1

Software quality

Engineering seeks quality; software engineering is the production of quality software. This book introduces a set of techniques which hold the potential for remarkable improvements in the quality of software products.

Before studying these techniques, we must clarify their goals. Software quality is best described as a combination of several factors. This chapter analyzes some of these factors, shows where improvements are most sorely needed, and points to the directions where we shall be looking for solutions in the rest of our journey.

1.1 EXTERNAL AND INTERNAL FACTORS

We all want our software systems to be fast, reliable, easy to use, readable, modular, structured and so on. But these adjectives describe two different sorts of qualities.

On one side, we are considering such qualities as speed or ease of use, whose presence or absence in a software product may be detected by its users. These properties may be called **external** quality factors.

Under "users" we should include not only the people who actually interact with the final products, like an airline agent using a flight reservation system, but also those who purchase the software or contract out its development, like an airline executive in charge of acquiring or commissioning flight reservation systems. So a property such as the ease with which the software may be adapted to changes of specifications — defined later in this discussion as *extendibility* — falls into the category of external factors even though it may not be of immediate interest to such "end users" as the reservations agent.

Other qualities applicable to a software product, such as being modular, or readable, are **internal** factors, perceptible only to computer professionals who have access to the actual software text.

In the end, only external factors matter. If I use a Web browser or live near a computer-controlled nuclear plant, little do I care whether the source program is readable or modular if graphics take ages to load, or if a wrong input blows up the plant. But the key to achieving these external factors is in the internal ones: for the users to enjoy the visible qualities, the designers and implementers must have applied internal techniques that will ensure the hidden qualities.

The following chapters present of a set of modern techniques for obtaining internal quality. We should not, however, lose track of the global picture; the internal techniques are not an end in themselves, but a means to reach external software qualities. So we must start by looking at external factors. The rest of this chapter examines them.

1.2 A REVIEW OF EXTERNAL FACTORS

Here are the most important external quality factors, whose pursuit is the central task of object-oriented software construction.

Correctness

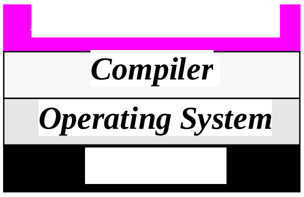
Definition: correctness

Correctness is the ability of software products to perform their exact tasks, as defined by their specification.

Correctness is the prime quality. If a system does not do what it is supposed to do, everything else about it — whether it is fast, has a nice user interface… — matters little.

But this is easier said than done. Even the first step to correctness is already difficult: we must be able to specify the system requirements in a precise form, by itself quite a challenging task.

Methods for ensuring correctness will usually be **conditional**. A serious software system, even a small one by today's standards, touches on so many areas that it would be impossible to guarantee its correctness by dealing with all components and properties on a single level. Instead, a layered approach is necessary, each layer relying on lower ones:

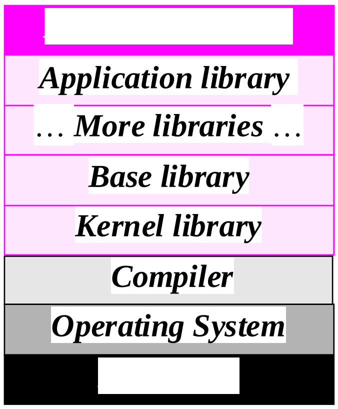


Layers in software development

In the conditional approach to correctness, we only worry about guaranteeing that each layer is correct *on the assumption* that the lower levels are correct. This is the only realistic technique, as it achieves separation of concerns and lets us concentrate at each stage on a limited set of problems. You cannot usefully check that a program in a high-level language X is correct unless you are able to assume that the compiler on hand implements X correctly. This does not necessarily mean that you trust the compiler blindly, simply that you separate the two components of the problem: compiler correctness, and correctness of your program relative to the language's semantics.

In the method described in this book, even more layers intervene: software development will rely on libraries of reusable components, which may be used in many different applications.

Layers in a development process that includes reuse



The conditional approach will also apply here: we should ensure that the libraries are correct and, separately, that the application is correct assuming the libraries are.

Many practitioners, when presented with the issue of software correctness, think about testing and debugging. We can be more ambitious: in later chapters we will explore a number of techniques, in particular typing and assertions, meant to help build software that is correct from the start — rather than debugging it into correctness. Debugging and testing remain indispensable, of course, as a means of double-checking the result.

It is possible to go further and take a completely formal approach to software construction. This book falls short of such a goal, as suggested by the somewhat timid terms "check", "guarantee" and "ensure" used above in preference to the word "prove". Yet many of the techniques described in later chapters come directly from the work on mathematical techniques for formal program specification and verification, and go a long way towards ensuring the correctness ideal.

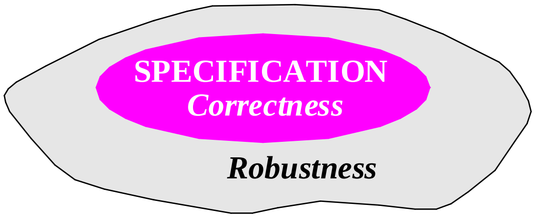
Robustness

Definition: robustness

Robustness is the ability of software systems to react appropriately to abnormal conditions.

Robustness complements correctness. Correctness addresses the behavior of a system in cases covered by its specification; robustness characterizes what happens outside of that specification.

Robustness versus correctness



As reflected by the wording of its definition, robustness is by nature a more fuzzy notion than correctness. Since we are concerned here with cases not covered by the specification, it is not possible to say, as with correctness, that the system should "perform its tasks" in such a case; were these tasks known, the abnormal case would become part of the specification and we would be back in the province of correctness.

This definition of "abnormal case" will be useful again when we study exception handling. It implies that the notions of normal and abnormal case are always relative to a certain specification; an abnormal case is simply a case that is not covered by the specification. If you widen the specification, cases that used to be abnormal become normal — even if they correspond to events such as erroneous user input that you would prefer not to happen. "Normal" in this sense does not mean "desirable", but simply "planned for in the design of the software". Although it may seem paradoxical at first that erroneous input should be called a normal case, any other approach would have to rely on subjective criteria, and so would be useless.

On exception handling see chapter 12.

There will always be cases that the specification does not explicitly address. The role of the robustness requirement is to make sure that if such cases do arise, the system does not cause catastrophic events; it should produce appropriate error messages, terminate its execution cleanly, or enter a so-called "graceful degradation" mode.

Extendibility

Definition: extendibility

Extendibility is the ease of adapting software products to changes of specification.

Software is supposed to be *soft*, and indeed is in principle; nothing can be easier than to change a program if you have access to its source code. Just use your favorite text editor.

The problem of extendibility is one of scale. For small programs change is usually not a difficult issue; but as software grows bigger, it becomes harder and harder to adapt. A large software system often looks to its maintainers as a giant house of cards in which pulling out any one element might cause the whole edifice to collapse.

We need extendibility because at the basis of all software lies some human phenomenon and hence fickleness. The obvious case of business software ("Management Information Systems"), where passage of a law or a company's acquisition may suddenly invalidate the assumptions on which a system rested, is not special; even in scientific computation, where we may expect the laws of physics to stay in place from one month to the next, our way of understanding and modeling physical systems will change.

Traditional approaches to software engineering did not take enough account of change, relying instead on an ideal view of the software lifecycle where an initial analysis stage freezes the requirements, the rest of the process being devoted to designing and building a solution. This is understandable: the first task in the progress of the discipline was to develop sound techniques for stating and solving fixed problems, before we could worry about what to do if the problem changes while someone is busy solving it. But now with the basic software engineering techniques in place it has become essential to recognize and address this central issue. Change is pervasive in software development: change of requirements, of our understanding of the requirements, of algorithms, of data representation, of implementation techniques. Support for change is a basic goal of object technology and a running theme through this book.

Although many of the techniques that improve extendibility may be introduced on small examples or in introductory courses, their relevance only becomes clear for larger projects. Two principles are essential for improving extendibility:

- *Design simplicity*: a simple architecture will always be easier to adapt to changes than a complex one.
- *Decentralization*: the more autonomous the modules, the higher the likelihood that a simple change will affect just one module, or a small number of modules, rather than triggering off a chain reaction of changes over the whole system.

The object-oriented method is, before anything else, a system architecture method which helps designers produce systems whose structure remains both simple (even for large systems) and decentralized. Simplicity and decentralization will be recurring themes in the discussions leading to object-oriented principles in the following chapters.

Reusability

Definition: reusability

Reusability is the ability of software elements to serve for the construction of many different applications.

The need for reusability comes from the observation that software systems often follow similar patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before. By capturing such a pattern, a reusable software element will be applicable to many different developments.

Reusability has an influence on all other aspects of software quality, for solving the reusability problem essentially means that less software must be written, and hence that more effort may be devoted (for the same total cost) to improving the other factors, such as correctness and robustness.

Here again is an issue that the traditional view of the software lifecycle had not properly recognized, and for the same historical reason: you must find ways to solve one problem before you worry about applying the solution to other problems. But with the growth of software and its attempts to become a true industry the need for reusability has become a pressing concern.

Chapter 4.

Reusability will play a central role in the discussions of the following chapters, one of which is in fact devoted entirely to an in-depth examination of this quality factor, its concrete benefits, and the issues it raises.

Compatibility

Definition: compatibility

Compatibility is the ease of combining software elements with others.

Compatibility is important because we do not develop software elements in a vacuum: they need to interact with each other. But they too often have trouble interacting because they make conflicting assumptions about the rest of the world. An example is the wide variety of incompatible file formats supported by many operating systems. A program can directly use another's result as input only if the file formats are compatible.

Lack of compatibility can yield disaster. Here is an extreme case:

DALLAS — Last week, AMR, the parent company of American Airlines, Inc., said it fell on its sword trying to develop a state-of-the-art, industry-wide system that could also handle car and hotel reservations.

AMR cut off development of its new Confirm reservation system only weeks after it was supposed to start taking care of transactions for partners Budget Rent-A-Car, Hilton Hotels Corp. and Marriott Corp. Suspension of the $125 million, 4-year-old project translated into a $165 million pre-tax charge against AMR's earnings and fractured the company's reputation as a pacesetter in travel technology. […]

As far back as January, the leaders of Confirm discovered that the labors of more than 200 programmers, systems analysts and engineers had apparently been for naught. The main pieces of the massive project — requiring 47,000 pages to describe — had been developed separately, by different methods. When put together, they did not work with each other. When the developers attempted to plug the parts together, they could not. Different "modules" could not pull the information needed from the other side of the bridge.

AMR Information Services fired eight senior project members, including the team leader. […] *In late June, Budget and Hilton said they were dropping out.*

San Jose (Calif.) Mercury News, July 20, 1992. Quoted in the "comp. risks" Usenet newsgroup, 13.67, July 1992. Slightly abridged.

The key to compatibility lies in homogeneity of design, and in agreeing on standardized conventions for inter-program communication. Approaches include:

- Standardized file formats, as in the Unix system, where every text file is simply a sequence of characters.
- Standardized data structures, as in Lisp systems, where all data, and programs as well, are represented by binary trees (called lists in Lisp).
- Standardized user interfaces, as on various versions of Windows, OS/2 and MacOS, where all tools rely on a single paradigm for communication with the user, based on standard components such as windows, icons, menus etc.

More general solutions are obtained by defining standardized access protocols to all important entities manipulated by the software. This is the idea behind abstract data types and the object-oriented approach, as well as so-called *middleware* protocols such as CORBA and Microsoft's OLE-COM (ActiveX).

On abstract data types see chapter 6.

Efficiency

Definition: efficiency

Efficiency is the ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.

Almost synonymous with efficiency is the word “performance”. The software community shows two typical attitudes towards efficiency:

- Some developers have an obsession with performance issues, leading them to devote a lot of efforts to presumed optimizations.
- But a general tendency also exists to downplay efficiency concerns, as evidenced by such industry lore as “make it right before you make it fast” and “next year’s computer model is going to be 50% faster anyway”.

It is not uncommon to see the same person displaying these two attitudes at different times, as in a software case of split personality (Dr. Abstract and Mr. Microsecond).

Where is the truth? Clearly, developers have often shown an exaggerated concern for micro-optimization. As already noted, efficiency does not matter much if the software is not correct (suggesting a new dictum, “*do not worry how fast it is unless it is also right*”, close to the previous one but not quite the same). More generally, the concern for efficiency must be balanced with other goals such as extendibility and reusability; extreme optimizations may make the software so specialized as to be unfit for change and reuse. Furthermore, the ever growing power of computer hardware does allow us to have a more relaxed attitude about gaining the last byte or microsecond.

All this, however, does not diminish the importance of efficiency. No one likes to wait for the responses of an interactive system, or to have to purchase more memory to run a program. So offhand attitudes to performance include much posturing; if the final system is so slow or bulky as to impede usage, those who used to declare that “speed is not that important” will not be the last to complain.

This issue reflects what I believe to be a major characteristic of software engineering, not likely to move away soon: software construction is difficult precisely because it requires taking into account many different requirements, some of which, such as correctness, are abstract and conceptual, whereas others, such as efficiency, are concrete and bound to the properties of computer hardware.

For some scientists, software development is a branch of mathematics; for some engineers, it is a branch of applied technology. In reality, it is both. The software developer must reconcile the abstract concepts with their concrete implementations, the mathematics of correct computation with the time and space constraints deriving from physical laws and from limitations of current hardware technology. This need to please the angels as well as the beasts may be the central challenge of software engineering.

The constant improvement in computer power, impressive as it is, is not an excuse for overlooking efficiency, for at least three reasons:

- Someone who purchases a bigger and faster computer wants to see some actual benefit from the extra power — to handle new problems, process previous problems faster, or process bigger versions of the previous problems in the same amount of time. Using the new computer to process the previous problems in the same amount of time will not do!

- One of the most visible effects of advances in computer power is actually to *increase* the lead of good algorithms over bad ones. Assume that a new machine is twice as fast as the previous one. Let n be the size of the problem to solve, and N the maximum n that can be handled by a certain algorithm in a given time. Then if the algorithm is in O (n), that is to say, runs in a time proportional to n, the new machine will enable you to handle problem sizes of about $2 * N$ for large N. For an algorithm in O (n^2) the new machine will only yield a 41% increase of N. An algorithm in O (2^n), similar to certain combinatorial, exhaustive-search algorithms, would just add one to N — not much of an improvement for your money.

- In some cases efficiency may affect correctness. A specification may state that the computer response to a certain event must occur no later than a specified time; for example, an in-flight computer must be prepared to detect and process a message from the throttle sensor fast enough to take corrective action. This connection between efficiency and correctness is not restricted to applications commonly thought of as “real time”; few people are interested in a weather forecasting model that takes twenty-four hours to predict the next day’s weather.

  Another example, although perhaps less critical, has been of frequent annoyance to me: a window management system that I used for a while was sometimes too slow to detect that the mouse cursor had moved from a window to another, so that characters typed at the keyboard, meant for a certain window, would occasionally end up in another.

  In this case a performance limitation causes a violation of the specification, that is to say of correctness, which even in seemingly innocuous everyday applications can cause nasty consequences: think of what can happen if the two windows are used to send electronic mail messages to two different correspondents. For less than this marriages have been broken, even wars started.

Because this book is focused on the concepts of object-oriented software engineering, not on implementation issues, only a few sections deal explicitly with the associated performance costs. But the concern for efficiency will be there throughout. Whenever the discussion presents an object-oriented solution to some problem, it will make sure that the solution is not just elegant but also efficient; whenever it introduces some new O-O mechanism, be it garbage collection (and other approaches to memory management for object-oriented computation), dynamic binding, genericity or repeated inheritance, it will do so based on the knowledge that the mechanism may be implemented at a reasonable cost in time and in space; and whenever appropriate it will mention the performance consequences of the techniques studied.

Efficiency is only one of the factors of quality; we should not (like some in the profession) let it rule our engineering lives. But it is a factor, and must be taken into consideration, whether in the construction of a software system or in the design of a programming language. If you dismiss performance, performance will dismiss you.

Portability

Definition: portability

Portability is the ease of transferring software products to various hardware and software environments.

Portability addresses variations not just of the physical hardware but more generally of the **hardware-software machine**, the one that we really program, which includes the operating system, the window system if applicable, and other fundamental tools. In the rest of this book the word “platform” will be used to denote a type of hardware-software machine; an example of platform is “Intel X86 with Windows NT” (known as “Wintel”).

Many of the existing platform incompatibilities are unjustified, and to a naïve observer the only explanation sometimes seems to be a conspiracy to victimize humanity in general and programmers in particular. Whatever its causes, however, this diversity makes portability a major concern for both developers and users of software.

Ease of use

Definition: ease of use

Ease of use is the ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.

The definition insists on the various levels of expertise of potential users. This requirement poses one of the major challenges to software designers preoccupied with ease of use: how to provide detailed guidance and explanations to novice users, without bothering expert users who just want to get right down to business.

As with many of the other qualities discussed in this chapter, one of the keys to ease of use is structural simplicity. A well-designed system, built according to a clear, well thought-out structure, will tend to be easier to learn and use than a messy one. The condition is not sufficient, of course (what is simple and clear to the designer may be difficult and obscure to users, especially if explained in designer’s rather than user’s terms), but it helps considerably.

This is one of the areas where the object-oriented method is particularly productive; many O-O techniques, which appear at first to address design and implementation, also yield powerful new interface ideas that help the end users. Later chapters will introduce several examples.

Software designers preoccupied with ease of use will also be well-advised to consider with some mistrust the precept most frequently quoted in the user interface literature, from an early article by Hansen: ***know the user***. The argument is that a good designer must make an effort to understand the system's intended user community. This view ignores one of the features of successful systems: they always outgrow their initial audience. (Two old and famous examples are Fortran, conceived as a tool to solve the problem of the small community of engineers and scientists programming the IBM 704, and Unix, meant for internal use at Bell Laboratories.) A system designed for a specific group will rely on assumptions that simply do not hold for a larger audience.

See Wilfred J. Hansen, "User Engineering Principles for Interactive Systems", Proceedings of FJCC 39, AFIPS Press, Montvale (*NJ*), *1971, pp 523-532.*

Good user interface designers follow a more prudent policy. They make as limited assumptions about their users as they can. When you design an interactive system, you may expect that users are members of the human race and that they can read, move a mouse, click a button, and type (slowly); not much more. If the software addresses a specialized application area, you may perhaps assume that your users are familiar with its basic concepts. But even that is risky. To reverse-paraphrase Hansen's advice:

User Interface Design principle

Do not pretend you know the user; you don't.

Functionality

Definition: functionality

Functionality is the extent of possibilities provided by a system.

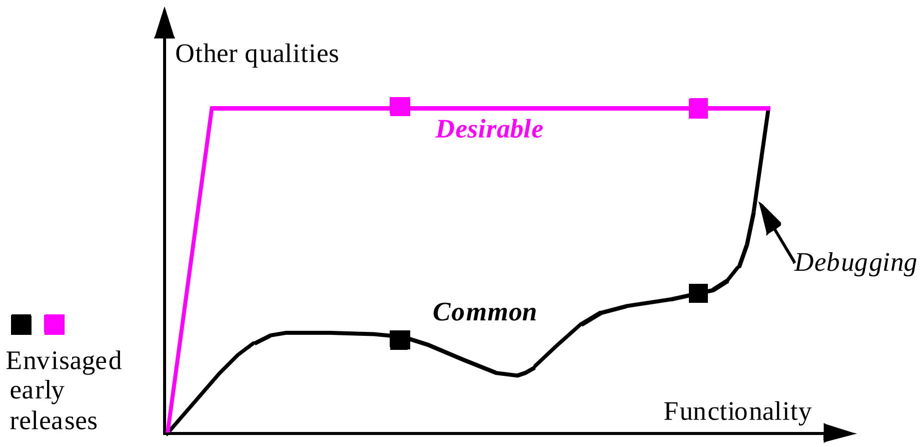
One of the most difficult problems facing a project leader is to know how much functionality is enough. The pressure for more facilities, known in industry parlance as *featurism* (often "*creeping featurism*"), is constantly there. Its consequences are bad for internal projects, where the pressure comes from users within the same company, and worse for commercial products, as the most prominent part of a journalist's comparative review is often the table listing side by side the features offered by competing products.

Featurism is actually the combination of two problems, one more difficult than the other. The easier problem is the loss of consistency that may result from the addition of new features, affecting its ease of use. Users are indeed known to complain that all the "bells and whistles" of a product's new version make it horrendously complex. Such comments should be taken with a grain of salt, however, since the new features do not come out of nowhere: most of the time they have been requested by users — *other* users. What to me looks like a superfluous trinket may be an indispensable facility to you.

The solution here is to work again and again on the consistency of the overall product, trying to make everything fit into a general mold. A good software product is based on a small number of powerful ideas; even if it has many specialized features, they should all be explainable as consequences of these basic concepts. The "grand plan" must be visible, and everything should have its place in it.

The more difficult problem is to avoid being so focused on features as to forget the other qualities. Projects commonly make such a mistake, a situation vividly pictured by Roger Osmond in the form of two possible paths to a project's completion:

***Osmond's curves; after* [Osmond 1995]**



The bottom curve (black) is all too common: in the hectic race to add more features, the development loses track of the overall quality. The final phase, intended to get things right at last, can be long and stressful. If, under users' or competitors' pressure, you are forced to release the product early — at stages marked by black squares in the figure — the outcome may be damaging to your reputation.

What Osmond suggests (the color curve) is, aided by the quality-enhancing techniques of O-O development, to maintain the quality level constant throughout the project for all aspects but functionality. You just do not compromise on reliability, extendibility and the like: you refuse to proceed with new features until you are happy with the features you have.

This method is tougher to enforce on a day-to-day basis because of the pressures mentioned, but yields a more effective software process and often a better product in the end. Even if the final result is the same, as assumed in the figure, it should be reached sooner (although the figure does not show time). Following the suggested path also means that the decision to release an early version — at one of the points marked by colored squares in the figure — becomes, if not easier, at least simpler: it will be based on your assessment of whether what you have so far covers a large enough share of the full feature set to attract prospective customers rather than drive them away. The question "is it good enough?" (as in "will it not crash?") should not be a factor.

As any reader who has led a software project will know, it is easier to approve such advice than to apply it. But every project should strive to follow the approach represented by the better one of the two Osmond curves. It goes well with the *cluster model* introduced in a later chapter as the general scheme for disciplined object-oriented development.

Timeliness

Definition: timeliness

Timeliness is the ability of a software system to be released when or before its users want it.

Timeliness is one of the great frustrations of our industry. A great software product that appears too late might miss its target altogether. This is true in other industries too, but few evolve as quickly as software.

Timeliness is still, for large projects, an uncommon phenomenon. When Microsoft announced that the latest release of its principal operating system, several years in the making, would be delivered one month early, the event was newsworthy enough to make (at the top of an article recalling the lengthy delays that affected earlier projects) the front-page headline of *ComputerWorld*.

"NT 4.0 Beats Clock", ComputerWorld, vol. 30, no. 30, 22 July 1996.

Other qualities

Other qualities beside the ones discussed so far affect users of software systems and the people who purchase these systems or commission their development. In particular:

- **Verifiability** is the ease of preparing acceptance procedures, especially test data, and procedures for detecting failures and tracing them to errors during the validation and operation phases.
- **Integrity** is the ability of software systems to protect their various components (programs, data) against unauthorized access and modification.
- **Repairability** is the ability to facilitate the repair of defects.
- **Economy**, the companion of timeliness, is the ability of a system to be completed on or below its assigned budget.

About documentation

In a list of software quality factors, one might expect to find the presence of good documentation as one of the requirements. But this is not a separate quality factor; instead, the need for documentation is a consequence of the other quality factors seen above. We may distinguish between three kinds of documentation:

- The need for *external* documentation, which enables users to understand the power of a system and use it conveniently, is a consequence of the definition of ease of use.
- The need for *internal* documentation, which enables software developers to understand the structure and implementation of a system, is a consequence of the extendibility requirement.
- The need for *module interface* documentation, enabling software developers to understand the functions provided by a module without having to understand its implementation, is a consequence of the reusability requirement. It also follows from extendibility, as module interface documentation makes it possible to determine whether a certain change need affect a certain module.

Rather than treating documentation as a product separate from the software proper, it is preferable to make the software as self-documenting as possible. This applies to all three kinds of documentation:

- By including on-line "help" facilities and adhering to clear and consistent user interface conventions, you alleviate the task of the authors of user manuals and other forms of external documentation.
- A good implementation language will remove much of the need for internal documentation if it favors clarity and structure. This will be one of the major requirements on the object-oriented notation developed throughout this book.
- The notation will support information hiding and other techniques (such as assertions) for separating the interface of modules from their implementation. It is then possible to use tools to produce module interface documentation automatically from module texts. This too is one of the topics studied in detail in later chapters.

All these techniques lessen the role of traditional documentation, although of course we cannot expect them to remove it completely.

Tradeoffs

In this review of external software quality factors, we have encountered requirements that may conflict with one another.

How can one get *integrity* without introducing protections of various kinds, which will inevitably hamper *ease of use*? *Economy* often seems to fight with *functionality*. Optimal *efficiency* would require perfect adaptation to a particular hardware and software environment, which is the opposite of *portability*, and perfect adaptation to a specification, where *reusability* pushes towards solving problems more general than the one initially given. *Timeliness* pressures might tempt us to use "Rapid Application Development" techniques whose results may not enjoy much *extendibility*.

Although it is in many cases possible to find a solution that reconciles apparently conflicting factors, you will sometimes need to make tradeoffs. Too often, developers make these tradeoffs implicitly, without taking the time to examine the issues involved and the various choices available; efficiency tends to be the dominating factor in such silent decisions. A true software engineering approach implies an effort to state the criteria clearly and make the choices consciously.

Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: correctness. There is never any justification for compromising correctness for the sake of other concerns such as efficiency. If the software does not perform its function, the rest is useless.

Key concerns

All the qualities discussed above are important. But in the current state of the software industry, four stand out:

- *Correctness* and *robustness*: it is still too difficult to produce software without defects (bugs), and too hard to correct the defects once they are there. Techniques for improving correctness and robustness are of the same general flavors: more systematic approaches to software construction; more formal specifications; built-in checks throughout the software construction process (not just after-the-fact testing and debugging); better language mechanisms such as static typing, assertions, automatic memory management and disciplined exception handling, enabling developers to state correctness and robustness requirements, and enabling tools to detect inconsistencies before they lead to defects. Because of this closeness of correctness and robustness issues, it is convenient to use a more general term, **reliability**, to cover both factors.

- *Extendibility* and *reusability*: software should be easier to change; the software elements we produce should be more generally applicable, and there should exist a larger inventory of general-purpose components that we can reuse when developing a new system. Here again, similar ideas are useful for improving both qualities: any idea that helps produce more decentralized architectures, in which the components are self-contained and only communicate through restricted and clearly defined channels, will help. The term **modularity** will cover reusability and extendibility.

As studied in detail in subsequent chapters, the object-oriented method can significantly improve these four quality factors — which is why it is so attractive. It also has significant contributions to make on other aspects, in particular:

- *Compatibility*: the method promotes a common design style and standardized module and system interfaces, which help produce systems that will work together.

- *Portability*: with its emphasis on abstraction and information hiding, object technology encourages designers to distinguish between specification and implementation properties, facilitating porting efforts. The techniques of polymorphism and dynamic binding will even make it possible to write systems that automatically adapt to various components of the hardware-software machine, for example different window systems or different database management systems.

- *Ease of use*: the contribution of O-O tools to modern interactive systems and especially their user interfaces is well known, to the point that it sometimes obscures other aspects (ad copy writers are not the only people who call “object-oriented” any system that uses icons, windows and mouse-driven input).

- *Efficiency*: as noted above, although the extra power or object-oriented techniques at first appears to carry a price, relying on professional-quality reusable components can often yield considerable performance improvements.

- *Timeliness*, *economy* and *functionality*: O-O techniques enable those who master them to produce software faster and at less cost; they facilitate addition of functions, and may even of themselves suggest new functions to add.

In spite of all these advances, we should keep in mind that the object-oriented method is not a panacea, and that many of the habitual issues of software engineering remain. Helping to address a problem is not the same as solving the problem.

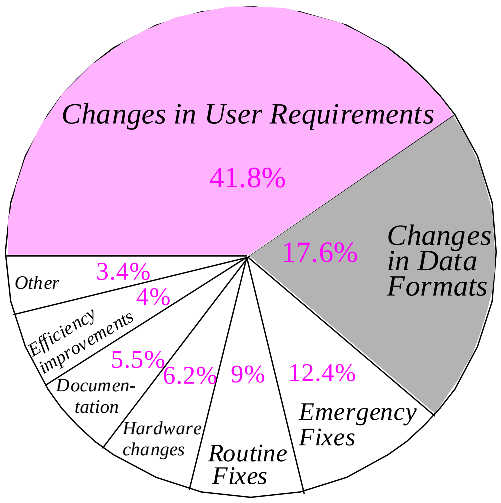
1.3 ABOUT SOFTWARE MAINTENANCE

The list of factors did not include a frequently quoted quality: maintainability. To understand why, we must take a closer look at the underlying notion, maintenance.

Maintenance is what happens after a software product has been delivered. Discussions of software methodology tend to focus on the development phase; so do introductory programming courses. But it is widely estimated that 70% of the cost of software is devoted to maintenance. No study of software quality can be satisfactory if it neglects this aspect.

What does "maintenance" mean for software? A minute's reflection shows this term to be a misnomer: a software product does not wear out from repeated usage, and thus need not be "maintained" the way a car or a TV set does. In fact, the word is used by software people to describe some noble and some not so noble activities. The noble part is modification: as the specifications of computer systems change, reflecting changes in the external world, so must the systems themselves. The less noble part is late debugging: removing errors that should never have been there in the first place.

***Breakdown of maintenance costs. Source*: [Lientz 1980]**



The above chart, drawn from a milestone study by Lientz and Swanson, sheds some light on what the catch-all term of maintenance really covers. The study surveyed 487 installations developing software of all kinds; although it is a bit old, more recent publications confirm the same general results. It shows the percentage of maintenance costs going into each of a number of maintenance activities identified by the authors.

More than two-fifths of the cost is devoted to user-requested extensions and modifications. This is what was called above the noble part of maintenance, which is also the inevitable part. The unanswered question is how much of the overall effort the industry could spare if it built its software from the start with more concern for extendibility. We may legitimately expect object technology to help.

The second item in decreasing order of percentage cost is particularly interesting: effect of changes in data formats. When the physical structure of files and other data items change, programs must be adapted. For example, when the US Postal Service, a few years ago, introduced the "5+4" postal code for large companies (using nine digits instead of five), numerous programs that dealt with addresses and "knew" that a postal code was exactly five digits long had to be rewritten, an effort which press accounts estimated in the hundreds of millions of dollars.

For another example, see "How long is a middle initial?", page 125.

Many readers will have received the beautiful brochures for a set of conferences — not a single event, but a sequence of sessions in many cities — devoted to the "millennium problem": how to go about upgrading the myriads of date-sensitive programs whose authors never for a moment thought that a date could exist beyond the twentieth century. The zip code adaptation effort pales in comparison. Jorge Luis Borges would have liked the idea: since presumably few people care about what will happen on 1 January 3000, this must be the tiniest topic to which a conference series, or for that matter a conference, has been or will ever be devoted in the history of humanity: *a single decimal digit*.

The issue is not that some part of the program knows the physical structure of data: this is inevitable since the data must eventually be accessed for internal handling. But with traditional design techniques this knowledge is spread out over too many parts of the system, causing unjustifiably large program changes if some of the physical structure changes — as it inevitably will. In other words, if postal codes go from five to nine digits, or dates require one more digit, it is reasonable to expect that a program manipulating the codes or the dates will need to be adapted; what is not acceptable is to have the knowledge of the exact length of the data plastered all across the program, so that changing that length will cause program changes of a magnitude out of proportion with the conceptual size of the specification change.

The theory of abstract data types will provide the key to this problem, by allowing programs to access data by external properties rather than physical implementation.

Chapter 6 covers abstract data types in detail.

Another significant item in the distribution of activities is the low percentage (5.5%) of documentation costs. Remember that these are costs of tasks done at maintenance time. The observation here — at least the speculation, in the absence of more specific data — is that a project will either take care of its documentation as part of development or not do it at all. We will learn to use a design style in which much of the documentation is actually embedded in the software, with special tools available to extract it.

The next items in Lientz and Swanson's list are also interesting, if less directly relevant to the topics of this book. Emergency bug fixes (done in haste when a user reports that the program is not producing the expected results or behaves in some catastrophic way) cost more than routine, scheduled corrections. This is not only because they must be performed under heavy pressure, but also because they disrupt the orderly process of delivering new releases, and may introduce new errors. The last two activities account for small percentages:

- One is efficiency improvements; this seems to suggest that once a system works, project managers and programmers are often reluctant to disrupt it in the hope of performance improvements, and prefer to leave good enough alone. (When considering the "first make it right, then make it fast" precept, many projects are probably happy enough to stop at the first of these steps.)
- Also accounting for a small percentage is "transfer to new environments". A possible interpretation (again a conjecture in the absence of more detailed data) is that there are two kinds of program with respect to portability, with little in-between: some programs are designed with portability in mind, and cost relatively little to port; others are so closely tied to their original platform, and would be so difficult to port, that developers do not even try.

1.4 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- The purpose of software engineering is to find ways of building quality software.
- Rather than a single factor, quality in software is best viewed as a tradeoff between a set of different goals.
- External factors, perceptible to users and clients, should be distinguished from internal factors, perceptible to designers and implementors.
- What matters is the external factors, but they can only be achieved through the internal factors.
- A list of basic external quality factors was presented. Those for which current software is most badly in need of better methods, and which the object-oriented method directly addresses, are the safety-related factors correctness and robustness, together known as reliability, and the factors requiring more decentralized software architectures: reusability and extendibility, together known as modularity.
- Software maintenance, which consumes a large portion of software costs, is penalized by the difficulty of implementing changes in software products, and by the over-dependence of programs on the physical structure of the data they manipulate.

1.5 BIBLIOGRAPHICAL NOTES

Several authors have proposed definitions of software quality. Among the first articles on subject, two in particular remain valuable today: [Hoare 1972], a guest editorial, and [Boehm 1978], the result of one of the first systematic studies, by a group at TRW.

The distinction between external and internal factors was introduced in a 1977 General Electric study commissioned by the US Air Force [McCall 1977]. McCall uses the terms "factors" and "criteria" for what this chapter has called external factors and internal factors. Many (although not all) of the factors introduced in this chapter correspond to some of McCall's; one of his factors, maintainability, was dropped, because, as explained, it is adequately covered by extendibility and verifiability. McCall's study discusses not only external factors but also a number of internal factors ("criteria"), as well as *metrics*, or quantitative techniques for assessing satisfaction of the internal factors. With object technology, however, many of that study's internal factors and metrics, too closely linked with older software practices, are obsolete. Carrying over this part of McCall's work to the techniques developed in this book would be a useful project; see the bibliography and exercises to chapter 3.

The argument about the relative effect of machine improvements depending on the complexity of the algorithms is derived from [Aho 1974].

On ease of use, a standard reference is [Shneiderman 1987], expanding on [Shneiderman 1980], which was devoted to the broader topic of software psychology. The Web page of Shneiderman's lab at *http://www.cs.umd.edu/projects/hcil/* contains many bibliographic references on these topics.

The Osmond curves come from a tutorial given by Roger Osmond at TOOLS USA [Osmond 1995]. Note that the form given in this chapter does not show time, enabling a more direct view of the tradeoff between functionality and other qualities in the two alternative curves, but not reflecting the black curve's potential for delaying a project. Osmond's original curves are plotted against time rather than functionality.

The chart of maintenance costs is derived from a study by Lientz and Swanson, based on a maintenance questionnaire sent to 487 organizations [Lientz 1980]. See also [Boehm 1979]. Although some of their input data may be considered too specialized and by now obsolete (the study was based on batch-type MIS applications of an average size of 23,000 instructions, large then but not by today's standards), the results generally seem still applicable. The Software Management Association performs a yearly survey of maintenance; see [Dekleva 1992] for a report about one of these surveys.

The expressions *programming-in-the-large* and *programming-in-the-small* were introduced by [DeRemer 1976].

For a general discussion of software engineering issues, see the textbook by Ghezzi, Jazayeri and Mandrioli [Ghezzi 1991]. A text on programming languages by some of the same authors, [Ghezzi 1997], provides complementary background for some of the issues discussed in the present book.

2

Criteria of object orientation

In the previous chapter we explored the goals of the object-oriented method. As a preparation for parts B and C, in which we will discover the technical details of the method, it is useful to take a quick but wide glance at the key aspects of object-oriented development. Such is the aim of this chapter.

One of the benefits will be to obtain a concise memento of what makes a system object-oriented. This expression has nowadays become so indiscriminately used that we need a list of precise properties under which we can assess any method, language or tool that its proponents claim to be O-O.

This chapter limits its explanations to a bare minimum, so if this is your first reading you cannot expect to understand in detail all the criteria listed; explaining them is the task of the rest of the book. Consider this discussion a preview — not the real movie, just a trailer.

Warning: *SPOILER*!

Actually a warning is in order because unlike any good trailer this chapter is also what film buffs call a *spoiler* — it gives away some of the plot early. As such it breaks the step-by-step progression of this book, especially part B, which patiently builds the case for object technology by looking at issue after issue before deducing and justifying the solutions. If you like the idea of reading a broad overview before getting into more depth, this chapter is for you. But if you prefer *not* to spoil the pleasure of seeing the problems unfold and of discovering the solutions one by one, then you should simply skip it. You will not need to have read it to understand subsequent chapters.

2.1 ON THE CRITERIA

Let us first examine the choice of criteria for assessing objectness.

How dogmatic do we need to be?

The list presented below includes all the facilities which I believe to be essential for the production of quality software using the object-oriented method. It is ambitious and may appear uncompromising or even dogmatic. What conclusion does this imply for an environment which satisfies some but not all of these conditions? Should one just reject such a half-hearted O-O environment as totally inadequate?

Only you, the reader, can answer this question relative to your own context. Several reasons suggest that some compromises may be necessary:

- "Object-oriented" is not a boolean condition: environment A, although not 100% O-O, may be "more" O-O than environment B; so if external constraints limit your choice to A and B you will have to pick A as the least bad object-oriented choice.
- Not everyone will need all of the properties all the time.
- Object orientation may be just one of the factors guiding your search for a software solution, so you may have to balance the criteria given here with other considerations.

All this does not change the obvious: to make informed choices, even if practical constraints impose less-than-perfect solutions, you need to know the complete picture, as provided by the list below.

Categories

The set of criteria which follows has been divided into three parts:

- *Method and language*: these two almost indistinguishable aspects cover the thought processes and the notations used to analyze and produce software. Be sure to note that (especially in object technology) the term "language" covers not just the programming language in a strict sense, but also the notations, textual or graphical, used for analysis and design.
- *Implementation and environment*: the criteria in this category describe the basic properties of the tools which allow developers to apply object-oriented ideas.
- *Libraries*: object technology relies on the reuse of software components. Criteria in this category cover both the availability of basic libraries and the mechanisms needed to use libraries and produce new ones.

This division is convenient but not absolute, as some criteria straddle two or three of the categories. For example the criterion labeled "memory management" has been classified under method and language because a language can support or prevent automatic garbage collection, but it also belongs to the implementation and environment category; the "assertion" criterion similarly includes a requirement for supporting tools.

2.2 METHOD AND LANGUAGE

The first set of criteria covers the method and the supporting notation.

Seamlessness

The object-oriented approach is ambitious: it encompasses the entire software lifecycle. When examining object-oriented solutions, you should check that the method and language, as well as the supporting tools, apply to analysis and design as well as implementation and maintenance. The language, in particular, should be a vehicle for thought which will help you through all stages of your work.

The result is a seamless development process, where the generality of the concepts and notations helps reduce the magnitude of the transitions between successive steps in the lifecycle.

These requirements exclude two cases, still frequently encountered but equally unsatisfactory:

- The use of object-oriented concepts for analysis and design only, with a method and notation that cannot be used to write executable software.
- The use of an object-oriented programming language which is not suitable for analysis and design.

In summary:

> An object-oriented language and environment, together with the supporting method, should apply to the entire lifecycle, in a way that minimizes the gaps between successive activities.

Classes

The object-oriented method is based on the notion of class. Informally, a class is a software element describing an abstract data type and its partial or total implementation. An abstract data type is a set of objects defined by the list of operations, or *features*, applicable to these objects, and the properties of these operations.

> The method and the language should have the notion of class as their central concept.

Assertions

The features of an abstract data type have formally specified properties, which should be reflected in the corresponding classes. Assertions — routine preconditions, routine postconditions and class invariants — play this role. They describe the effect of features on objects, independently of how the features have been implemented.

Assertions have three major applications: they help produce reliable software; they provide systematic documentation; and they are a central tool for testing and debugging object-oriented software.

> The language should make it possible to equip a class and its features with assertions (preconditions, postconditions and invariants), relying on tools to produce documentation out of these assertions and, optionally, monitor them at run time.

In the society of software modules, with classes serving as the cities and instructions (the actual executable code) serving as the executive branch of government, assertions provide the legislative branch. We shall see below who takes care of the judicial system.

Classes as modules

Object orientation is primarily an architectural technique: its major effect is on the modular structure of software systems.

The key role here is again played by classes. A class describes not just a type of objects but also a modular unit. In a pure object-oriented approach:

> Classes should be the only modules.

In particular, there is no notion of main program, and subprograms do not exist as independent modular units. (They may only appear as part of classes.) There is also no need for the "packages" of languages such as Ada, although we may find it convenient for management purposes to group classes into administrative units, called *clusters*.

Classes as types

The notion of class is powerful enough to avoid the need for any other typing mechanism:

> Every type should be based on a class.

Even basic types such as *INTEGER* and *REAL* can be derived from classes; normally such classes will be built-in rather than defined anew by each developer.

Feature-based computation

In object-oriented computation, there is only one basic computational mechanism: given a certain object, which (because of the previous rule) is always an instance of some class, call a feature of that class on that object. For example, to display a certain window on a screen, you call the feature *display* on an object representing the window — an instance of class *WINDOW*. Features may also have arguments: to increase the salary of an employee *e* by *n* dollars, effective at date *d*, you call the feature *raise* on *e*, with *n* and *d* as arguments.

Just as we treat basic types as predefined classes, we may view basic operations (such as addition of numbers) as special, predefined cases of feature call, a very general mechanism for describing computations:

> Feature call should be the primary computational mechanism.

A class which contains a call to a feature of a class *C* is said to be a **client** of *C*. Feature call is also known as **message passing**; in this terminology, a call such as the above will be described as passing to *e* the message "raise your pay", with arguments *d* and *n*.

Information hiding

When writing a class, you will sometimes have to include a feature which the class needs for internal purposes only: a feature that is part of the implementation of the class, but not of its interface. Others features of the class — possibly available to clients — may call the feature for their own needs; but it should not be possible for a client to call it directly.

The mechanism which makes certain features unfit for clients' calls is called information hiding. As explained in a later chapter, it is essential to the smooth evolution of software systems.

In practice, it is not enough for the information hiding mechanism to support exported features (available to all clients) and secret features (available to no client); class designers must also have the ability to export a feature selectively to a set of designated clients.

> It should be possible for the author of a class to specify that a feature is available to all clients, to no client, or to specified clients.

An immediate consequence of this rule is that communication between classes should be strictly limited. In particular, a good object-oriented language should not offer any notion of global variable; classes will exchange information exclusively through feature calls, and through the inheritance mechanism.

Exception handling

Abnormal events may occur during the execution of a software system. In object-oriented computation, they often correspond to calls that cannot be executed properly, as a result of a hardware malfunction, of an unexpected impossibility (such as numerical overflow in an addition), or of a bug in the software.

To produce reliable software, it is necessary to have the ability to recover from such situations. This is the purpose of an exception mechanism.

> The language should provide a mechanism to recover from unexpected abnormal situations.

In the society of software systems, as you may have guessed, the exception mechanism is the third branch of government, the judicial system (and the supporting police force).

Static typing

When the execution of a software system causes the call of a certain feature on a certain object, how do we know that this object will be able to handle the call? (In message terminology: how do we know that the object can process the message?)

To provide such a guarantee of correct execution, the language must be typed. This means that it enforces a few compatibility rules; in particular:

- Every entity (that is to say, every name used in the software text to refer to run-time objects) is explicitly declared as being of a certain type, derived from a class.
- Every feature call on a certain entity uses a feature from the corresponding class (and the feature is available, in the sense of information hiding, to the caller's class).
- Assignment and argument passing are subject to **conformance rules**, based on inheritance, which require the source's type to be compatible with the target's type.

In a language that imposes such a policy, it is possible to write a **static type checker** which will accept or reject software systems, guaranteeing that the systems it accepts will not cause any "feature not available on object" error at run time.

> A well-defined type system should, by enforcing a number of type declaration and compatibility rules, guarantee the run-time type safety of the systems it accepts.

Genericity

For typing to be practical, it must be possible to define type-parameterized classes, known as generic. A generic class *LIST* [*G*] will describe lists of elements of an arbitrary type represented by *G*, the "formal generic parameter"; you may then declare specific lists through such derivations as *LIST* [*INTEGER*] and *LIST* [*WINDOW*], using types *INTEGER* and *WINDOW* as "actual generic parameters". All derivations share the same class text.

> It should be possible to write classes with formal generic parameters representing arbitrary types.

This form of type parameterization is called **unconstrained** genericity. A companion facility mentioned below, constrained genericity, involves inheritance.

Single inheritance

Software development involves a large number of classes; many are variants of others. To control the resulting potential complexity, we need a classification mechanism, known as inheritance. A class will be an heir of another if it incorporates the other's features in addition to its own. (A *descendant* is a direct or indirect heir; the reverse notion is *ancestor*.)

> It should be possible to define a class as inheriting from another.

Inheritance is one of the central concepts of the object-oriented methods and has profound consequences on the software development process.

Multiple inheritance

We will often encounter the need to combine several abstractions. For example a class might model the notion of "infant", which we may view both as a "person", with the associated features, and, more prosaically, as a "tax-deductible item", which earns some deduction at tax time. Inheritance is justified in both cases. *Multiple* inheritance is the guarantee that a class may inherit not just from one other but from as many as is conceptually justified.

Multiple inheritance raises a few technical problems, in particular the resolution of *name clashes* (cases in which different features, inherited from different classes, have the same name). Any notation offering multiple inheritance must provide an adequate solution to these problems.

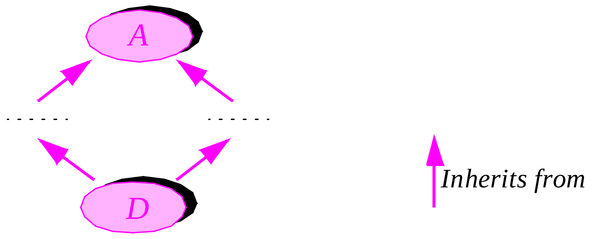
> It should be possible for a class to inherit from as many others as necessary, with an adequate mechanism for disambiguating name clashes.

The solution developed in this book is based on *renaming* the conflicting features in the heir class.

Repeated inheritance

Multiple inheritance raises the possibility of *repeated* inheritance, the case in which a class inherits from another through two or more paths, as shown.

Repeated inheritance



In such a case the language must provide precise rules defining what happens to features inherited repeatedly from the common ancestor, *A* in the figure. As the discussion of repeated inheritance will show, it may be desirable for a feature of *A* to yield just one feature of *D* in some cases (*sharing*), but in others it should yield two (*replication*). Developers must have the flexibility to prescribe either policy separately for each feature.

> Precise rules should govern the fate of features under repeated inheritance, allowing developers to choose, separately for each repeatedly inherited feature, between sharing and replication.

Constrained genericity

The combination of genericity and inheritance brings about an important technique, constrained genericity, through which you can specify a class with a generic parameter that represents not an arbitrary type as with the earlier (unconstrained) form of genericity, but a type that is a descendant of a given class.

A generic class *SORTABLE_LIST*, describing lists with a *sort* feature that will reorder them sequentially according to a certain order relation, needs a generic parameter representing the list elements' type. That type is not arbitrary: it must support an order relation. To state that any actual generic parameter must be a descendant of the library class *COMPARABLE*, describing objects equipped with an order relation, use constrained genericity to declare the class as *SORTABLE_LIST* [*G* –> *COMPARABLE*].

> The genericity mechanism should support the constrained form of genericity.

Redefinition

When a class is an heir of another, it may need to change the implementation or other properties of some of the inherited features. A class *SESSION* describing user sessions in an operating system may have a feature *terminate* to take care of cleanup operations at the end of a session; an heir might be *REMOTE_SESSION*, handling sessions started from a different computer on a network. If the termination of a remote session requires supplementary actions (such as notifying the remote computer), class *REMOTE_SESSION* will redefine feature *terminate*.

Redefinition may affect the implementation of a feature, its signature (type of arguments and result), and its specification.

> It should be possible to redefine the specification, signature and implementation of an inherited feature.

Polymorphism

With inheritance brought into the picture, the static typing requirement listed earlier would be too restrictive if it were taken to mean that every entity declared of type *C* may only refer to objects whose type is exactly *C*. This would mean for example that an entity of type *C* (in a navigation control system) could not be used to refer to an object of type *MERCHANT_SHIP* or *SPORTS_BOAT*, both assumed to be classes inheriting from *BOAT*.

As noted earlier, an "entity" is a name to which various values may become attached at run time. This is a generalization of the traditional notion of variable.

Polymorphism is the ability for an entity to become attached to objects of various possible types. In a statically typed environment, polymorphism will not be arbitrary, but controlled by inheritance; for example, we should not allow our *BOAT* entity to become attached to an object representing an object of type *BUOY*, a class which does not inherit from *BOAT*.

> It should be possible to attach entities (names in the software texts representing run-time objects) to run-time objects of various possible types, under the control of the inheritance-based type system.

Dynamic binding

The combination of the last two mechanisms mentioned, redefinition and polymorphism, immediately suggests the next one. Assume a call whose target is a polymorphic entity, for example a call to the feature *turn* on an entity declared of type *BOAT*. The various descendants of *BOAT* may have redefined the feature in various ways. Clearly, there must be an automatic mechanism to guarantee that the version of *turn* will always be the one deduced from the actual object's type, regardless of how the entity has been declared. This property is called dynamic binding.

> Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call.

Dynamic binding has a major influence on the structure of object-oriented applications, as it enables developers to write simple calls (meaning, for example, "call feature *turn* on entity *my_boat*") to denote what is actually several possible calls depending on the corresponding run-time situations. This avoids the need for many of the repeated tests ("Is this a merchant ship? Is this a sports boat?") which plague software written with more conventional approaches.

Run-time type interrogation

Object-oriented software developers soon develop a healthy hatred for any style of computation based on explicit choices between various types for an object. Polymorphism and dynamic binding provide a much preferable alternative. In some cases, however, an object comes from the outside, so that the software author has no way to predict its type with certainty. This occurs in particular if the object is retrieved from external storage, received from a network transmission or passed by some other system.

The software then needs a mechanism to access the object in a safe way, without violating the constraints of static typing. Such a mechanism should be designed with care, so as not to cancel the benefits of polymorphism and dynamic binding.

The **assignment attempt** operation described in this book satisfies these requirements. An assignment attempt is a conditional operation: it tries to attach an object to an entity; if in a given execution the object's type conforms to the type declared for the entity, the effect is that of a normal assignment; otherwise the entity gets a special "void" value. So you can handle objects whose type you do not know for sure, without violating the safety of the type system.

> It should be possible to determine at run time whether the type of an object conforms to a statically given type.

Deferred features and classes

In some cases for which dynamic binding provides an elegant solution, obviating the need for explicit tests, there is no initial version of a feature to be redefined. For example class *BOAT* may be too general to provide a default implementation of *turn*. Yet we want to be able to call feature *turn* to an entity declared of type *BOAT* if we have ensured that at run time it will actually be attached to objects of such fully defined types as *MERCHANT_SHIP* and *SPORTS_BOAT*.

In such cases *BOAT* may be declared as a deferred class (one which is not fully implemented), and with a deferred feature *turn*. Deferred features and classes may still possess assertions describing their abstract properties, but their implementation is postponed to descendant classes. A non-deferred class is said to be *effective*.

> It should be possible to write a class or a feature as deferred, that is to say specified but not fully implemented.

Deferred classes (also called abstract classes) are particularly important for object-oriented analysis and high-level design, as they make it possible to capture the essential aspects of a system while leaving details to a later stage.

Memory management and garbage collection

The last point on our list of method and language criteria may at first appear to belong more properly to the next category — implementation and environment. In fact it belongs to both. But the crucial requirements apply to the language; the rest is a matter of good engineering.

Object-oriented systems, even more than traditional programs (except in the Lisp world), tend to create many objects with sometimes complex interdependencies. A policy leaving developers in charge of managing the associated memory, especially when it comes to reclaiming the space occupied by objects that are no longer needed, would harm both the efficiency of the development process, as it would complicate the software and occupy a considerable part of the developers’ time, and the safety of the resulting systems, as it raises the risk of improper recycling of memory areas. In a good object-oriented environment memory management will be automatic, under the control of the *garbage collector*, a component of the runtime system.

The reason this is a language issue as much as an implementation requirement is that a language that has not been explicitly designed for automatic memory management will often render it impossible. This is the case with languages where a pointer to an object of a certain type may disguise itself (through conversions known as “casts”) as a pointer of another type or even as an integer, making it impossible to write a safe garbage collector.

> The language should make safe automatic memory management possible, and the implementation should provide an automatic memory manager taking care of garbage collection.

2.3 IMPLEMENTATION AND ENVIRONMENT

We come now to the essential features of a development environment supporting object-oriented software construction.

Automatic update

Software development is an incremental process. Developers do not commonly write thousands of lines at a time; they proceed by addition and modification, starting most of the time from a system that is already of substantial size.

When performing such an update, it is essential to have the guarantee that the resulting system will be consistent. For example, if you change a feature *f* of class *C*, you must be certain that every descendant of *C* which does not redefine *f* will be updated to have the new version of *f*, and that every call to *f* in a client of *C* or of a descendant of *C* will trigger the new version.

Conventional approaches to this problem are manual, forcing the developers to record all dependencies, and track their changes, using special mechanisms known as “make files” and “include files”. This is unacceptable in modern software development, especially in the object-oriented world where the dependencies between classes, resulting from the client and inheritance relations, are often complex but may be deduced from a systematic examination of the software text.

> System updating after a change should be automatic, the analysis of inter-class dependencies being performed by tools, not manually by developers.

It is possible to meet this requirement in a compiled environment (where the compiler will work together with a tool for dependency analysis), in an interpreted environment, or in one combining both of these language implementation techniques.

Fast update

In practice, the mechanism for updating the system after some changes should not only be automatic, it should also be fast. More precisely, it should be proportional to the size of the changed parts, not to the size of the system as a whole. Without this property, the method and environment may be applicable to small systems, but not to large ones.

> The time to process a set of changes to a system, enabling execution of the updated version, should be a function of the size of the changed components, independent of the size of the system as a whole.

Here too both interpreted and compiled environments may meet the criterion, although in the latter case the compiler must be incremental. Along with an incremental compiler, the environment may of course include a global optimizing compiler working on an entire system, as long as that compiler only needs to be used for delivering a final product; development will rely on the incremental compiler.

Persistence

Many applications, perhaps most, will need to conserve objects from one session to the next. The environment should provide a mechanism to do this in a simple way.

An object will often contain references to other objects; since the same may be true of these objects, this means that every object may have a large number of *dependent* objects, with a possibly complex dependency graph (which may involve cycles). It would usually make no sense to store or retrieve the object without all its direct and indirect dependents. A persistence mechanism which can automatically store an object's dependents along with the object is said to support **persistence closure**.

> A persistent storage mechanism supporting persistence closure should be available to store an object and all its dependents into external devices, and to retrieve them in the same or another session.

For some applications, mere persistence support is not sufficient; such applications will need full **database support**. The notion of object-oriented database is covered in a later chapter, which also explores other persistent issues such as *schema evolution*, the ability to retrieve objects safely even if the corresponding classes have changed.

Documentation

Developers of classes and systems must provide management, customers and other developers with clear, high-level descriptions of the software they produce. They need tools to assist them in this effort; as much as possible of the documentation should be produced automatically from the software texts. Assertions, as already noted, help make such software-extracted documents precise and informative.

> Automatic tools should be available to produce documentation about classes and systems.

Browsing

When looking at a class, you will often need to obtain information about other classes; in particular, the features used in a class may have been introduced not in the class itself but in its various ancestors. This puts on the environment the burden of providing developers with tools to examine a class text, find its dependencies on other classes, and switch rapidly from one class text to another.

S is a "supplier" of C if C is a client of S. "Client" was defined on page 24.

This task is called browsing. Typical facilities offered by good browsing tools include: find the clients, suppliers, descendants, ancestors of a class; find all the redefinitions of a feature; find the original declaration of a redefined feature.

> Interactive browsing facilities should enable software developers to follow up quickly and conveniently the dependencies between classes and features.

2.4 LIBRARIES

One of the characteristic aspects of developing software the object-oriented way is the ability to rely on libraries. An object-oriented environment should provide good libraries, and mechanisms to write more.

Basic libraries

The fundamental data structures of computing science — sets, lists, trees, stacks… — and the associated algorithms — sorting, searching, traversing, pattern matching — are ubiquitous in software development. In conventional approaches, each developer implements and re-implements them independently all the time; this is not only wasteful of efforts but detrimental to software quality, as it is unlikely that an individual developer who implements a data structure not as a goal in itself but merely as a component of some application will attain the optimum in reliability and efficiency.

An object-oriented development environment must provide reusable classes addressing these common needs of software systems.

> Reusable classes should be available to cover the most frequently needed data structures and algorithms.

Graphics and user interfaces

Many modern software systems are interactive, interacting with their users through graphics and other pleasant interface techniques. This is one of the areas where the object-oriented model has proved most impressive and helpful. Developers should be able to rely on graphical libraries to build interactive applications quickly and effectively.

> Reusable classes should be available for developing applications which provide their users with pleasant graphical user interface.

Library evolution mechanisms

Developing high-quality libraries is a long and arduous task. It is impossible to guarantee that the design of library will be perfect the first time around. An important problem, then, is to enable library developers to update and modify their designs without wreaking havoc in existing systems that depend on the library. This important criterion belongs to the library category, but also to the method and language category.

> Mechanisms should be available to facilitate library evolution with minimal disruption of client software.

Library indexing mechanisms

Another problem raised by libraries is the need for mechanisms to identify the classes addressing a certain need. This criterion affects all three categories: libraries, language (as there must be a way to enter indexing information within the text of each class) and tools (to process queries for classes satisfying certain conditions).

> Library classes should be equipped with indexing information allowing property-based retrieval.

2.5 FOR MORE SNEAK PREVIEW

Although to understand the concepts in depth it is preferable to read this book sequentially, readers who would like to complement the preceding theoretical overview with an advance glimpse of the method at work on a practical example can at this point read chapter 20, a case study of a practical design problem, on which it compares an O-O solution with one employing more traditional techniques.

That case study is mostly self-contained, so that you will understand the essentials without having read the intermediate chapters. (But if you do go ahead for this quick peek, you must promise to come back to the rest of the sequential presentation, starting with chapter 3, as soon as you are done.)

2.6 BIBLIOGRAPHICAL NOTES AND OBJECT RESOURCES

This introduction to the criteria of object orientation is a good opportunity to list a selection of books that offer quality introductions to object technology in general.

[Waldén 1995] discusses the most important issues of object technology, focusing on analysis and design, on which it is probably the best reference.

[Page-Jones 1995] provides an excellent overview of the method.

[Cox 1990] (whose first edition was published in 1986) is based on a somewhat different view of object technology and was instrumental in bringing O-O concepts to a much larger audience than before.

[Henderson-Sellers 1991] (a second edition is announced) provides a short overview of O-O ideas. Meant for people who are asked by their company to "go out and find out what that object stuff is about", it includes ready-to-be-photocopied transparency masters, precious on such occasions. Another overview is [Eliëns 1995].

The *Dictionary of Object Technology* [Firesmith 1995] provides a comprehensive reference on many aspects of the method.

All these books are to various degrees intended for technically-minded people. There is also a need to educate managers. [M 1995] grew out of a chapter originally planned for the present book, which became a full-fledged discussion of object technology for executives. It starts with a short technical presentation couched in business terms and continues with an analysis of management issues (lifecycle, project management, reuse policies). Another management-oriented book, [Goldberg 1995], provides a complementary perspective on many important topics. [Baudoin 1996] stresses lifecycle issues and the importance of standards.

Coming back to technical presentations, three influential books on object-oriented languages, written by the designers of these languages, contain general methodological discussions that make them of interest to readers who do not use the languages or might even be critical of them. (The history of programming languages and books about them shows that designers are not always the best to write about their own creations, but in these cases they were.) The books are:

- *Simula BEGIN* [Birtwistle 1973]. (Here two other authors joined the language designers Nygaard and Dahl.)
- *Smalltalk-80*: *The Language and its Implementation* [Goldberg 1983].
- *The C++ Programming Language*, *second edition* [Stroustrup 1991].

Chapter 29 discusses teaching the technology.

More recently, some introductory programming textbooks have started to use object-oriented ideas right from the start, as there is no reason to let "ontogeny repeat phylogeny", that is to say, take the poor students through the history of the hesitations and mistakes through which their predecessors arrived at the right ideas. The first such text (to my knowledge) was [Rist 1995]. Another good book covering similar needs is [Wiener 1996]. At the next level — textbooks for a second course on programming, discussing data structures and algorithms based on the notation of this book — you will find [Gore 1996] and [Wiener 1997]; [Jézéquel 1996] presents the principles of object-oriented software engineering.

The Usenet newsgroup *comp.object*, archived on several sites around the Web, is the natural medium of discussion for many issues of object technology. As with all such forums, be prepared for a mixture of the good, the bad and the ugly. The Object Technology department of *Computer* (IEEE), which I have edited since it started in 1995, has frequent invited columns by leading experts.

Magazines devoted to Object Technology include:

- The *Journal of Object-Oriented Programming* (the first journal in the field, emphasizing technical discussions but for a large audience), *Object Magazine* (of a more general scope, with some articles for managers), *Objekt Spektrum* (German), *Object Currents* (on-line), all described at *http://www.sigs.com*.
- *Theory and Practice of Object Systems*, an archival journal.
- *L'OBJET* (French), described at *http://www.tools.com/lobjet*.

The major international O-O conferences are OOPSLA (yearly, USA or Canada, see *http://www.acm.org*); *Object Expo* (variable frequency and locations, described at *http://www.sigs.com*); and TOOLS (Technology of Object-Oriented Languages and Systems), organized by ISE with three sessions a year (USA, Europe, Pacific), whose home page at *http://www.tools.com* also serves as a general resource on object technology and the topics of this book.

3 Modularity

From the goals of extendibility and reusability, two of the principal quality factors introduced in chapter 1, follows the need for flexible system architectures, made of autonomous software components. This is why chapter 1 also introduced the term *modularity* to cover the combination of these two quality factors.

Modular programming was once taken to mean the construction of programs as assemblies of small pieces, usually subroutines. But such a technique cannot bring real extendibility and reusability benefits unless we have a better way of guaranteeing that the resulting pieces — the **modules** — are self-contained and organized in stable architectures. Any comprehensive definition of modularity must ensure these properties.

A software construction method is modular, then, if it helps designers produce software systems made of autonomous elements connected by a coherent, simple structure. The purpose of this chapter is to refine this informal definition by exploring what precise properties such a method must possess to deserve the "modular" label. The focus will be on design methods, but the ideas also apply to earlier stages of system construction (analysis, specification) and must of course be maintained at the implementation and maintenance stages.

As it turns out, a single definition of modularity would be insufficient; as with software quality, we must look at modularity from more than one viewpoint. This chapter introduces a set of complementary properties: five *criteria*, five *rules* and five *principles* of modularity which, taken collectively, cover the most important requirements on a modular design method.

For the practicing software developer, the principles and the rules are just as important as the criteria. The difference is simply one of causality: the criteria are mutually independent — and it is indeed possible for a method to satisfy one of them while violating some of the others — whereas the rules follow from the criteria and the principles follow from the rules.

You might expect this chapter to begin with a precise description of what a module looks like. This is not the case, and for a good reason: our goal for the exploration of modularity issues, in this chapter and the next two, is precisely to analyze the properties which a satisfactory module structure must satisfy; so the form of modules will be a conclusion of the discussion, not a premise. Until we reach that conclusion the word "module" will denote the basic unit of decomposition of our systems, whatever it actually is. If you are familiar with non-object-oriented methods you will probably think of the subroutines present in most programming and design languages, or perhaps of packages as present in Ada and (under a different name) in Modula. The discussion will lead in a later chapter to the O-O form of module — the class — which supersedes these ideas. If you have encountered classes and O-O techniques before, you should still read this chapter to understand the requirements that classes address, a prerequisite if you want to use them well.

3.1 FIVE CRITERIA

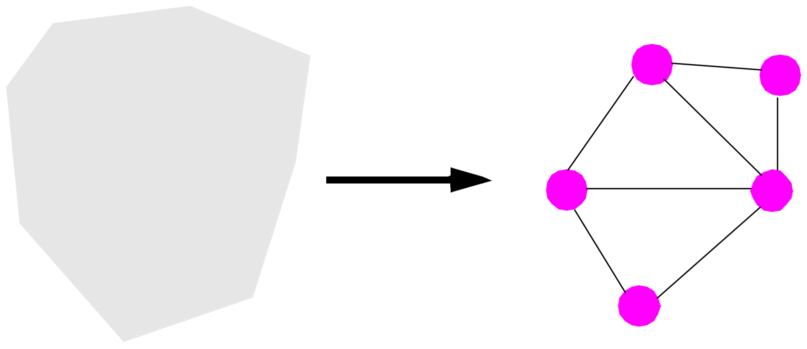
A design method worthy of being called "modular" should satisfy five fundamental requirements, explored in the next few sections:

- Decomposability.
- Composability.
- Understandability.
- Continuity.
- Protection.

Modular decomposability

A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them

The process will often be self-repeating since each subproblem may still be complex enough to require further decomposition.


Decomposability

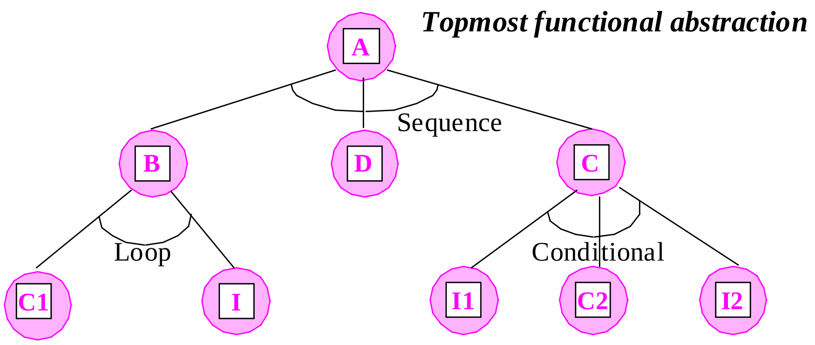
A corollary of the decomposability requirement is *division of labor*: once you have decomposed a system into subsystems you should be able to distribute work on these subsystems among different people or groups. This is a difficult goal since it limits the dependencies that may exist between the subsystems:

- You must keep such dependencies to the bare minimum; otherwise the development of each subsystem would be limited by the pace of the work on the other subsystems.
- The dependencies must be known: if you fail to list all the relations between subsystems, you may at the end of the project get a set of software elements that appear to work individually but cannot be put together to produce a complete system satisfying the overall requirements of the original problem.

As discussed below, top-down design is not as well suited to other modularity criteria.

The most obvious *example* of a method meant to satisfy the decomposability criterion is **top-down design**. This method directs designers to start with a most abstract description of the system's function, and then to refine this view through successive steps, decomposing each subsystem at each step into a small number of simpler subsystems, until all the remaining elements are of a sufficiently low level of abstraction to allow direct implementation. The process may be modeled as a tree.

A top-down hierarchy



The term "temporal cohesion" comes from the method known as structured design; see the bibliographical notes.

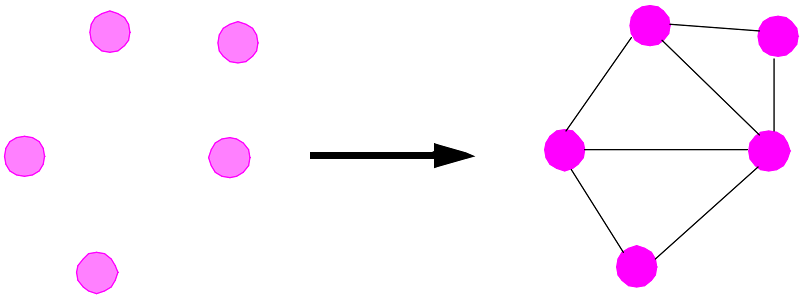
A typical *counter-example* is any method encouraging you to include, in each software system that you produce, a global initialization module. Many modules in a system will need some kind of initialization — actions such as the opening of certain files or the initialization of certain variables, which the module must execute before it performs its first directly useful tasks. It may seem a good idea to concentrate all such actions, for all modules of the system, in a module that initializes everything for everybody. Such a module will exhibit good "temporal cohesion" in that all its actions are executed at the same stage of the system's execution. But to obtain this temporal cohesion the method would endanger the autonomy of modules: you will have to grant the initialization module authorization to access many separate data structures, belonging to the various modules of the system and requiring specific initialization actions. This means that the author of the initialization module will constantly have to peek into the internal data structures of the other modules, and interact with their authors. This is incompatible with the decomposability criterion.

In the object-oriented method, every module will be responsible for the initialization of its own data structures.

Modular composability

A method satisfies Modular Composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.

Where decomposability was concerned with the derivation of subsystems from overall systems, composability addresses the reverse process: extracting existing software elements from the context for which they were originally designed, so as to use them again in different contexts.



Composability

A modular design method should facilitate this process by yielding software elements that will be sufficiently autonomous — sufficiently independent from the immediate goal that led to their existence — as to make the extraction possible.

Composability is directly connected with the goal of reusability: the aim is to find ways to design software elements performing well-defined tasks and usable in widely different contexts. This criterion reflects an old dream: transforming the software design process into a construction box activity, so that we would build programs by combining standard prefabricated elements.

- *Example 1*: *subprogram libraries*. Subprogram libraries are designed as sets of composable elements. One of the areas where they have been successful is numerical computation, which commonly relies on carefully designed subroutine libraries to solve problems of linear algebra, finite elements, differential equations etc.

- *Example 2*: *Unix Shell conventions*. Basic Unix commands operate on an input viewed as a sequential character stream, and produce an output with the same standard structure. This makes them potentially composable through the | operator of the command language (“shell”): *A* | *B* represents a program which will take *A*’s input, have *A* process it, send the output to *B* as input, and have it processed by *B*. This systematic convention favors the composability of software tools.

- *Counter-example*: *preprocessors*. A popular way to extend the facilities of programming languages, and sometimes to correct some of their deficiencies, is to use "preprocessors" that accept an extended syntax as input and map it into the standard form of the language. Typical preprocessors for Fortran and C support graphical primitives, extended control structures or database operations. Usually, however, such extensions are not compatible; then you cannot combine two of the preprocessors, leading to such dilemmas as whether to use graphics or databases.

The figure illustrating top-down design was on page 41.

Composability is independent of decomposability. In fact, these criteria are often at odds. Top-down design, for example, which we saw as a technique favoring decomposability, tends to produce modules that are *not* easy to combine with modules coming from other sources. This is because the method suggests developing each module to fulfill a specific requirement, corresponding to a subproblem obtained at some point in the refinement process. Such modules tend to be closely linked to the immediate context that led to their development, and unfit for adaptation to other contexts. The method provides neither hints towards making modules more general than immediately required, nor any incentives to do so; it helps neither avoid nor even just detect commonalities or redundancies between modules obtained in different parts of the hierarchy.

That composability and decomposability are both part of the requirements for a modular method reflects the inevitable mix of top-down and bottom-up reasoning — a complementarity that René Descartes had already noted almost four centuries ago, as shown by the contrasting two paragraphs of the *Discourse* extract at the beginning of part B.

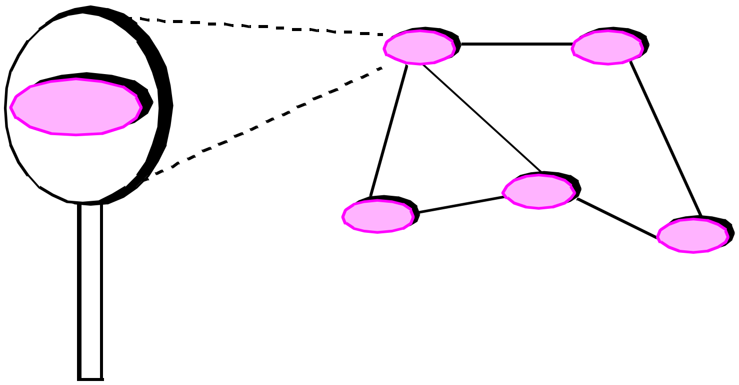
Modular understandability

A method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.

See "ABOUT SOFTWARE MAINTENANCE", 1.3, page 17.

The importance of this criterion follows from its influence on the maintenance process. Most maintenance activities, whether of the noble or not-so-noble category, involve having to dig into existing software elements. A method can hardly be called modular if a reader of the software is unable to understand its elements separately.

Understandability



This criterion, like the others, applies to the modules of a system description at any level: analysis, design, implementation.

- *Counter-example*: *sequential dependencies*. Assume some modules have been so designed that they will only function correctly if activated in a certain prescribed order; for example, *B* can only work properly if you execute it after *A* and before *C*, perhaps because they are meant for use in "piped" form as in the Unix notation encountered earlier:

  A | *B* | *C*

  Then it is probably hard to understand *B* without understanding *A* and *C* too.

In later chapters, the modular understandability criterion will help us address two important questions: how to document reusable components; and how to index reusable components so that software developers can retrieve them conveniently through queries. The criterion suggests that information about a component, useful for documentation or for retrieval, should whenever possible appear in the text of the component itself; tools for documentation, indexing or retrieval can then process the component to extract the needed pieces of information. Having the information included *in* each component is preferable to storing it elsewhere, for example in a database of information *about* components.

See also, later in this chapter, "Self-Documentation", page 54.

Modular continuity

A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.

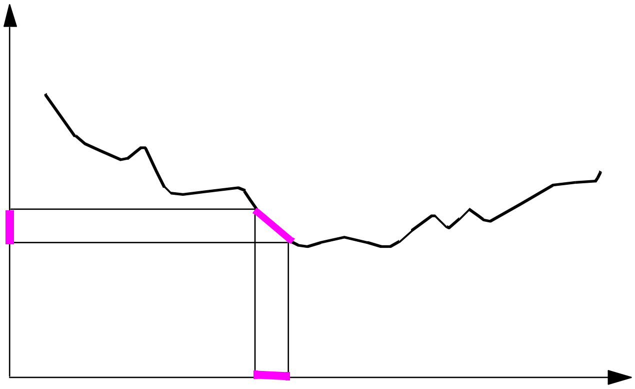
This criterion is directly connected to the general goal of extendibility. As emphasized in an earlier chapter, change is an integral part of the software construction process. The requirements will almost inevitably change as the project progresses. Continuity means that small changes should affect individual modules in the structure of the system, rather than the structure itself.

See "Extendibility", page 6.

The term "continuity" is drawn from an analogy with the notion of a continuous function in mathematical analysis. A mathematical function is continuous if (informally) a small change in the argument will yield a proportionally small change in the result. Here the function considered is the software construction method, which you can view as a mechanism for obtaining systems from specifications:

$software\_construction\_method: Specification \rightarrow System$

Continuity



This mathematical term only provides an analogy, since we lack formal notions of size for software. More precisely, it would be possible to define a generally acceptable measure of what constitutes a "small" or "large" change to a program; but doing the same for the specifications is more of a challenge. If we make no pretense of full rigor, however, the concepts should be intuitively clear and correspond to an essential requirement on any modular method.

This will be one of our principles of style: Symbolic Constant Principle, page 884.

- *Example 1: symbolic constants*. A sound style rule bars the instructions of a program from using any numerical or textual constant directly; instead, they rely on symbolic names, and the actual values only appear in a constant definition (**constant** in Pascal or Ada, preprocessor macros in C, *PARAMETER* in Fortran 77, constant attributes in the notation of this book). If the value changes, the only thing to update is the constant definition. This small but important rule is a wise precaution for continuity since constants, in spite of their name, are remarkably prone to change.

See "Uniform Access", page 55.

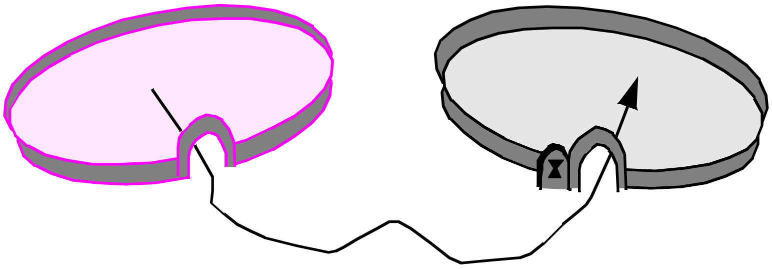
- *Example 2: the Uniform Access principle*. Another rule states that a single notation should be available to obtain the features of an object, whether they are represented as data fields or computed on demand. This property is sufficiently important to warrant a separate discussion later in this chapter.
- *Counter-example 1: using physical representations*. A method in which program designs are patterned after the physical implementation of data will yield designs that are very sensitive to slight changes in the environment.
- *Counter-example 2: static arrays*. Languages such as Fortran or standard Pascal, which do not allow the declaration of arrays whose bounds will only be known at run time, make program evolution much harder.

Modular protection

A method satisfies Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

The underlying issue, that of failures and errors, is central to software engineering. The errors considered here are run-time errors, resulting from hardware failures, erroneous input or exhaustion of needed resources (for example memory storage). The criterion does not address the avoidance or correction of errors, but the aspect that is directly relevant to modularity: their propagation.

The question of how to handle abnormal cases is discussed in detail in chapter 12.



Protection violation

- *Example*: *validating input at the source*. A method requiring that you make every module that inputs data also responsible for checking their validity is good for modular protection.

More on this topic in "Assertions are not an input checking mechanism", page 346

- *Counter-example*: *undisciplined exceptions*. Languages such as PL/I, CLU, Ada, C++ and Java support the notion of exception. An exception is a special signal that may be "raised" by a certain instruction and "handled" in another, possibly remote part of the system. When the exception is raised, control is transferred to the handler. (Details of the mechanism vary between languages; Ada or CLU are more disciplined in this respect than PL/I.) Such facilities make it possible to decouple the algorithms for normal cases from the processing of erroneous cases. But they must be used carefully to avoid hindering modular protection. The chapter on exceptions will investigate how to design a disciplined exception mechanism satisfying the criterion.

On exception handling, see chapter 12.

3.2 FIVE RULES

From the preceding criteria, five rules follow which we must observe to ensure modularity:

- Direct Mapping.
- Few Interfaces.
- Small interfaces (weak coupling).
- Explicit Interfaces.
- Information Hiding.

The first rule addresses the connection between a software system and the external systems with which it is connected; the next four all address a common issue — how modules will communicate. Obtaining good modular architectures requires that communication occur in a controlled and disciplined way.

Direct Mapping

Any software system attempts to address the needs of some problem domain. If you have a good model for describing that domain, you will find it desirable to keep a clear correspondence (mapping) between the structure of the solution, as provided by the software, and the structure of the problem, as described by the model. Hence the first rule:

> The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain.

This advice follows in particular from two of the modularity criteria:

- Continuity: keeping a trace of the problem's modular structure in the solution's structure will make it easier to assess and limit the impact of changes.
- Decomposability: if some work has already been done to analyze the modular structure of the problem domain, it may provide a good starting point for the modular decomposition of the software.

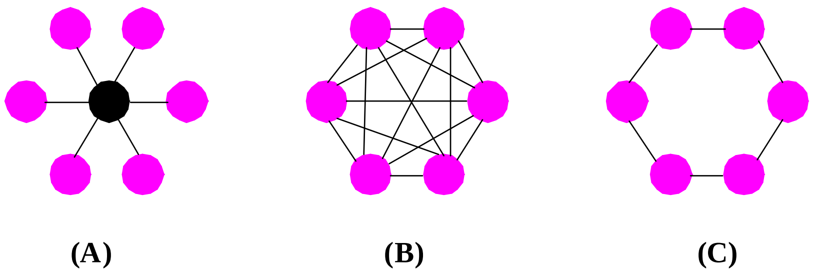
Few Interfaces

The Few Interfaces rule restricts the overall number of communication channels between modules in a software architecture:

> Every module should communicate with as few others as possible.

Communication may occur between modules in a variety of ways. Modules may call each other (if they are procedures), share data structures etc. The Few Interfaces rule limits the number of such connections.

Types of module interconnection structures



More precisely, if a system is composed of n modules, then the number of intermodule connections should remain much closer to the minimum, $n-1$, shown as **(A)** in the figure, than to the maximum, $n\ (n-1)\ /2$, shown as **(B)**.

This rule follows in particular from the criteria of continuity and protection: if there are too many relations between modules, then the effect of a change or of an error may propagate to a large number of modules. It is also connected to composability (if you want a module to be usable by itself in a new environment, then it should not depend on too many others), understandability and decomposability.

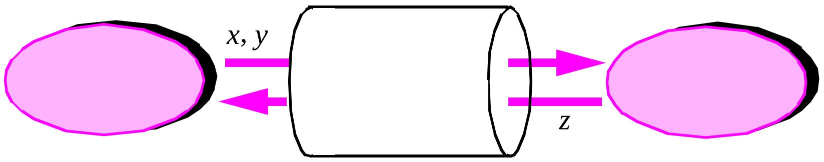
Case **(A)** on the last figure shows a way to reach the minimum number of links, *n – 1*, through an extremely centralized structure: one master module; everybody else talks to it and to it only. But there are also much more "egalitarian" structures, such as **(C)** which has almost the same number of links. In this scheme, every module just talks to its two immediate neighbors, but there is no central authority. Such a style of design is a little surprising at first since it does not conform to the traditional model of functional, top-down design. But it can yield robust, extendible architectures; this is the kind of structure that object-oriented techniques, properly applied, will tend to yield.

Small Interfaces

The Small Interfaces or "Weak Coupling" rule relates to the size of intermodule connections rather than to their number:

> If two modules communicate, they should exchange as little information as possible

An electrical engineer would say that the channels of communication between modules must be of limited bandwidth:



Communication bandwidth between modules

The Small Interfaces requirement follows in particular from the criteria of continuity and protection.

An extreme *counter-example* is a Fortran practice which some readers will recognize: the "garbage common block". A common block in Fortran is a directive of the form

COMMON /common_name/ variable$_1$*,… variable*$_n$

indicating that the variables listed are accessible not just to the enclosing module but also to any other module which includes a *COMMON* directive with the same *common_name*. It is not infrequent to see Fortran systems whose every module includes an identical gigantic *COMMON* directive, listing all significant variables and arrays so that every module may directly use every piece of data.

The problem, of course, is that every module may also misuse the common data, and hence that modules are tightly coupled to each other; the problems of modular continuity (propagation of changes) and protection (propagation of errors) are particularly nasty. This time-honored technique has nevertheless remained a favorite, no doubt accounting for many a late-night debugging session.

Developers using languages with nested structures can suffer from similar troubles. With block structure as introduced by Algol and retained in a more restricted form by Pascal, it is possible to include blocks, delimited by **begin** … **end** pairs, within other blocks. In addition every block may introduce its own variables, which are only meaningful within the syntactic scope of the block. For example:

The Body of a block is a sequence of instructions. The syntax used here is compatible with the notation used in subsequent chapters, so it is not exactly Algol's. "--" introduces a comment.

```
local-- Beginning of block B1
        x, y: INTEGER
do
        … Instructions of B1 …

        local -- Beginning of block B2
                z: BOOLEAN
        do
                … Instructions of B2 …
        end --- of block B2

        local -- Beginning of block B3
                y, z: INTEGER
        do
                … Instructions of B3 …
        end -- of block B3

        … Instructions of B1 (continued) …
end -- of block B1
```

Variable *x* is accessible to all instructions throughout this extract, whereas the two variables called *z* (one *BOOLEAN*, the other *INTEGER*) have scopes limited to B2 and B3 respectively. Like *x*, variable *y* is declared at the level of B1, but its scope does not include B3, where another variable of the same name (and also of type *INTEGER*) locally takes precedence over the outermost *y*. In Pascal this form of block structure exists only for blocks associated with routines (procedures and functions).

With block structure, the equivalent of the Fortran garbage common block is the practice of declaring all variables at the topmost level. (The equivalent in C-based languages is to introduce all variables as external.)

On clusters see chapter 28. The O-O alternative to nesting is studied in "The architectural role of selective exports", page 209.

Block structure, although an ingenious idea, introduces many opportunities to violate the Small Interfaces rule. For that reason we will refrain from using it in the object-oriented notation devised later in this book, especially since the experience of Simula, an object-oriented Algol derivative supporting block structure, shows that the ability to nest classes is redundant with some of the facilities provided by inheritance. The architecture of object-oriented software will involve three levels: a system is a set of clusters; a cluster is a set of classes; a class is a set of features (attributes and routines). Clusters, an organizational tool rather than a linguistic construct, can be nested to allow a project leader to structure a large system in as many levels as necessary; but classes as well as features have a flat structure, since nesting at either of those levels would cause unnecessary complication.

Explicit Interfaces

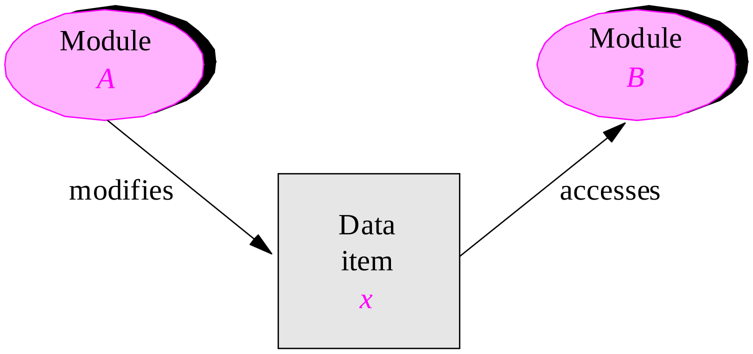
With the fourth rule, we go one step further in enforcing a totalitarian regime upon the society of modules: not only do we demand that any conversation be limited to few participants and consist of just a few words; we also require that such conversations must be held in public and loudly!

> Whenever two modules *A* and *B* communicate, this must be obvious from the text of *A* or *B* or both.

Behind this rule stand the criteria of decomposability and composability (if you need to decompose a module into several submodules or compose it with other modules, any outside connection should be clearly visible), continuity (it should be easy to find out what elements a potential change may affect) and understandability (how can you understand *A* by itself if *B* can influence its behavior in some devious way?).

One of the problems in applying the Explicit Interfaces rule is that there is more to intermodule coupling than procedure call; data sharing, in particular, is a source of indirect coupling:

Data sharing



Assume that module *A* modifies and module *B* uses the same data item *x*. Then *A* and *B* are in fact strongly coupled through *x* even though there may be no apparent connection, such as a procedure call, between them.

Information Hiding

The rule of Information Hiding may be stated as follows:

> The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

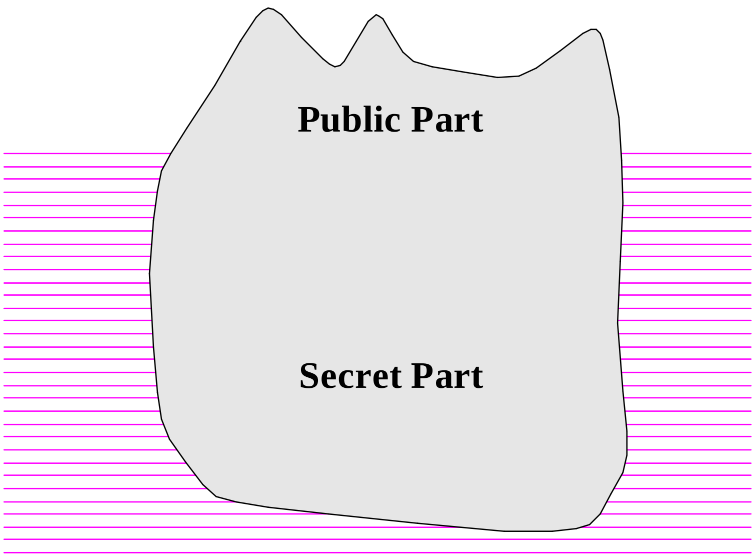
Application of this rule assumes that every module is known to the rest of the world (that is to say, to designers of other modules) through some official description, or **public** properties.

Of course, the whole text of the module itself (program text, design text) could serve as the description: it provides a correct view of the module since it *is* the module! The Information Hiding rule states that this should not in general be the case: the description should only include *some* of the module's properties. The rest should remain non-public, or **secret**. Instead of public and secret properties, one may also talk of exported and private properties. The public properties of a module are also known as the **interface** of the module (not to be confused with the user interface of a software system).

The fundamental reason behind the rule of Information Hiding is the continuity criterion. Assume a module changes, but the changes apply only to its secret elements, leaving the public ones untouched; then other modules who use it, called its *clients*, will not be affected. The smaller the public part, the higher the chances that changes to the module will indeed be in the secret part.

We may picture a module supporting Information Hiding as an iceberg; only the tip — the interface — is visible to the clients.

A module under Information Hiding



As a typical example, consider a procedure for retrieving the attributes associated with a key in a certain table, such as a personnel file or the symbol table of a compiler. The procedure will internally be very different depending on how the table is stored (sequential array or file, hash table, binary or B-Tree etc.). Information hiding implies that uses of this procedure should be independent of the particular implementation chosen. That way client modules will not suffer from any change in implementation.

Information hiding emphasizes separation of function from implementation. Besides continuity, this rule is also related to the criteria of decomposability, composability and understandability. You cannot develop the modules of a system separately, combine various existing modules, or understand individual modules, unless you know precisely what each of them may and may not expect from the others.

Which properties of a module should be public, and which ones secret? As a general guideline, the public part should include the specification of the module's functionality; anything that relates to the implementation of that functionality should be kept secret, so as to preserve other modules from later reversals of implementation decisions.

This first answer is still fairly vague, however, as it does not tell us what is the specification and what is the implementation; in fact, one might be tempted to reverse the definition by stating that the specification consists of whatever public properties the module has, and the implementation of its secrets! The object-oriented approach will give us a much more precise guideline thanks to the theory of abstract data types.

See chapter 6, in particular "Abstract data types and information hiding", page 144.

To understand information hiding and apply the rule properly, it is important to avoid a common misunderstanding. In spite of its name, information hiding does not imply *protection* in the sense of security restrictions — physically prohibiting authors of client modules from accessing the internal text of a supplier module. Client authors may well be permitted to read all the details they want: preventing them from doing so may be reasonable in some circumstances, but it is a project management decision which does not necessarily follow from the information hiding rule. As a technical requirement, information hiding means that client modules (whether or not their authors are permitted to read the secret properties of suppliers) should only rely on the suppliers' public properties. More precisely, it should be impossible to write client modules whose correct functioning depends on secret information.

In a completely formal approach to software construction, this definition would be stated as follows. To prove the correctness of a module, you will need to assume some properties about its suppliers. Information hiding means that such proofs are only permitted to rely on public properties of the suppliers, never on their secret properties.

See the comments on conditional correctness on page 4.

Consider again the example of a module providing a table searching mechanism. Some client module, which might belong to a spreadsheet program, uses a table, and relies on the table module to look for a certain element in the table. Assume further that the algorithm uses a binary search tree implementation, but that this property is secret — not part of the interface. Then you may or may not allow the author of the table searching module to tell the author of the spreadsheet program what implementation he has used for tables. This is a project management decision, or perhaps (for commercially released software) a marketing decision; in either case it is irrelevant to the question of information hiding. Information hiding means something else: that *even if the author of the spreadsheet program knows* that the implementation uses a binary search tree, he should be unable to write a client module which will only function correctly with this implementation — and would not work any more if the table implementation was changed to something else, such as hash coding.

One of the reasons for the misunderstanding mentioned above is the very term "information hiding", which tends to suggest physical protection. "Encapsulation", sometimes used as a synonym for information hiding, is probably preferable in this respect, although this discussion will retain the more common term.

By default "Ada" always means the most widespread form of the language (83), not the more recent Ada 95. Chapter 33 presents both versions.

As a summary of this discussion: the key to information hiding is not management or marketing policies as to who may or may not access the source text of a module, but strict **language rules** to define what access rights a module has on properties of its suppliers. As explained in the next chapter, "encapsulation languages" such as Ada and Modula-2 made the first steps in the right direction. Object technology will bring a more complete solution.

3.3 FIVE PRINCIPLES

From the preceding rules, and indirectly from the criteria, five principles of software construction follow:

- The Linguistic Modular Units principle.
- The Self-Documentation principle.
- The Uniform Access principle.
- The Open-Closed principle.
- The Single Choice principle.

Linguistic Modular Units

The Linguistic Modular Units principle expresses that the formalism used to describe software at various levels (specifications, designs, implementations) must support the view of modularity retained:

Linguistic Modular Units principle

Modules must correspond to syntactic units in the language used.

The language mentioned may be a programming language, a design language, a specification language etc. In the case of programming languages, modules should be separately compilable.

What this principle excludes at any level — analysis, design, implementation — is combining a method that suggests a certain module concept and a language that does not offer the corresponding modular construct, forcing software developers to perform manual translation or restructuring. It is indeed not uncommon to see companies hoping to apply certain methodological concepts (such as modules in the Ada sense, or object-oriented principles) but then implement the result in a programming language such as Pascal or C which does not support them. Such an approach defeats several of the modularity criteria:

- Continuity: if module boundaries in the final text do not correspond to the logical decomposition of the specification or design, it will be difficult or impossible to maintain consistency between the various levels when the system evolves. A change of the specification may be considered small if it affects only a small number of specification modules; to ensure continuity, there must be a direct correspondence between specification, design and implementation modules.
- Direct Mapping: to maintain a clear correspondence between the structure of the model and the structure of the solution, you must have a clear syntactical identification of the conceptual units on both sides, reflecting the division suggested by your development method.
- Decomposability: to divide system development into separate tasks, you need to make sure that every task results in a well-delimited syntactic unit; at the implementation stage, these units must be separately compilable.
- Composability: how could we combine anything other than modules with unambiguous syntactic boundaries?
- Protection: you can only hope to control the scope of errors if modules are syntactically delimited.

Self-Documentation

Like the rule of Information Hiding, the Self-Documentation principle governs how we should document modules:

Self-Documentation principle

The designer of a module should strive to make all information about the module part of the module itself.

What this precludes is the common situation in which information about the module is kept in separate project documents.

The documentation under review here is **internal** documentation about components of the software, not **user** documentation about the resulting product, which may require separate products, whether paper, CD-ROM or Web pages — although, as noted in the discussion of software quality, one may see in the modern trend towards providing more and more on-line help a consequence of the same general idea.

"About documentation", page 14.

The most obvious justification for the Self-Documentation principle is the criterion of modular understandability. Perhaps more important, however, is the role of this principle in helping to meet the continuity criterion. If the software and its documentation are treated as separate entities, it is difficult to guarantee that they will remain compatible — "in sync" — when things start changing. Keeping everything at the same place, although not a guarantee, is a good way to help maintain this compatibility.

Innocuous as this principle may seem at first, it goes against much of what the software engineering literature has usually suggested as good software development practices. The dominant view is that software developers, to deserve the title of software engineers, need to do what other engineers are supposed to: produce a kilogram of paper for every gram of actual deliverable. The encouragement to keep a record of the software construction process is good advice — but not the implication that software and its documentation are different products.

Such an approach ignores the specific property of software, which again and again comes back in this discussion: its changeability. If you treat the two products as separate, you risk finding yourself quickly in a situation where the documentation says one thing and the software does something else. If there is any worse situation than having no documentation, it must be having wrong documentation.

A major advance of the past few years has been the appearance of *quality standards* for software, such as ISO certification, the "2167" standard and its successors from the US Department of Defense, and the Capability Maturity Model of the Software Engineering Institute. Perhaps because they often sprang out of models from other disciplines, they tend to specify a heavy paper trail. Several of these standards could have a stronger effect on software quality (beyond providing a mechanism for managers to cover their bases in case of later trouble) by enforcing the Self-Documentation principle.

"Using assertions for documentation: the short form of a class", page 390. See also chapter 23 and its last two exercises.

This book will draw on the Self-Documentation principle to define a method for documenting classes — the modules of object-oriented software construction — that includes the documentation of every module in the module itself. Not that the module *is* its documentation: there is usually too much detail in the software text to make it suitable as documentation (this was the argument for information hiding). Instead, the module should *contain* its documentation.

In this approach software becomes a single product that supports multiple **views**. One view, suitable for compilation and execution, is the full source code. Another is the abstract interface documentation of each module, enabling software developers to write client modules without having to learn the module's own internals, in accordance with the rule of Information Hiding. Other views are possible.

We will need to remember this rule when we examine the question of how to document the classes of object-oriented software construction.

Uniform Access

Also known as the Uniform Reference principle.

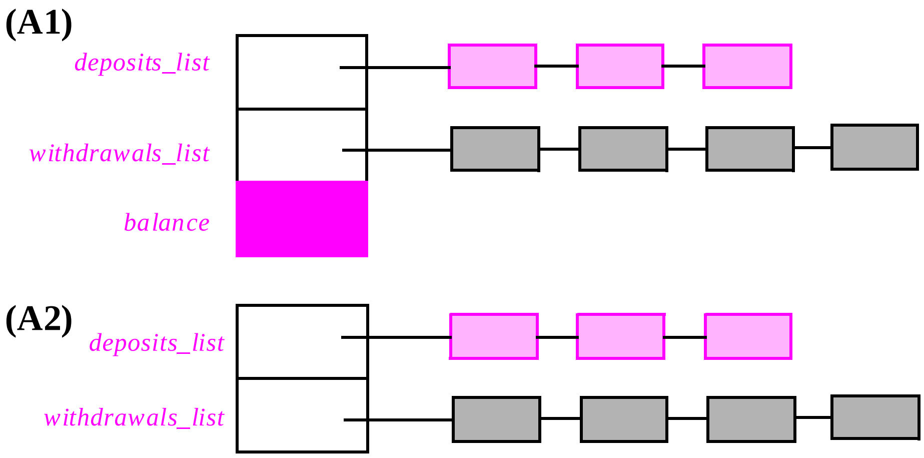
Although it may at first appear just to address a notational issue, the Uniform Access principle is in fact a design rule which influences many aspects of object-oriented design and the supporting notation. It follows from the Continuity criterion; you may also view it as a special case of Information Hiding.

Let *x* be a name used to access a certain data item (what will later be called an object) and *f* the name of a feature applicable to *x*. (A feature is an operation; this terminology will also be defined more precisely.) For example, *x* might be a variable representing a bank account, and *f* the feature that yields an account's current balance. Uniform Access addresses the question of how to express the result of applying *f* to *x*, using a notation that does not make any premature commitment as to how *f* is implemented.

In most design and programming languages, the expression denoting the application of *f* to *x* depends on what implementation the original software developer has chosen for feature *f*: is the value stored along with *x*, or must it be computed whenever requested? Both techniques are possible in the example of accounts and their balances:

A1 • You may represent the balance as one of the fields of the record describing each account, as shown in the figure. With this technique, every operation that changes the balance must take care of updating the *balance* field.

A2 • Or you may define a function which computes the balance using other fields of the record, for example fields representing the lists of withdrawals and deposits. With this technique the balance of an account is not stored (there is no *balance* field) but computed on demand.



Two representation for a bank account

A common notation, in languages such as Pascal, Ada, C, C++ and Java, uses *x*.*f* in case A1 and *f* (*x*) in case A2.

Choosing between representations A1 and A2 is a space-time tradeoff: one economizes on computation, the other on storage. The resolution of this tradeoff in favor of one of the solutions is typical of representation decisions that developers often reverse at least once during a project's lifetime. So for continuity's sake it is desirable to have a feature access notation that does not distinguish between the two cases; then if you are in charge of *x*'s implementation and change your mind at some stage, it will not be necessary to change the modules that use *f*. This is an example of the Uniform Access principle.

In its general form the principle may be expressed as:

Uniform Access principle

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

Few languages satisfy this principle. An older one that did was Algol W, where both the function call and the access to a field were written *a* (*x*). Object-oriented languages should satisfy Uniform Access, as did the first of them, Simula 67, whose notation is *x*.*f* in both cases. The notation developed in part C will retain this convention.

The Open-Closed principle

Another requirement that any modular decomposition technique must satisfy is the Open-Closed principle:

Open-Closed principle

Modules should be both open and closed.

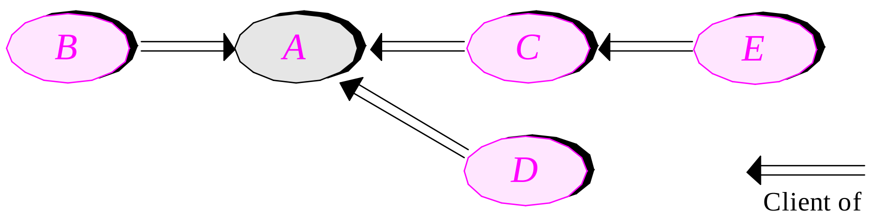
The contradiction between the two terms is only apparent as they correspond to goals of a different nature:

- A module is said to be open if it is still available for extension. For example, it should be possible to expand its set of operations or add fields to its data structures.
- A module is said to be closed if it is available for use by other modules. This assumes that the module has been given a well-defined, stable description (its interface in the sense of information hiding). At the implementation level, closure for a module also implies that you may compile it, perhaps store it in a library, and make it available for others (its *clients*) to use. In the case of a design or specification module, closing a module simply means having it approved by management, adding it to the project's official repository of accepted software items (often called the project *baseline*), and publishing its interface for the benefit of other module authors.

The need for modules to be closed, and the need for them to remain open, arise for different reasons. Openness is a natural concern for software developers, as they know that it is almost impossible to foresee all the elements — data, operations — that a module will need in its lifetime; so they will wish to retain as much flexibility as possible for future changes and extensions. But it is just as necessary to close modules, especially from a project manager's viewpoint: in a system comprising many modules, most will depend on some others; a user interface module may depend on a parsing module (for parsing command texts) and on a graphics module, the parsing module itself may depend on a lexical analysis module, and so on. If we never closed a module until we were sure it includes all the needed features, no multi-module software would ever reach completion: every developer would always be waiting for the completion of someone else's job.

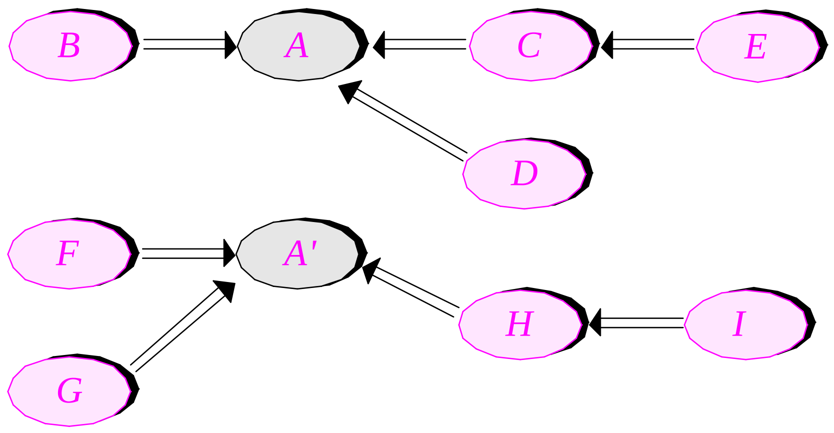
With traditional techniques, the two goals are incompatible. Either you keep a module open, and others cannot use it yet; or you close it, and any change or extension can trigger a painful chain reaction of changes in many other modules, which relied on the original module directly or indirectly.

The two figures below illustrate a typical situation where the needs for open and closed modules are hard to reconcile. In the first figure, module *A* is used by client modules *B*, *C*, *D*, which may themselves have their own clients (*E*, *F*, …).



A module and its clients

Later on, however, the situation is disrupted by the arrival of new clients — *B'* and others — which need an extended or adapted version of *A*, which we may call *A'*:



Old and new clients

With non-O-O methods, there seem to be only two solutions, equally unsatisfactory:

N1 • You may adapt module *A* so that it will offer the extended or modified functionality (*A'*) required by the new clients.

N2 • You may also decide to leave *A* as it is, make a copy, change the module's name to *A'* in the copy, and perform all the necessary adaptations on the new module. With this technique *A'* retains no further connection to *A*.

The potential for disaster with solution N1 is obvious. *A* may have been around for a long time and have many clients such as *B*, *C* and *D*. The adaptations needed to satisfy the new clients' requirements may invalidate the assumptions on the basis of which the old ones used *A*; if so the change to *A* may start a dramatic series of changes in clients, clients of clients and so on. For the project manager, this is a nightmare come true: suddenly, entire parts of the software that were supposed to have been finished and sealed off ages ago get reopened, triggering a new cycle of development, testing, debugging and documentation. If many a software project manager has the impression of living the Sisyphus syndrome — the impression of being sentenced forever to carry a rock to the top of the hill, only to see it roll back down each time — it is for a large part because of the problems caused by this need to reopen previously closed modules.

On the surface, solution N2 seems better: it avoids the Sisyphus syndrome since it does not require modifying any existing software (anything in the top half of the last figure). But in fact this solution may be even more catastrophic since it only postpones the day of reckoning. If you extrapolate its effects to many modules, many modification requests and a long period, the consequences are appalling: an explosion of variants of the original modules, many of them very similar to each other although never quite identical.

In many organizations, this abundance of modules, not matched by abundance of available functionality (many of the apparent variants being in fact quasi-clones), creates a huge *configuration management* problem, which people attempt to address through the use of complex tools. Useful as these tools may be, they offer a cure in an area where the first concern should be prevention. Better avoid redundancy than manage it.

Exercise E3.6, page 66, asks you to discuss how much need will remain for configuration management in an O-O context.

Configuration management will remain useful, of course, if only to find the modules which must be reopened after a change, and to avoid unneeded module recompilations.

But how can we have modules that are both open and closed? How can we keep *A* and everything in the top part of the figure unchanged, while providing *A'* to the bottom clients, and avoiding duplication of software? The object-oriented method will offer a particularly elegant contribution thanks to inheritance.

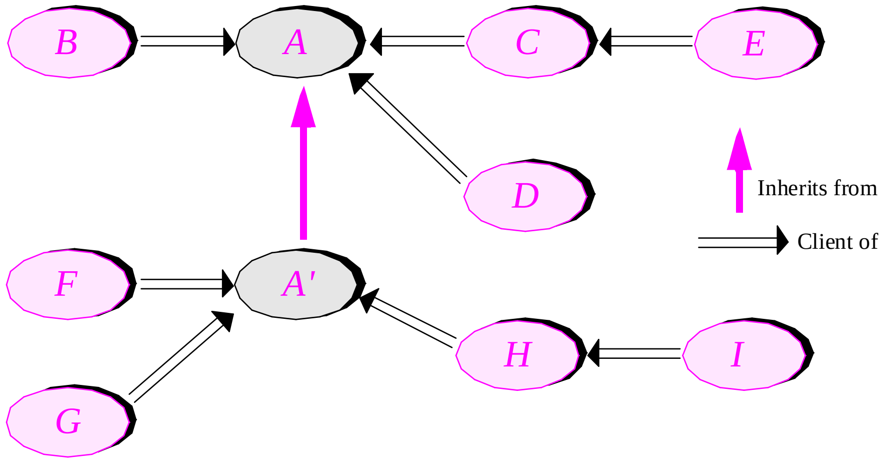
The detailed study of inheritance appears in later chapters, but here is a preview of the basic idea. To get us out of the *change or redo* dilemma, inheritance will allow us to define a new module *A'* in terms of an existing module *A* by stating the differences only. We will write *A'* as

```
class A' inherit
        A
                redefine f, g, … end
feature
        f is …
        g is …
        …
        u is …
        …
end
```

where the **feature** clause contains both the definition of the new features specific to *A'*, such as *u*, and the redefinition of those features (such as *f*, *g*, …) whose form in *A'* is different from the one they had in *A*.

The pictorial representation for inheritance will use an arrow from the heir (the new class, here *A'*) to the parent (here *A*):

Adapting a module to new clients



Thanks to inheritance, O-O developers can adopt a much more incremental approach to software development than used to be possible with earlier methods.

One way to describe the open-closed principle and the consequent object-oriented techniques is to think of them as a *organized hacking*. "Hacking" is understood here as a slipshod approach to building and modifying code (not in the more recent sense of breaking into computer networks, which, organized or not, no one should condone). The hacker may seem bad but often his heart is pure. He sees a useful piece of software, which is *almost* able to address the needs of the moment, more general than the software's original purpose. Spurred by a laudable desire not to redo what can be reused, our hacker starts modifying the original to add provisions for new cases. The impulse is good but the effect is often to pollute the software with many clauses of the form **if** *that_special_case* **then**…, so that after a few rounds of hacking, perhaps by a few different hackers, the software starts resembling a chunk of Swiss cheese that has been left outside for too long in August (if the tastelessness of this metaphor may be forgiven on the grounds that it does its best to convey the presence in such software of both holes and growth).

The organized form of hacking will enable us to cater to the variants without affecting the consistency of the original version.

A word of caution: nothing in this discussion suggests *dis*organized hacking. In particular:

- If you have control over the original software and can rewrite it so that it will address the needs of several kinds of client at no extra complication, you should do so.

- Neither the Open-Closed principle nor redefinition in inheritance is a way to address design flaws, let alone bugs. *If there is something wrong with a module, you should fix it* — not leave the original as it is and try to correct the problem in a derived module. (The only potential exception to this rule is the case of flawed software which you are not at liberty to modify.) The Open-Closed principle and associated techniques are intended for the adaptation of healthy modules: modules that, although they may not suffice for some new uses, meet their own well-defined requirements, to the satisfaction of their own clients.

Single Choice

The last of the five modularity principles may be viewed as a consequence of both the Open-Closed and Information Hiding rules.

Before examining the Single Choice principle in its full generality, let us look at a typical example. Assume you are building a system to manage a library (in the non-software sense of the term: a collection of books and other publications, not software modules). The system will manipulate data structures representing publications. You may have declared the corresponding type as follows in Pascal-Ada syntax:

```
type PUBLICATION =
    record
        author, title: STRING;
        publication_year: INTEGER
    case pubtype: (book, journal, conference_proceedings) of
        book: (publisher: STRING);
        journal: (volume, issue: STRING);
        proceedings: (editor, place: STRING)   -- Conference proceedings
    end
```

This particular form uses the Pascal-Ada notion of "record type with variants" to describe sets of data structures with some fields (here *author*, *title*, *publication_year*) common to all instances, and others specific to individual variants.

The use of a particular syntax is not crucial here; Algol 68 and C provide an equivalent mechanism through the notion of union type. A union type is a type *T* defined as the union of pre-existing types *A*, *B*, …: a value of type *T* is either a value of type *A*, or a value of type *B*, … Record types with variants have the advantage of clearly associating a tag, here *book*, *journal*, *conference_proceedings*, with each variant.

Let *A* be the module that contains the above declaration or its equivalent using another mechanism. As long as *A* is considered open, you may add fields or introduce new variants. To enable *A* to have clients, however, you must close the module; this means that you implicitly consider that you have listed all the relevant fields and variants. Let *B* be a typical client of *A*. *B* will manipulate publications through a variable such as

```
p: PUBLICATION
```

and, to do just about anything useful with *p*, will need to discriminate explicitly between the various cases, as in:

```
case p of
    book: … Instructions which may access the field p.publisher …
    journal: … Instructions which may access fields p.volume, p.issue …
    proceedings: … Instructions which may access fields p.editor, p.place …
end
```

The **case** instruction of Pascal and Ada comes in handy here; it is of course on purpose that its syntax mirrors the form of the declaration of a record type with variants. Fortran and C will emulate the effect through multi-target goto instructions (**switch** in C). In these and other languages a multi-branch conditional instruction (**if** … **then** … **elseif** … **elseif** … **else** … **end**) will also do the job.

Aside from syntactic variants, the principal observation is that to perform such a discrimination every client must know the exact list of variants of the notion of publication supported by *A*. The consequence is easy to foresee. Sooner or later, you will realize the need for a new variant, such as technical reports of companies and universities. Then you will have to extend the definition of type *PUBLICATION* in module *A* to support the new case. Fair enough: you have modified the conceptual notion of publication, so you should update the corresponding type declaration. This change is logical and inevitable. Far harder to justify, however, is the other consequence: any client of *A*, such as *B*, will also require updating if it used a structure such as the above, relying on an explicit list of cases for *p*. This may, as we have seen, be the case for most clients.

What we observe here is a disastrous situation for software change and evolution: a simple and natural addition may cause a chain reaction of changes across many client modules.

The issue will arise whenever a certain notion admits a number of variants. Here the notion was "publication" and its initial variants were book, journal article, conference proceedings; other typical examples include:

- In a graphics system: the notion of figure, with such variants as polygon, circle, ellipse, segment and other basic figure types.
- In a text editor: the notion of user command, with such variants as line insertion, line deletion, character deletion, global replacement of a word by another.
- In a compiler for a programming language, the notion of language construct, with such variants as instruction, expression, procedure.

In any such case, we must accept the possibility that the list of variants, although fixed and known at some point of the software's evolution, may later be changed by the addition or removal of variants. To support our long-term, software engineering view of the software construction process, we must find a way to **protect** the software's structure against the effects of such changes. Hence the Single Choice principle:

Single Choice principle

Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

By requiring that knowledge of the list of choices be confined to just one module, we prepare the scene for later changes: if variants are added, we will only have to update the module which has the information — the point of single choice. All others, in particular its clients, will be able to continue their business as usual.

See "DYNAMIC BINDING", 14.4, page 480.

Once again, as the publications example shows, traditional methods do not provide a solution; once again, object technology will show the way, here thanks to two techniques connected with inheritance: polymorphism and dynamic binding. No sneak preview in this case, however; these techniques must be understood in the context of the full method.

The Single Choice principle prompts a few more comments:

- The number of modules that know the list of choices should be, according to the principle, exactly one. The modularity goals suggest that we want *at most one* module to have this knowledge; but then it is also clear that *at least one* module must possess it. You cannot write an editor unless at least one component of the system has the list of all supported commands, or a graphics system unless at least one component has the list of all supported figure types, or a Pascal compiler unless at least one component "knows" the list of Pascal constructs.

- Like many of the other rules and principles studied in this chapter, the principle is about **distribution of knowledge** in a software system. This question is indeed crucial to the search for extendible, reusable software. To obtain solid, durable system architectures you must take stringent steps to limit the amount of information available to each module. By analogy with the methods employed by certain human organizations, we may call this a **need-to-know** policy: barring every module from accessing any information that is not strictly required for its proper functioning.

See the second figure on page 58.

- You may view the Single Choice principle as a direct consequence of the Open-Closed principle. Consider the publications example in light of the figure that illustrated the need for open-closed modules: *A* is the module which includes the original declaration of type *PUBLICATION*; the clients *B*, *C*, … are the modules that relied on the initial list of variants; *A'* is the updated version of *A* offering an extra variant (technical reports).

- You may also understand the principle as a strong form of Information Hiding. The designer of supplier modules such as *A* and *A'* seeks to hide information (regarding the precise list of variants available for a certain notion) from the clients.

3.4 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- The choice of a proper module structure is the key to achieving the aims of reusability and extendibility.
- Modules serve for both software decomposition (the top-down view) and software composition (bottom-up).
- Modular concepts apply to specification and design as well as implementation.
- A comprehensive definition of modularity must combine several perspectives; the various requirements may sometimes appear at odds with each other, as with decomposability (which encourages top-down methods) and composability (which favors a bottom-up approach).
- Controlling the amount and form of communication between modules is a fundamental step in producing a good modular architecture.
- The long-term integrity of modular system structures requires information hiding, which enforces a rigorous separation of interface and implementation.
- Uniform access frees clients from internal representation choices in their suppliers.
- A closed module is one that may be used, through its interface, by client modules.
- An open module is one that is still subject to extension.
- Effective project management requires support for modules that are both open and closed. But traditional approaches to design and programming do not permit this.
- The principle of Single Choice directs us to limit the dissemination of exhaustive knowledge about variants of a certain notion.

3.5 BIBLIOGRAPHICAL NOTES

The design method known as “structured design” [Yourdon 1979] emphasized the importance of modular structures. It was based on an analysis of module “cohesion” and “coupling”. But the view of modules implicit in structured design was influenced by the traditional notion of subroutine, which limits the scope of the discussion.

The principle of uniform access comes originally (under the name “uniform reference”) from [Geschke 1975].

The discussion of uniform access cited the Algol W language, a successor to Algol 60 and forerunner to Pascal (but offering some interesting mechanisms not retained in Pascal), designed by Wirth and Hoare and described in [Hoare 1966].

Information hiding was introduced in two milestone articles by David Parnas [Parnas 1972] [Parnas 1972a].

Configuration management tools that will recompile the modules affected by modifications in other modules, based on an explicit list of module dependencies, are based on the ideas of the Make tool, originally for Unix [Feldman 1979]. Recent tools — there are many on the market — have added considerable functionality to the basic ideas.

Some of the exercises below ask you to develop metrics to evaluate quantitatively the various informal measures of modularity developed in this chapter. For some results in O-O metrics, see the work of Christine Mingins [Mingins 1993] [Mingins 1995] and Brian Henderson-Sellers [Henderson-Sellers 1996a].

EXERCISES

E3.1 Modularity in programming languages

Examine the modular structures of any programming language which you know well and assess how they support the criteria and principles developed in this chapter.

E3.2 The Open-Closed principle (for Lisp programmers)

Many Lisp implementations associate functions with function names at run time rather than statically. Does this feature make Lisp more supportive of the Open-Closed principle than more static languages?

E3.3 Limits to information hiding

Can you think of circumstances where information hiding should *not* be applied to relations between modules?

E3.4 Metrics for modularity (term project)

The criteria, rules and principles of modularity of this chapter were all introduced through qualitative definitions. Some of them, however, may be amenable to quantitative analysis. The possible candidates include:

- Modular continuity.
- Few Interfaces.
- Small Interfaces.
- Explicit Interfaces.
- Information Hiding.
- Single Choice.

Explore the possibility of developing modularity metrics to evaluate how modular a software architecture is according to some of these viewpoints. The metrics should be size-independent: increasing the size of a system without changing its modular structure should not change its complexity measures. (See also the next exercise.)

E3.5 Modularity of existing systems

Apply the modularity criteria, rules and principles of this chapter to evaluate a system to which you have access. If you have answered the previous exercise, apply any proposed modularity metric.

Can you draw any correlations between the results of this analysis (qualitative, quantitative or both) and assessments of structural complexity for the systems under study, based either on informal analysis or, if available, on actual measurements of debugging and maintenance costs?

E3.6 Configuration management and inheritance

(This exercise assumes knowledge of inheritance techniques described in the rest of this book. It is not applicable if you have read this chapter as part of a first, sequential reading of the book.)

The discussion of the open-closed principle indicated that in non-object-oriented approaches the absence of inheritance places undue burden on configuration management tools, since the desire to avoid reopening closed modules may lead to the creation of too many module variants. Discuss what role remains for configuration management in an object-oriented environment where inheritance *is* present, and more generally how the use of object technology affects the problem of configuration management.

If you are familiar with specific configuration management tools, discuss how they interact with inheritance and other principles of O-O development.

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.

See "Programs with holes", page 506.

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.

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.

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

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:

- 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

"GENERALIZATION", 28.5, page 928

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.

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?

"Self-Documentation", page 54.

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.

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

More details in "Indexing clauses", page 890.

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

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.

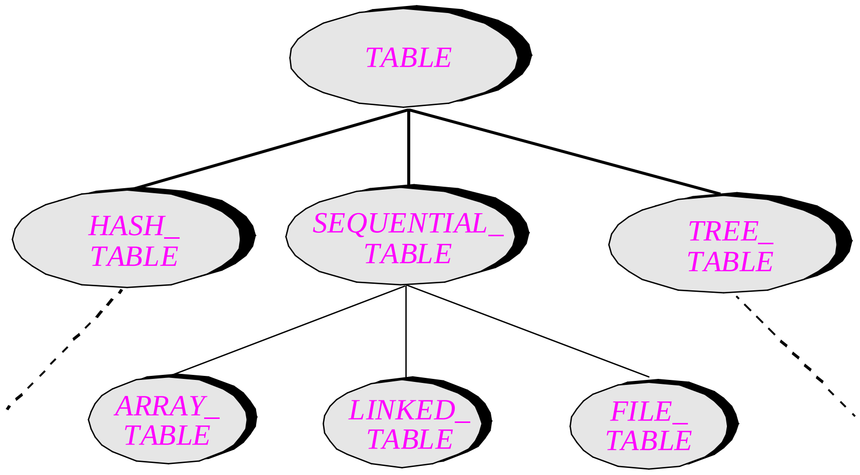
Factoring Out Common Behaviors

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:

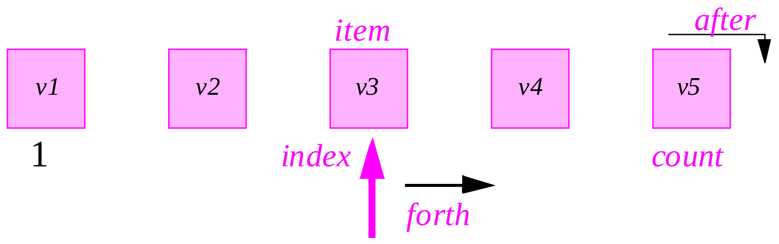
"ACTIVE DATA STRUCTURES", 23.4, page 774, will explore details of the cursor technique.

```
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



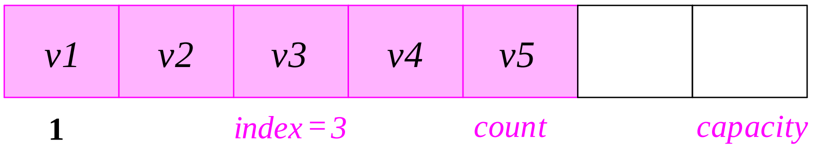
The general routine pattern was on page 82.

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*.

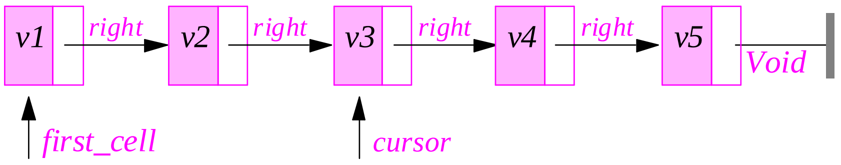
For each possible sequential table representation you will need a representation for the cursor. Three example representations are by an array, a linked list and a file.

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

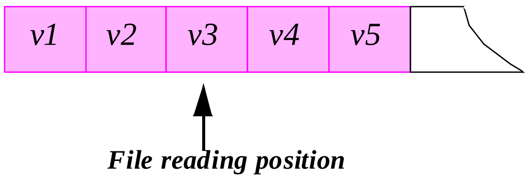


The second representation uses a linked list, where the first cell is accessible through a reference *first_cell* and each cell is linked to the next one through a reference *right*. Then you may represent the cursor as a reference *cursor*.



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].

5

Towards object technology

Extendibility, reusability and reliability, our principal goals, require a set of conditions defined in the preceding chapters. To achieve these conditions, we need a systematic method for decomposing systems into modules.

This chapter presents the basic elements of such a method, based on a simple but far-reaching idea: build every module on the basis of some object type. It explains the idea, develops the rationale for it, and explores some of the immediate consequences.

A word of warning. Given today's apparent prominence of object technology, some readers might think that the battle has been won and that no further rationale is necessary. This would be a mistake: we need to understand the basis for the method, if only to avoid common misuses and pitfalls. It is in fact frequent to see the word "object-oriented" (like "structured" in an earlier era) used as mere veneer over the most conventional techniques. Only by carefully building the case for object technology can we learn to detect improper uses of the buzzword, and stay away from common mistakes reviewed later in this chapter.

5.1 THE INGREDIENTS OF COMPUTATION

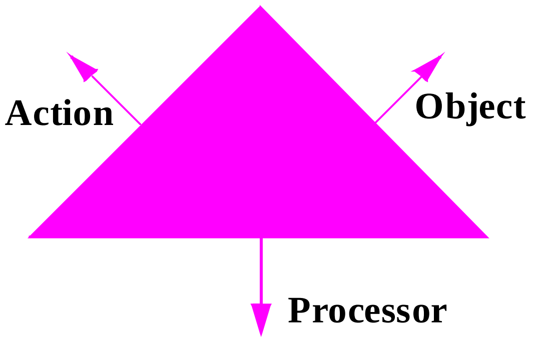
The crucial question in our search for proper software architectures is *modularization*: what criteria should we use to find the modules of our software?

To obtain the proper answer we must first examine the contending candidates.

The basic triangle

Three forces are at play when we use software to perform some computations:

The three forces of computation



To execute a software system is to use certain *processors* to apply certain *actions* to certain *objects*.

The processors are the computation devices, physical or virtual, that execute instructions. A processor can be an actual processing unit (the CPU of a computer), a process on a conventional operating system, or a "thread" if the OS is multi-threaded.

The actions are the operations making up the computation. The exact form of the actions that we consider will depend on the level of granularity of our analysis: at the hardware level, actions are machine language operations; at the level of the hardware-software machine, they are instructions of the programming language; at the level of a software system, we can treat each major step of a complex algorithm as a single action.

The objects are the data structures to which the actions apply. Some of these objects, the data structures built by a computation for its own purposes, are internal and exist only while the computation proceeds; others (contained in the files, databases and other persistent repositories) are external and may outlive individual computations.

Processors will become important when we discuss **concurrent** forms of computation, in which several sub-computations can proceed in parallel; then we will need to consider two or more processors, physical or virtual. But that is the topic of a later chapter; for the moment we can limit our attention to non-concurrent, or *sequential* computations, relying on a single processor which will remain implicit.

Concurrency is the topic of chapter 30.

This leaves us with actions and objects. The duality between actions and objects — what a system does *vs*. what it does it to — is a pervasive theme in software engineering.

A note of terminology. Synonyms are available to denote each of the two aspects: the word *data* will be used here as a synonym for *objects*; for *action* the discussion will often follow common practice and talk about the *functions* of a system.

The term "function" is not without disadvantages, since software discussions also use it in at least two other meanings: the mathematical sense, and the programming sense of subprogram returning a result. But we can use it without ambiguity in the phrase *the functions of a system*, which is what we need here.

The reason for using this word rather than "action" is the mere grammatical convenience of having an associated adjective, used in the phrase *functional decomposition*. "Action" has no comparable derivation. Another term whose meaning is equivalent to that of "action" for the purpose of this discussion is *operation*.

Any discussion of software issues must account for both the object and function aspects; so must the design of any software system. But there is one question for which we must choose — the question of this chapter: what is the appropriate criterion for finding the modules of a system? Here we must decide whether modules will be built as units of functional decomposition, or around major types of objects.

From the answer will follow the difference between the object-oriented approach and other methods. Traditional approaches build each module around some unit of functional decomposition — a certain piece of the action. The object-oriented method, instead, builds each module around some type of objects.

This book, predictably, develops the latter approach. But we should not just embrace O-O decomposition because the title of the book so implies, or because it is the "in" thing to do. The next few sections will carefully examine the arguments that justify using object types as the basis for modularization — starting with an exploration of the merits and limitations of traditional, non-O-O methods. Then we will try to get a clearer understanding of what the word "object" really means for software development, although the full answer, requiring a little theoretical detour, will only emerge in the next chapter.

We will also have to wait until the next chapter for the final settlement of the formidable and ancient fight that provides the theme for the rest of the present discussion: the War of the Objects and the Functions. As we prepare ourselves for a campaign of slander against the functions as a basis for system decomposition, and of corresponding praise for the objects, we must not forget the observation made above: in the end, our solution to the software structuring problem must provide space for both functions and objects — although not necessarily on an equal basis. To discover this new world order, we will need to define the respective roles of its first-class and second-class citizens.

5.2 FUNCTIONAL DECOMPOSITION

We should first examine the merits and limitations of the traditional approach: using functions as a basis for the architecture of software systems. This will not only lead us to appreciate why we need something else — object technology — but also help us avoid, when we do move into the object world, certain methodological pitfalls such as premature operation ordering, which have been known to fool even experienced O-O developers.

Continuity

"Modular continuity", page 44.

A key element in answering the question "should we structure systems around functions or around data?" is the problem of extendibility, and more precisely the goal called *continuity* in our earlier discussions. As you will recall, a design method satisfies this criterion if it yields stable architectures, keeping the amount of design change commensurate with the size of the specification change.

Continuity is a crucial concern if we consider the real lifecycle of software systems, including not just the production of an acceptable initial version, but a system's long-term evolution. Most systems undergo numerous changes after their first delivery. Any model of software development that only considers the period leading to that delivery and ignores the subsequent era of change and revision is as remote from real life as those novels which end when the hero marries the heroine — the time which, as everyone knows, marks the beginning of the really interesting part.

To evaluate the quality of an architecture (and of the method that produced it), we should not just consider how easy it was to obtain this architecture initially: it is just as important to ascertain how well the architecture will weather change.

Top-down design was sketched in "Modular decomposability", page 40.

The traditional answer to the question of modularization has been top-down functional decomposition, briefly introduced in an earlier chapter. How well does top-down design respond to the requirements of modularity?

Top-down development

There was a most ingenious architect who had contrived a new method for building houses, by beginning at the roof, and working downwards to the foundation, which he justified to me by the like practice of those two prudent insects, the bee and the spider.

Jonathan Swift: *Gulliver's Travels*, Part III, *A Voyage to Laputa*, *etc*., Chapter 5.

The top-down approach builds a system by stepwise refinement, starting with a definition of its abstract function. You start the process by expressing a topmost statement of this function, such as

```
[C0]

    "Translate a C program to machine code"
```

or:

```
[P0]

    "Process a user command"
```

and continue with a sequence of refinement steps. Each step must decrease the level of abstraction of the elements obtained; it decomposes every operation into a combination of one or more simpler operations. For example, the next step in the first example (the C compiler) could produce the decomposition

```
[C1]

    "Read program and produce sequence of tokens"
    "Parse sequence of tokens into abstract syntax tree"
    "Decorate tree with semantic information"
    "Generate code from decorated tree"
```

or, using an alternative structure (and making the simplifying assumption that a C program is a sequence of function definitions):

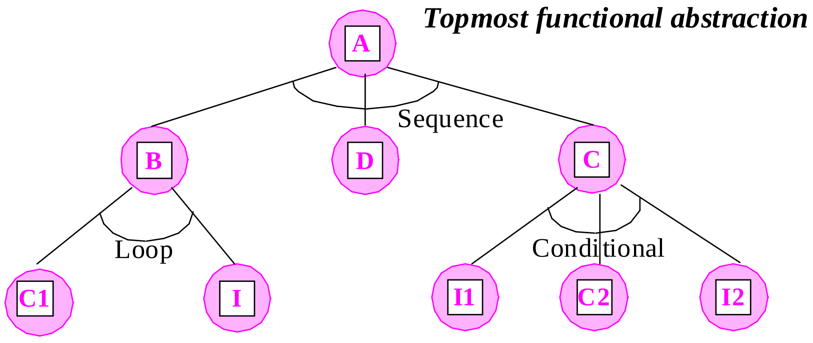
```
[C'1]

    from
        "Initialize data structures"
    until
        "All function definitions processed"
    loop
        "Read in next function definition"
        "Generate partial code"
    end
    "Fill in cross references"
```

In either case, the developer must at each step examine the remaining incompletely expanded elements (such as "Read program …" and "All function definitions processed") and expand them, using the same refinement process, until everything is at a level of abstraction low enough to allow direct implementation.

We may picture the process of top-down refinement as the development of a tree. Nodes represent elements of the decomposition; branches show the relation "*B* is part of the refinement of *A*".

Top-down design: tree structure

(*This figure first appeared on page* 41.)



The top-down approach has a number of advantages. It is a logical, well-organized thought discipline; it can be taught effectively; it encourages orderly development of systems; it helps the designer find a way through the apparent complexity that systems often present at the initial stages of their design.

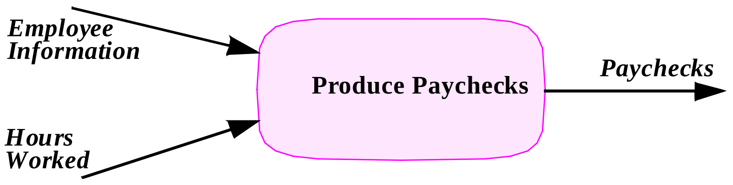
The top-down approach can indeed be useful for developing individual algorithms. But it also suffers from limitations that make it questionable as a tool for the design of entire systems:

- The very idea of characterizing a system by just one function is subject to doubt.
- By using as a basis for modular decomposition the properties that tend to change the most, the method fails to account for the evolutionary nature of software systems.

Not just one function

In the evolution of a system, what may originally have been perceived as the system's main function may become less important over time.

Consider a typical payroll system. When stating his initial requirement, the customer may have envisioned just what the name suggests: a system to produce paychecks from the appropriate data. His view of the system, implicit or explicit, may have been a more ambitious version of this:



Structure of a simple payroll program

The system takes some inputs (such as record of hours worked and employee information) and produces some outputs (paychecks and so on). This is a simple enough functional specification, in the strict sense of the word functional: it defines the program as a mechanism to perform one function — pay the employees. The top-down functional method is meant precisely for such well-defined problems, where the task is to perform a single function — the “top” of the system to be built.

Assume, however, that the development of our payroll program is a success: the program does the requisite job. Most likely, the development will not stop there. Good systems have the detestable habit of giving their users plenty of ideas about all the other things they could do. As the system’s developer, you may initially have been told that all you had to do was to generate paychecks and a few auxiliary outputs. But now the requests for extensions start landing on your desk: Could the program gather some statistics on the side? I did tell you that next quarter we are going to start paying some employees monthly and others biweekly, did I not? And, by the way, I need a summary every month for management, and one every quarter for the shareholders. The accountants want their own output for tax preparation purposes. Also, you are keeping all this salary information, right? It would be really nifty to let Personnel access it interactively. I cannot imagine why that would be a difficult functionality to add.

This phenomenon of having to add unanticipated functions to successful systems occurs in all application areas. A nuclear code that initially just applied some algorithm to produce tables of numbers from batch input will be extended to handle graphical input and output or to maintain a database of previous results. A compiler that just translated valid source into object code will after a while double up as a syntax verifier, a static analyzer, a pretty-printer, even a programming environment.

This change process is often incremental. The new requirements evolve from the initial ones in a continuous way. The new system is still, in many respects, “the same system” as the old one: still a payroll system, a nuclear code, a compiler. But the original “main function”, which may have seemed so important at first, often becomes just one of many functions; sometimes, it just vanishes, having outlived its usefulness.

If analysis and design have used a decomposition method based on the function, the system structure will follow from the designers’ original understanding of the system’s main function. As the system evolves, the designers may feel sorry (or its maintainers, if different people, may feel angry) about that original assessment. Each addition of a new function, however incremental it seems to the customer, risks invalidating the entire structure.

It is crucial to find, as a criterion for decomposition, properties less volatile than the system’s main function.

Finding the top

Top-down methods assume that every system is characterized, at the most abstract level, by its main function. Although it is indeed easy to specify textbook examples of algorithmic problems — the Tower of Hanoi, the Eight Queens and the like — through their functional "tops", a more useful description of practical software systems considers each of them as offering a number of services. Defining such a system by a single function is usually possible, but yields a rather artificial view.

Take an operating system. It is best understood as a system that provides certain services: allocating CPU time, managing memory, handling input and output devices, decoding and carrying out users' commands. The modules of a well-structured OS will tend to organize themselves around these groups of functions. But this is not the architecture that you will get from top-down functional decomposition; the method forces you, as the designer, to answer the artificial question "what is the topmost function?", and then to use the successive refinements of the answer as a basis for the structure. If hard pressed you could probably come up with an initial answer of the form

"Process all user requests"

which you could then refine into something like

```
from
    boot
until
    halted or crashed
loop
    "Read in a user's request and put it into input queue"
    "Get a request r from input queue"
    "Process r"
    "Put result into output queue"
    "Get a result o from output queue"
    "Output o to its recipient"
end
```

Refinements can go on. From such premises, however, it is unlikely that anyone can ever develop a reasonably structured operating system.

Even systems which may at first seem to belong to the "one input, one abstract function, one output" category reveal, on closer examination, a more diverse picture. Consider the earlier example of a compiler. Reduced to its bare essentials, or to the view of older textbooks, a compiler is the implementation of one input-to-output function: transforming source text in some programming language into machine code for a certain platform. But that is not a sufficient view of a modern compiler. Among its many services, a compiler will perform error detection, program formating, some configuration management, logging, report generation.

Another example is a typesetting program, taking input in some text processing format — $\TeX$, Microsoft Word, FrameMaker … — and generating output in HTML, Postscript or Adobe Acrobat format. Again we may view it at first as just an input-to-output filter. But most likely it will perform a number of other services as well, so it seems more interesting, when we are trying to characterize the system in the most general way, to consider the various types of data it manipulates: documents, chapters, sections, paragraphs, lines, words, characters, fonts, running heads, titles, figures and others.

The seemingly obvious starting point of top-down design — the view that each new development fulfills a request for a specific function — is subject to doubt:

Real systems have no top.

Functions and evolution

Not only is the main function often not the best criterion to characterize a system initially: it may also, as the system evolves, be among the first properties to change, forcing the top-down designer into frequent redesign and defeating our attempts to satisfy the continuity requirement.

Consider the example of a program that has two versions, a “batch” one which handles every session as a single big run over the problem, and an interactive one in which a session is a sequence of transactions, with a much finer grain of user-system communication. This is typical of large scientific programs, which often have a “let it run a big chunk of computation for the whole night” version and a “let me try out a few things and see the results at once then continue with something else” version.

The top-down refinement of the batch version might begin as

```
[B0] -- Top-level abstraction

        “Solve a complete instance of the problem”


[B1] -- First refinement

    “Read input values”

    “Compute results”

    “Output results”
```

and so on. The top-down development of the interactive version, for its part, could proceed in the following style:

[I1]

```
“Process one transaction”
```

[I2]

```
if “New information provided by the user” then
    “Input information”
    “Store it”
elseif “Request for information previously given” then
    “Retrieve requested information”
    “Output it”
elseif “Request for result” then
    if “Necessary information available” then
        “Retrieve requested result”
        “Output it”
    else
        “Ask for confirmation of the request”
        if Yes then
            “Obtain required information”
            “Compute requested result”
            “Output result”
        end
    end
else
    (Etc.)
```

Started this way, the development will yield an entirely different result. The top-down approach fails to account for the property that the final programs are but two different versions of the same software system — whether they are developed concurrently or one has evolved from the other.

This example brings to light two of the most unpleasant consequences of the top-down approach: its focus on the external interface (implying here an early choice between batch and interactive) and its premature binding of temporal relations (the order in which actions will be executed).

Interfaces and software design

System architecture should be based on substance, not form. But top-down development tends to use the most superficial aspect of the system — its external interface — as a basis for its structure.

The focus on external interfaces is inevitable in a method that asks “What will the system do for the end user?” as the key question: the answer will tend to emphasize the most external aspects.

The user interface is only one of the components of a system. Often, it is also among the most volatile, if only because of the difficulty of getting it right the first time; initial versions may be of the mark, requiring experimentation and user feedback to obtain a satisfactory solution. A healthy design method will try to separate the interface from the rest of the system, using more stable properties as the basis for system structuring.

It is in fact often possible to build the interface separately from the rest of the system, using one of the many tools available nowadays to produce elegant and user-friendly interfaces, often based on object-oriented techniques. The user interface then becomes almost irrelevant to the overall system design.

Chapter 32 discusses techniques and tools for user interfaces.

Premature ordering

The preceding examples illustrate another drawback of top-down functional decomposition: premature emphasis on temporal constraints. Each refinement expands a piece of the abstract structure into a more detailed *control* architecture, specifying the order in which various functions (various pieces of the action) will be executed. Such ordering constraints become essential properties of the system architecture; but they too are subject to change.

Recall the two alternative candidate structures for the first refinement of a compiler:

[C1]

```
"Read program and produce sequence of tokens"
"Parse sequence of tokens into abstract syntax tree"
"Decorate tree with semantic information"
"Generate code from decorated tree"
```

[C'1]

```
from
    "Initialize data structures"
until
    "All function definitions processed"
loop
    "Read in next function definition"
    "Generate partial code"
end

"Fill in cross references"
```

As in the preceding example we start with two completely different architectures. Each is defined by a control structure (a sequence of instructions in the first case, a loop followed by an instruction in the second), implying strict ordering constraints between the elements of the structure. But freezing such ordering relations at the earliest stages of design is not reasonable. Issues such as the number of passes in a compiler and the sequencing of various activities (lexical analysis, parsing, semantic processing, optimization) have many possible solutions, which the designers must devise by considering space-time tradeoffs and other criteria which they do not necessarily master at the beginning of a project. They can perform fruitful design and implementation work on the components long before freezing their temporal ordering, and will want to retain this sequencing freedom for as long as possible. Top-down functional design does not provide such flexibility: you must specify the order of executing operations before you have had a chance to understand properly what these operations will do.

See the bibliographical notes for references on the methods cited.

Some design methods that attempt to correct some of the deficiencies of functional top-down design also suffer from this premature binding of temporal relationships. This is the case, among others, with the dataflow-directed method known as structured analysis and with Merise (a method popular in some European countries).

Object-oriented development, for its part, stays away from premature ordering. The designer studies the various operations applicable to a certain kind of data, and specifies the effect of each, but defers for as long as possible specifying the operations' order of execution. This may be called the **shopping list** approach: list needed operations — all the operations that you may need; ignore their ordering constraints until as late as possible in the software construction process. The result is much more extendible architectures.

Ordering and O-O development

The observations on the risks of premature ordering deserve a little more amplification because even object-oriented designers are not immune. The shopping list approach is one of the least understood parts of the method and it is not infrequent to see O-O projects fall into the old trap, with damaging effects on quality. This can result in particular from misuse of the *use case* idea, which we will encounter in the study of O-O methodology.

Chapter 11 presents assertions.

The problem is that the order of operations may seem so obvious a property of a system that it will weasel itself into the earliest stages of its design, with dire consequences if it later turns out to be not so final after all. The alternative technique (under the "shopping list" approach), perhaps less natural at first but much more flexible, uses logical rather than temporal constraints. It relies on the assertion concept developed later in this book; we can get the basic idea now through a simple non-software example.

Consider the problem of buying a house, reduced (as a gross first approximation) to three operations: finding a house that suits you; getting a loan; signing the contract. With a method focusing on ordering we will describe the design as a simple sequence of steps:

[H]

```
    find_house
    get_loan
    sign_contract
```

In the shopping list approach of O-O development we will initially refuse to attach too much importance to this ordering property. But of course constraints exist between the operations: you cannot sign a contract unless (let us just avoid saying *until* for the time being!) you have a desired house and a loan. We can express these constraints in logical rather than temporal form:

[H'1]

```
find_property
        ensure
                property_found

get_loan
        ensure
                loan_approved

sign_contract
        require
                property_found and loan_approved
```

The notation will only be introduced formally in chapter 11, but it should be clear enough here: **require** states a precondition, a logical property that an operation requires for its execution; and **ensure** states a postcondition, a logical property that will follow from an operation's execution. We have expressed that each of the first two operations achieves a certain property, and that the last operation requires both of these properties.

Why is the logical form of stating the constraints, H'1, better than the temporal form, H1? The answer is clear: H'1 expresses the minimum requirements, avoiding the overspecification of H1. And indeed H1 is too strong, as it rules out the scheme in which you get the loan first and then worry about the property — not at all absurd for a particular buyer whose main problem is financing. Another buyer might prefer the reverse order; we should support both schemes as long as they observe the logical constraint.

Now imagine that we turn this example into a realistic model of the process with the many tasks involved — title search, termite inspection, pre-qualifying for the loan, finding a real estate agent, selling your previous house if applicable, inviting your friends to the house-warming party… It may be possible to express the ordering constraints, but the result will be complicated and probably fragile (you may have to reconsider everything if you later include another task). The logical constraint approach scales up much more smoothly; each operation simply states what it needs and what it guarantees, all in terms of abstract properties.

Exercise E6.7, page 162 (in the next chapter).

These observations are particularly important for the would-be object designer, who may still be influenced by functional ideas, and might be tempted to rely on early identification of system usage scenarios ("use cases") as a basis for analysis. This is incompatible with object-oriented principles, and often leads to top-down functional decomposition of the purest form — even when the team members are convinced that they are using an object-oriented method.

We will examine, in our study of O-O methodological principles, what role can be found for use cases in object-oriented software construction.

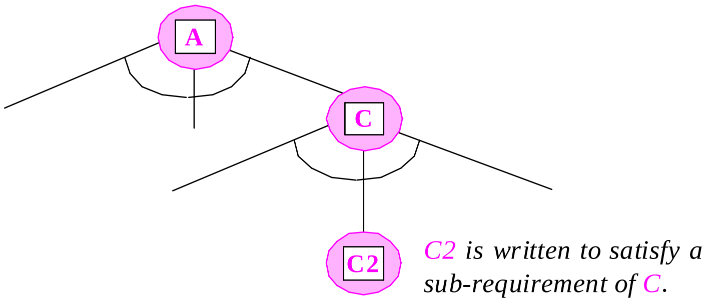
"Use cases", page 738.

Reusability

After this short advance incursion into the fringes of object territory, let us resume our analysis of the top-down method, considering it this time in relation to one of our principal goals, reusability.

Working top-down means that you develop software elements in response to particular subspecifications encountered in the tree-like development of a system. At a given point of the development, corresponding to the refinement of a certain node, you will detect the need for a specific function — such as analyzing an input command line — and write down its specification, which you or someone else will then implement.

The context of a module in top-down design



The figure, which shows part of a top-down refinement tree, illustrates this property: *C2* is written to satisfy some sub-requirement of *C*; but the characteristics of *C2* are entirely determined by its immediate context — the needs of *C*. For example, *C* could be a module in charge of analyzing some user input, and *C2* could be the module in charge of analyzing one line (part of a longer input).

This approach is good at ensuring that the design will meet the initial specification, but it does not promote reusability. Modules are developed in response to specific subproblems, and tend to be no more general than implied by their immediate context. Here if *C* is meant for input texts of a specific kind, it is unlikely that *C2*, which analyzes one line of those texts, will be applicable to any other kind of input.

One can in principle include the concern for extendibility and generality in a top-down design process, and encourage developers to write modules that transcend the immediate needs which led to their development. But nothing in the method encourages generalization, and in practice it tends to produce modules with narrow specifications.

The very notion of top-down design suggests the reverse of reusability. Designing for reusability means building components that are as general as possible, then combining them into systems. This is a bottom-up process, at the opposite of the top-down idea of starting with the definition of "the problem" and deriving a solution through successive refinements.

On the project and product culture see [M 1995].

This discussion makes top-down design appear as a byproduct of what we can call the *project culture* in software engineering: the view that the unit of discourse is the individual project, independently of earlier and later projects. The reality is less simple: project *n* in a company is usually a variation on project *n – 1*, and a preview of project *n + 1*. By focusing on just one project, top-down design ignores this property of practical software construction,

Production and description

One of the reasons for the original attraction of top-down ideas is that a top-down style may be convenient to explain a design once it is in place. But what is good to document an existing design is not necessarily the best way to produce designs. This point was eloquently argued by Michael Jackson in *System Development*:

Quotation from [Jackson 1983], pages 370-371.

> *Top-down is a reasonable way of describing things which are already fully understood... But top-down is not a reasonable way of developing, designing, or discovering anything. There is a close parallel with mathematics. A mathematical textbook describes a branch of mathematics in a logical order: each theorem stated and proved is used in the proofs of subsequent theorems. But the theorems were not developed or discovered in this way, or in this order...*
>
> *When the developer of a system, or of a program, already has a clear idea of the completed result in his mind, he can use top-down to describe on paper what is in his head. This is why people can believe that they are performing top-down design or development, and doing so successfully: they confuse the method of description with the method of development... When the top-down phase begins, the problem is already solved, and only details remain to be solved.*

Top-down design: an assessment

This discussion of top-down functional design shows the method to be poorly adapted to the development of significant systems. It remains a useful paradigm for small programs and individual algorithms; it is certainly a helpful technique to *describe* well-understood algorithms, especially in programming courses. But it does not scale up to large practical software. By developing a system top-down you trade short-term convenience for long-term inflexibility; you unduly privilege one function over the others; you may be led to devoting your attention to interface characteristics at the expense of more fundamental properties; you lose sight of the data aspect; and you risk sacrificing reusability.

5.3 OBJECT-BASED DECOMPOSITION

The case for using objects (or more precisely, as seen below, object types) as the key to system modularization is based on the quality aims defined in chapter 1, in particular extendibility, reusability and compatibility.

The plea for using objects will be fairly short, since the case has already been made at least in part: many of the arguments against top-down, function-based design reappear naturally as evidence in favor of bottom-up, object-based design.

This evidence should not, however, lead us to dismiss the functions entirely. As noted at the beginning of this chapter, no approach to software construction can be complete unless it accounts for both the function and object parts. So we will need to retain a clear role for functions in the object-oriented method, even if they must submit to the objects in the resulting system architectures. The notion of abstract data type will provide us with a definition of objects which reserves a proper place for the functions.

Extendibility

If the functions of a system, as discussed above, tend to change often over the system's life, can we find a more stable characterization of its essential properties, so as to guide our choice of modules and meet the goal of continuity?

The types of objects manipulated by the system are more promising candidates. Whatever happens to the payroll processing system used earlier as an example, it likely will still manipulate objects representing employees, salary scales, company regulations, hours worked, pay checks. Whatever happens to a compiler or other language processing tool, it likely will still manipulate source texts, token sequences, parse trees, abstract syntax trees, target code. Whatever happens to a finite element system, it likely will still manipulate matrices, finite elements and grids.

This argument is based on pragmatic observation, not on a proof that object types are more stable than functions. But experience seems to support it overwhelmingly.

The argument only holds if we take a high-level enough view of objects. If we understood objects in terms of their physical representations, we would not be much better off than with functions — as a matter of fact probably worse, since a top-down functional decomposition at least encourages abstraction. So the question of finding a suitably abstract description of objects is crucial; it will occupy all of the next chapter.

Reusability

The discussion of reusability pointed out that a routine (a unit of functional decomposition) was usually not sufficient as a unit of reusability.

See "Factoring Out Common Behaviors", page 85.

The presentation used a typical example: table searching. Starting with a seemingly natural candidate for reuse, a searching routine, it noted that we cannot easily reuse such a routine separately from the other operations that apply to a table, such as creation, insertion and deletion; hence the idea that a satisfactory reusable module for such a problem should be a collection of such operations. But if we try to understand the conceptual thread that unites all these operations, we find the type of objects to which they apply — tables.

Such examples suggest that object types, fully equipped with the associated operations, will provide stable units of reuse.

Compatibility

Another software quality factor, compatibility, was defined as the ease with which software products (for this discussion, modules) can be combined with each other.

It is difficult to combine actions if the data structures they access are not designed for that purpose. Why not instead try to combine entire data structures?

5.4 OBJECT-ORIENTED SOFTWARE CONSTRUCTION

We have by now accumulated enough background to consider a tentative definition of object-oriented software construction. This will only be a first attempt; a more concrete definition will follow from the discussion of abstract data types in the next chapter.

See page 147 for the final definition.

Object-oriented software construction (definition 1)

Object-oriented software construction is the software development method which bases the architecture of any software system on modules deduced from the types of objects it manipulates (rather than the function or functions that the system is intended to ensure).

An informal characterization of this approach may serve as a motto for the object-oriented designer:

OBJECT MOTTO

***Ask not first what the system does*:**

***Ask what it does it to*!**

To get a working implementation, you will of course, sooner or later, have to find out what it does. Hence the word *first*. Better later than sooner, says object-oriented wisdom. In this approach, the choice of main function is one of the very last steps to be taken in the process of system construction.

The developers will stay away, as long as possible, from the need to describe and implement the topmost function of the system. Instead, they will analyze the types of objects of the system. System design will progress through the successive improvements of their understanding of these object classes. It is a bottom-up process of building robust and extendible solutions to parts of the problem, and combining them into more and more powerful assemblies — until the final assembly which yields a solution of the original problem but, everyone hopes, is not the *only* possible one: the same components, assembled differently and probably combined with others, should be general enough to yield as a byproduct, if you have applied the method well and enjoyed your share of good luck, solutions to future problems as well.

For many software people this change in viewpoint is as much of a shock as may have been for others, in an earlier time, the idea of the earth orbiting around the sun rather than the reverse. It is also contrary to much of the established software engineering wisdom, which tends to present system construction as the fulfillment of a system's function as expressed in a narrow, binding requirements document. Yet this simple idea — look at the data first, forget the immediate purpose of the system — may hold the key to reusability and extendibility.

5.5 ISSUES

The above definition provides a starting point to discuss the object-oriented method. But besides providing components of the answer it also raises many new questions, such as:

- How to find the relevant object types.
- How to describe the object types.
- How to describe the relations and commonalities between object types.
- How to use object types to structure software.

The rest of this book will address these issues. Let us preview a few answers.

Finding the object types

See chapter 22.

The question "how shall we find the objects?" can seem formidable at first. A later chapter will examine it in some detail (in its more accurate version, which deals with object *types* rather than individual objects) but it is useful here to dispel some of the possible fears. The question does not necessarily occupy much of the time of experienced O-O developers, thanks in part to the availability of three sources of answers:

- Many objects are there just for the picking. They directly model objects of the physical reality to which the software applies. One of the particular strengths of object technology is indeed its power as a modeling tool, using software object types (classes) to model physical object types, and the method's inter-object-type relations (client, inheritance) to model the relations that exist between physical object types, such as aggregation and specialization. It does not take a treatise on object-oriented analysis to convince a software developer that a call monitoring system, in a telecommunications application, will have a class *CALL* and a class *LINE*, or that a document processing system will have a class *DOCUMENT*, a class *PARAGRAPH* and a class *FONT*.

- A source of object types is reuse: classes previously developed by others. This technique, although not always prominent in the O-O analysis literature, is often among the most useful in practice. We should resist the impulse to invent something if the problem has already been solved satisfactorily by others.

- Finally, experience and imitation also play a role. As you become familiar with successful object-oriented designs and design patterns (such as some of those described in this book and the rest of the O-O literature), even those which are not directly reusable in your particular application, you will be able to gain inspiration from these earlier efforts.

We will be in a much better position to understand these object-finding techniques and others once we have gained a better technical insight into the software notion of object — not to be confused with the everyday meaning of the word.

Describing types and objects

A question of more immediate concern, assuming we know how to obtain the proper object types to serve as a basis for modularizing our systems, is how to describe these types and their objects.

Two criteria must guide us in answering this question:

- The need to provide representation-independent descriptions, for fear of losing (as noted) the principal benefit of top-down functional design: abstraction.
- The need to re-insert the functions, giving them their proper place in software architectures whose decomposition is primarily based on the analysis of object types since (as also noted) we must in the end accommodate both aspects of the object-function duality.

The next chapter develops an object description technique achieving these goals.

Describing the relations and structuring software

Another question is what kind of relation we should permit between object types; since the modules will be based on object types, the answer also determines the structuring techniques that will be available to make up software systems from components.

In the purest form of object technology, only two relations exist: client and inheritance. They correspond to different kinds of possible dependency between two object types *A* and *B*:

- *B* is a client of *A* if every object of type *B* may contain information about one or more objects of type *A*.
- *B* is an heir of *A* if *B* denotes a specialized version of *A*.

Some widely used approaches to analysis, in particular information modeling approaches such as entity-relationship modeling, have introduced rich sets of relations to describe the many possible connections that may exist between the element of a system. To people used to such approaches, having to do with just two kinds of relation often seems restrictive at first. But this impression is not necessarily justified:

- The client relation is broad enough to cover many different forms of dependency. Examples include what is often called aggregation (the presence in every object of type *B* of a subobject of type *A*), reference dependency, and generic dependency.
- The inheritance relation covers specialization in its many different forms.
- Many properties of dependencies will be expressed in a more general form through other techniques. For example, to describe a 1-to-*n* dependency (every object of type *B* is connected to at least one and at most *n* objects of type *A*) we will express that *B* is a client of *A*, and include a **class invariant** specifying the exact nature of the client relation. The class invariant, being expressed in the language of logic, covers many more cases than the finite set of primitive relations offered by entity-relationship modeling or similar approaches.

5.6 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- Computation involves three kinds of ingredient: processors (or threads of control), actions (or functions), and data (or objects).
- A system's architecture may be obtained from the functions or from the object types.
- A description based on object types tends to provide better stability over time and better reusability than one based on an analysis of the system's functions.
- It is usually artificial to view a system as consisting of just one function. A realistic system usually has more than one "top" and is better described as providing a set of services.
- It is preferable not to pay too much attention to ordering constraints during the early stages of system analysis and design. Many temporal constraints can be described more abstractly as logical constraints.
- Top-down functional design is not appropriate for the long-term view of software systems, which involves change and reuse.
- Object-oriented software construction bases the structure of systems on the types of objects they manipulate.
- In object-oriented design, the primary design issue is not what the system does, but what types of objects it does it to. The design process defers to the last steps the decision as to what is the topmost function, if any, of the system.
- To satisfy the requirements of extendibility and reusability, object-oriented software construction needs to deduce the architecture from sufficiently abstract descriptions of objects.
- Two kinds of relation may exist between object types: client and inheritance.

5.7 BIBLIOGRAPHICAL NOTES

The case for object-based decomposition is made, using various arguments, in [Cox 1990] (original 1986), [Goldberg 1981], [Goldberg 1985], [Page-Jones 1995] and [M 1978], [M 1979], [M 1983], [M 1987], [M 1988].

The top-down method has been advocated in many books and articles. [Wirth 1971] developed the notion of stepwise refinement.

Of other methods whose rationales start with some of the same arguments that have led this discussion to object-oriented concepts, the closest is probably Jackson's JSD [Jackson 1983], a higher-level extension of JSP [Jackson 1975]. See also Warnier's data-directed design method [Orr 1977]. For a look at the methods that object technology is meant to replace, see books on: Constantine's and Yourdon's structured design [Yourdon 1979]; structured analysis [DeMarco 1978], [Page-Jones 1980], [McMenamin 1984], [Yourdon 1989]; Merise [Tardieu 1984], [Tabourier 1986].

Entity-relationship modeling was introduced by [Chen 1976].