [NSIA 1985].

The two best-known books on "design patterns" are [Gamma 1995] and [Pree 1994].

[Weiser 1987] is a plea for the distribution of software in source form. That article, however, downplays the need for abstraction; as pointed out in this chapter, it is possible to keep the source form available if needed but use a higher-level form as the default documentation for the users of a module. For different reasons, Richard Stallman, the creator of the League for Programming Freedom, has been arguing that the source form should always be available; see [Stallman 1992].

[Cox 1992] describes the idea of superdistribution.

A form of overloading was present in Algol 68 [van Wijngaarden 1975]; Ada (which extended it to routines), C++ and Java, all discussed in later chapters, make extensive use of the mechanism.

Genericity appears in Ada and CLU and in an early version of the Z specification language [Abrial 1980]; in that version the Z syntax is close to the one used for genericity in this book. The LPG language [Bert 1983] was explicitly designed to explore genericity. (The initials stand for Language for Programming Generically.)

The work cited at the beginning of this chapter as the basic reference on table searching is [Knuth 1973]. Among the many algorithms and data structures textbooks which cover the question, see [Aho 1974], [Aho 1983] or [M 1978].

Two books by the author of the present one explore further the question of reusability. *Reusable Software* [M 1994a], entirely devoted to the topic, provides design and implementation principles for building quality libraries, and the complete specification of a set of fundamental libraries. *Object Success* [M 1995] discusses management aspects, especially the areas in which a company interested in reuse should exert its efforts, and areas in which efforts will probably be wasted (such as preaching reuse to application developers, or rewarding reuse). See also a short article on the topic, [M 1996].