1

Software quality

*E*ngineering 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

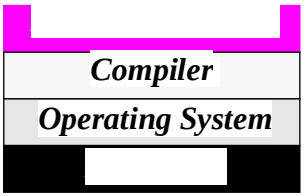
Definition: correctness

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:

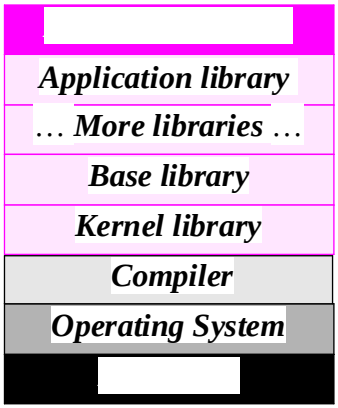


Layers in software development

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.

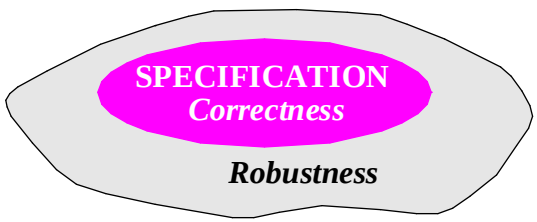
Robustness

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:

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

> *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.*

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.

On abstract data types see chapter 6.

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).

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.

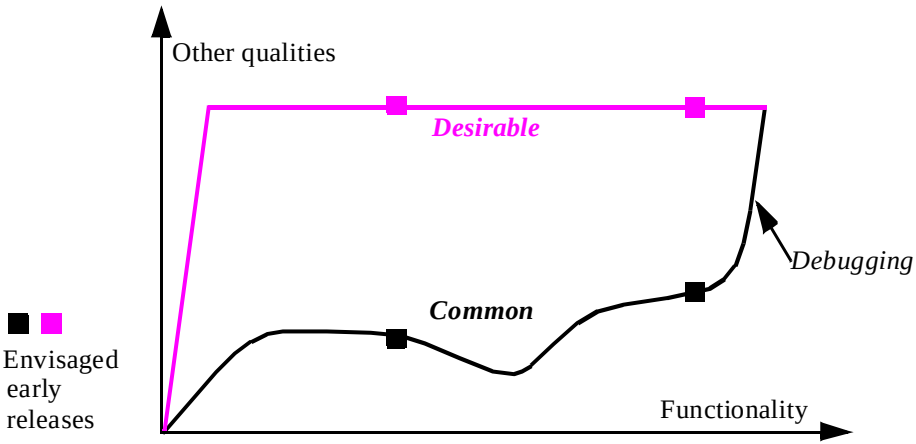
One of the most difficult problems facing a project leader is to know how much functionality is enough. The pressure for more facilities, known in industry parlance as *featurism* (often "*creeping featurism*"), is constantly there. Its consequences are bad for internal projects, where the pressure comes 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", Computer-World, 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.

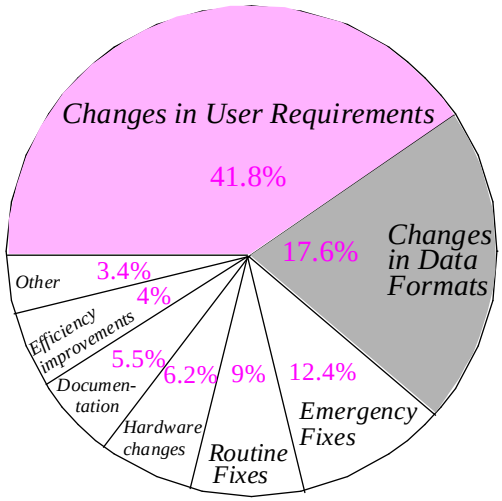
1.3 ABOUT SOFTWARE MAINTENANCE

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.