Chapter 2

Software: Its Nature And Qualities

The goal of any engineering activity is to build something—a product. The civil engineer builds a bridge, the aerospace engineer builds an airplane, and the electrical engineer builds a circuit. The product of software engineering is a “software system.” It is not as tangible as the other products, but it is a product nonetheless. It serves a function.

In some ways software products are similar to other engineering products, and in some ways they are very different. The characteristic that perhaps sets software apart from other engineering products the most is that software is malleable. We can modify the product itself—as opposed to its design—rather easily. This makes software quite different from other products such as cars or ovens.

The malleability of software is often misused. While it is certainly possible to modify a bridge or an airplane to satisfy some new need—for example, to make the bridge support more traffic or the airplane carry more cargo—such a modification is not taken lightly and certainly is not attempted without first making a design change and verifying the impact of the change extensively. Software engineers, on the other hand, are often asked to perform such modifications on software. Because of its malleability, we seem to think that changing software is easy. In practice, it is not.

We may be able to change the code easily with a text editor, but meeting the need for which the change was intended is not necessarily done so easily. Indeed, we need to treat software like other engineering products in this regard: a change in software must be viewed as a change in the design rather than in the code, which is just an instance of the product. We can indeed exploit the malleability property, but we need to do it with discipline.
Another characteristic of software is that its creation is human intensive: it requires mostly engineering rather than manufacturing. In most other engineering disciplines, the manufacturing process is considered carefully because it determines the final cost of the product. Also, the process has to be managed closely to ensure that defects are not introduced. The same considerations apply to computer hardware products. For software, on the other hand, "manufacturing" is a trivial process of duplication. The software production process deals with design and implementation, rather than manufacturing. This process has to meet certain criteria to ensure the production of high-quality software.

Any product is expected to fulfill some need and meet some acceptance standards that set forth the qualities it must have. A bridge performs the function of making it easier to travel from one point to another; one of the qualities it is expected to have is that it will not collapse when the first strong wind blows or a convoy of trucks travels across it. In traditional engineering disciplines, the engineer has tools for describing the qualities of the product distinctly from the design of the product. In software engineering, the distinction is not yet so clear. The qualities of the software product are often intermixed in specifications with the qualities of the design.

In this chapter, we examine the qualities that are pertinent to software products and software production processes. These qualities will become our goals in the practice of software engineering. In the next chapter, we will present software engineering principles that can be applied to achieve these goals. The presence of any quality will also have to be verified and measured. We will introduce this topic in Section 2.4, and we will study it in Chapter 6.

2.1 CLASSIFICATION OF SOFTWARE QUALITIES

There are many desirable software qualities. Some of these apply both to the product and to the process used to produce the product. The user wants the software product to be reliable, efficient, and easy to use. The producer of the software wants it to be verifiable, maintainable, portable, and extensible. The manager of the software project wants the process of software development to be productive and easy to control.

In this section, we consider two different classifications of software-related qualities: internal versus external and product versus process.

2.1.1 External Versus Internal Qualities

We can divide software qualities into external and internal qualities. The external qualities are visible to the users of the system; the internal qualities are those that concern the developers of the system. In general, users of the software only care about the external qualities, but it is the internal qualities—which deal largely with the structure of the software—that help developers achieve the external qualities. For example, the internal quality of verifiability is necessary for achieving the external quality of reliability. In many cases, however, the qualities are related closely and the distinction between internal and external is not sharp.
2.1.2 Product And Process Qualities

We use a process to produce the software product. We can also attribute some qualities to the process, although process qualities often are closely related to product qualities. For example, if the process requires careful planning of system test data before any design and development of the system starts, product reliability will increase. Some qualities, such as efficiency, apply both to the product and to the process.

It is interesting to examine the word product here. It usually refers to what is delivered to the customer. Even though this is an acceptable definition from the customer’s perspective, it is not adequate for the developer who requires a general definition of a software product that encompasses not only the object code and the user manual that are delivered to the customer, but also the requirements, design, source code, test data, etc. With such a definition, all of the artifacts that are produced during the process constitute parts of the product. In fact, it is possible to deliver different subsets of the same product to different customers.

For example, a computer manufacturer might sell to a process control company the object code to be installed in the specialized hardware for an embedded application. It might sell the object code and the user’s manual to software dealers. It might even sell the design and the source code to software vendors who modify them to build other products. In this case, the developers of the original system see one product, the salespersons in the same company see a set of related products, and the end user and the software vendor see still other, different products.

Configuration management is the part of the software production process that is concerned with maintaining and controlling the relationship between all the related pieces of the various versions of a product. Configuration management tools allow the maintenance of families of products and their components. We will discuss configuration management in Chapter 7.

2.2 REPRESENTATIVE QUALITIES

In this section, we present the most important qualities of software products and processes. Where appropriate, we analyze a quality with respect to the classifications discussed above.

2.2.1 Correctness, Reliability, And Robustness

The terms “correctness,” “reliability,” and “robustness” are often used interchangeably to characterize a quality of software that implies that the application performs its functions as expected. At other times, the terms are used with different meanings by different people, but the terminology is not standardized. This is quite unfortunate, because these terms deal with important issues. We will try to clarify these issues below, not only because we need a uniform terminology to be used throughout the book, but also because we believe that a clarification of the terminology is needed to better understand and analyze the underlying issues.
2.2.1.1 Correctness

A program is *functionally correct* if it behaves according to the specification of the functions it should provide (called *functional requirements specifications*). It is common simply to use the term “correct” rather than “functionally correct”; similarly, in this context, the term “specifications” implies “functional requirements specifications.” We will follow this convention when the context is clear.

The definition of correctness assumes that a specification of the system is available and that it is possible to determine unambiguously whether or not a program meets the specifications. With most current software systems, no such specification exists. If a specification does exist, it is usually written in an informal style using natural language. Such a specification is likely to contain many ambiguities. Regardless of these difficulties with current specifications, however, the definition of correctness is useful. Clearly, correctness is a desirable property for software systems.

Correctness is a mathematical property that establishes the equivalence between the software and its specification. Obviously, we can be more systematic and precise in assessing correctness depending on how rigorous we are in specifying functional requirements. As we will see in Chapter 6, correctness can be assessed through a variety of methods, some stressing an experimental approach (e.g., testing), others stressing an analytic approach (e.g., formal verification of correctness). Correctness can also be enhanced by using appropriate tools such as high-level languages, particularly those supporting extensive static analysis. Likewise, it can be improved by using standard algorithms or using libraries of standard modules, rather than inventing new ones.

2.2.1.2 Reliability

Informally, software is reliable if the user can depend on it.\(^1\) The specialized literature on software reliability defines reliability in terms of statistical behavior—the probability that the software will operate as expected over a specified time interval; we will discuss this approach in Section 6.7.2. For the purpose of this chapter, however, the informal definition is sufficient.

Correctness is an absolute quality: any deviation from the requirements makes the system incorrect, regardless of how minor or serious is the consequence of the deviation. The notion of reliability is, on the other hand, relative: if the consequence of a software error is not serious, the incorrect software may still be reliable.

Engineering products are expected to be reliable. Unreliable products, in general, disappear quickly from the marketplace. Unfortunately, software products have not achieved this enviable status yet. Software products are commonly released along with a list of “Known Bugs.” Users of software take it for granted that “Release 1” of a product is “buggy.” This is one of the most striking symptoms of the immaturity of the software engineering field as an engineering discipline.\(^2\)

In classic engineering disciplines, a product is not released if it has “bugs.” You do not expect to take delivery of an automobile along with a list of shortcomings or a bridge

\(^1\) “Dependable” is a term used as a synonym for “reliable.”

\(^2\) Dijkstra [1989] claims that even the sloppy term “bug,” which is often used by software engineers, is a symptom of unprofessionalism.
Sec. 2.2 Representative Qualities

with a warning not to use the railing. Design errors are extremely rare and worthy of news headlines. A bridge that collapses may even cause the designers to be prosecuted in court.

On the contrary, software design errors are generally treated as unavoidable. Far from being surprised with the occurrence of software errors, we expect them. Whereas with all other products the customer receives a guarantee of reliability, with software we get a disclaimer that the software manufacturer is not responsible for any damages due to product errors. Software engineering can truly be called an engineering discipline only when we can achieve software reliability comparable to the reliability of other products.

Figure 2.1 illustrates the relationship between reliability and correctness, under the assumption that the functional requirements specification indeed captures all the desirable properties of the application and that no undesirable properties are erroneously specified in it. The figure shows that the set of all reliable programs includes the set of correct programs, but not vice versa. Unfortunately, things are different in practice. In fact, the specification is a model of what the user wants, but the model may or may not be an accurate statement of the user's needs and actual requirements. All the software can do is meet the specified requirements of the model—it cannot assure the accuracy of the model.

Thus, Figure 2.1 represents an idealized situation where the requirements are themselves assumed to be correct, i.e., they are a faithful representation of what the implementation must ensure in order to satisfy the needs of the expected users. As we will discuss thoroughly in Chapter 7, there are often insurmountable obstacles to achieving this goal. The upshot is that we sometimes have correct applications that are designed for "incorrect" requirements, so that correctness of the software may not be sufficient to guarantee the user that the software behaves "as expected." This situations is discussed in the next subsection.

![Reliability and Correctness](image)

**Figure 2.1** Relationship between correctness and reliability in the ideal case.

### 2.2.1.3 Robustness

A program is robust if it behaves "reasonably," even in circumstances that were not anticipated in the requirements specification—for example, when it encounters incorrect input data or some hardware malfunction (say, a disk crash). A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command would not be robust. It might be correct,
though, if the requirements specification does not state what the action should be upon entry of an incorrect command. Obviously, robustness is a difficult-to-define quality; after all, if we could state precisely what we should do to make an application robust, we would be able to specify its "reasonable" behavior completely. Thus, robustness would become equivalent to correctness (or reliability, in the sense of Figure 2.1).

Again, an analogy with bridges is instructive. Two bridges connecting two sides of the same river are both "correct" if they each satisfy the stated requirements. If, however, during an unexpected, unprecedented, torrential rain, one collapses and the other one does not, we can call the latter more robust than the former. Notice that the lesson learned from the collapse of the bridge will probably lead to more complete requirements for future bridges, establishing the resistance to torrential rains as a correctness requirement. In other words, as the phenomenon under study becomes more and more known, we will approach the ideal case shown in Figure 2.1, where specifications capture exactly the expected requirements.

The amount of code devoted to robustness depends on the application area. For example, a system written to be used by novice computer users must be more prepared to deal with ill-formatted input than an embedded system that receives its input from a sensor—although, if the embedded system is controlling the space shuttle or some life-critical devices, then extra robustness is advisable.

In conclusion, we can see that robustness and correctness are strongly related, without a sharp dividing line between them. If we put a requirement in the specification, its accomplishment becomes an issue of correctness; if we leave it out of the specification, it may become an issue of robustness. The border line between the two qualities is the specification of the system. Finally, reliability comes in because not all incorrect behaviors signify equally serious problems; some incorrect behaviors may actually be tolerated.

Correctness, robustness, and reliability also apply to the software production process. A process is robust, for example, if it can accommodate unanticipated changes in the environment, such as a new release of the operating system or the sudden transfer of half the employees to another location. A process is reliable if it consistently leads to the production of high-quality products. In many engineering disciplines, considerable research is devoted to the discovery of reliable processes.

2.2.2 Performance

Any engineering product is expected to meet a certain level of performance. Unlike other disciplines, in software engineering we often equate performance with efficiency. We will follow this practice here. A software system is efficient if it uses computing resources economically.

Performance is important because it affects the usability of the system. If a software system is too slow, it reduces the productivity of the users, possibly to the point of not meeting their needs. If a software system uses too much disk space, it may be too expensive to run. If a software system uses too much memory, it may affect the other applications that are run on the same system, or it may run slowly while the operating system tries to balance the memory usage of the different applications.
Sec. 2.2 Representative Qualities

Underlying all of these statements—and also what makes the efficiency issue
difficult—are the changing limits of efficiency as technology changes. Our view of what is
"too expensive" is constantly changing as advances in technology extend the limits. The
computers of today cost orders of magnitude less than computers of a few years ago, yet
they provide orders of magnitude more power.

Performance is also important because it affects the scalability of a software system. An
algorithm that is quadratic may work on small inputs but not work at all for larger
inputs. For example, a compiler that uses a register allocation algorithm whose running
time is the square of the number of program variables will run slower and slower as the
length of the program being compiled increases.

There are several ways to evaluate the performance of a system. One method is to
measure efficiency by analyzing the complexity of algorithms. An extensive theory exists
for characterizing the average or worst case behavior of algorithms, in terms of
significant resource requirements such as time and space, or—less traditionally—in terms
of number of message exchanges (in the case of distributed systems).

Analysis of the complexity of algorithms provides only average or worst case
information, rather than specific information, about a particular implementation. For
more specific information, we can use techniques of performance evaluation. The three
basic approaches to evaluating the performance of a system are measurement, analysis,
and simulation. We can measure the actual performance of a system by means of
monitors that collect data while the system is working and allow us to discover bottlenecks in
the system. Or we can build a model of the product and analyze it. Or, finally, we can even build a model that simulates the product. Analytic models—often
based on queuing theory—are usually easier to build but are less accurate while simulation
models are more costly to build but are more accurate. We can combine the two
techniques as follows: at the start of a large project, an analytic model can provide a
general understanding of the performance-critical areas of the product, pointing out areas
where more thorough study is required; then we can build simulation models of these
particular areas.

In many software development projects, performance is addressed only after the
initial version of the product is implemented. It is very difficult—sometimes even
impossible—to achieve significant improvements in performance without redesigning the
software. Even a simple model, however, is useful for predicting system performance and
guiding design choices so as to minimize the need for redesign.

In some complex projects, where the feasibility of the performance requirements is
not clear, much effort is devoted to building performance models. Such projects start
with a performance model and use it initially to answer feasibility questions and later in
making design decisions. These models can help resolve issues such as whether a
function should be provided by software or a special-purpose hardware device.

The preceding remarks apply in the large, i.e., when the overall structure is
conceived. They often do not apply in the small, where individual programs may first be
designed with an eye toward ensuring correctness, and then be locally modified to
improve efficiency. For example, inner loops are obvious candidates for efficiency-
improving modifications.
The notion of performance also applies to a process, in which case we call it productivity. Productivity is important enough to be treated as an independent quality and is discussed as such in Section 2.2.10.

2.2.3 User Friendliness

A software system is user friendly if its human users find it easy to use. This definition reflects the subjective nature of user friendliness. An application that is used by novice programmers qualifies as user friendly by virtue of different properties than an application that is used by expert programmers. For example, a novice user may appreciate verbose messages, while an experienced user grows to detest and ignore them. Similarly, a nonprogrammer may appreciate the use of menus, while a programmer may be more comfortable with typing a command.

The user interface is an important component of user friendliness. A software system that presents the novice user with a window interface and a mouse is friendlier than one that requires the user to use a set of one-letter commands. On the other hand, an experienced user might prefer a set of commands that minimize the number of keystrokes rather than a fancy window interface through which he has to navigate to get to the command that he knew all along he wanted to execute. We will discuss user interface issues in Chapter 9.

There is more to user friendliness, however, than the user interface. For example, an embedded software system does not have a human user interface. Instead, it interacts with hardware and perhaps other software systems. In this case, the user friendliness is reflected in the ease with which the system can be configured and adapted to the hardware environment.

In general, the user friendliness of a system depends on the consistency of its user and operator interfaces. Clearly, however, the other qualities mentioned above—such as correctness and performance—also affect user friendliness. A software system that produces wrong answers is not friendly, regardless of how fancy its user interface is. Also, a software system that produces answers more slowly than the user requires is not friendly even if the answers are displayed in color.

User friendliness is also discussed under the subject "human factors." Human factors or human engineering plays a major role in many engineering disciplines. For example, automobile manufacturers devote significant effort to deciding the position of the various control knobs on the dashboard. Television manufacturers and microwave oven makers also try to make their products easy to use. User-interface decisions in these classical engineering fields are made, not randomly by engineers, but only after extensive study of user needs and attitudes by specialists in fields such as industrial design or psychology.

Interestingly, ease of use in many of these engineering disciplines is achieved through standardization of the human interface. Once a user knows how to use one television set, he or she can operate almost any other television set.\footnote{Although the new remote control devices are quite complicated!}
current research and development activity in the area of standard user interfaces for
software systems will lead to more user-friendly systems in the future.

**Exercise**

2.1 Discuss the relationship between the human-interface aspects of software and reliability.

**2.2.4 Verifiability**

A software system is verifiable if its properties can be verified easily. For example, the
correctness or the performance of a software system are properties we would be
interested in verifying. As we will see in Chapter 6, verification can be performed either
by formal analysis methods or through testing. A common technique for improving
verifiability is the use of “software monitors,” that is, code inserted in the software to
monitor various qualities such as performance or correctness.

Modular design, disciplined coding practices, and the use of an appropriate
programming language all contribute to verifiability.

Verifiability is usually an internal quality, although it sometimes becomes an
external quality also. For example, in many security-critical applications, the customer
requires the verifiability of certain properties. The highest level of the security standard
for a “trusted computer system” requires the verifiability of the operating system kernel.

**2.2.5 Maintainability**

The term “software maintenance” is commonly used to refer to the modifications that are
made to a software system after its initial release. Maintenance used to be viewed as
merely “bug fixing,” and it was distressing to discover that so much effort was being
spent on fixing defects. Studies have shown, however, that the majority of time spent on
maintenance is in fact spent on enhancing the product with features that were not in the
original specifications or were stated incorrectly there.

“Maintenance” is indeed not the proper word to use with software. First, as it is used
today, the term covers a wide range of activities, all having to do with modifying an
existing piece of software in order to make an improvement. A term that perhaps captures
the essence of this process better is “software evolution.” Second, in other engineering
products, such as computer hardware or automobiles or washing machines,
“maintenance” refers to the upkeep of the product in response to the gradual deterioration
of parts due to extended use of the product. For example, transmissions are oiled and air
filters are dusted and periodically changed. To use the word “maintenance” with software
gives the wrong connotation because software does not wear out. Unfortunately,
however, the term is used so widely that we will continue using it.

There is evidence that maintenance costs exceed 60% of the total costs of software.
To analyze the factors that affect such costs, it is customary to divide software
maintenance into three categories: corrective, adaptive, and perfective maintenance.
Corrective maintenance has to do with the removal of residual errors present in the product when it is delivered as well as errors introduced into the software during its maintenance. Corrective maintenance accounts for about 20 percent of maintenance costs.

Adaptive and perfective maintenance are the real sources of change in software; they motivate the introduction of evolvability (defined below) as a fundamental software quality and anticipation of change (defined in Chapter 3) as a general principle that should guide the software engineer. Adaptive maintenance accounts for nearly another 20 percent of maintenance costs while over 50 percent is absorbed by perfective maintenance.

Adaptive maintenance involves adjusting the application to changes in the environment, e.g., a new release of the hardware or the operating system or a new database system. In other words, in adaptive maintenance the need for software changes cannot be attributed to a feature in the software itself, such as the presence of residual errors or the inability to provide some functionality required by the user. Rather, the software must change because the environment in which it is embedded changes.

Finally, perfective maintenance involves changing the software to improve some of its qualities. Here, changes are due to the need to modify the functions offered by the application, add new functions, improve the performance of the application, make it easier to use, etc. The requests to perform perfective maintenance may come directly from the software engineer, in order to improve the status of the product on the market, or they may come from the customer, to meet some new requirements.

We will view maintainability as two separate qualities: repairability and evolvability. Software is repairable if it allows the fixing of defects; it is evolvable if it allows changes that enable it to satisfy new requirements.

The distinction between repairability and evolvability is not always clear. For example, if the requirements specifications are vague, it may not be clear whether we are fixing a defect or satisfying a new requirement. We will discuss this point further in Chapter 7. In general, however, the distinction between the two qualities is useful.

### 2.2.5.1 Repairability

A software system is repairable if it allows the correction of its defects with a limited amount of work. In many engineering products, repairability is a major design goal. For example, automobile engines are built with the parts that are most likely to fail as the most accessible. In computer hardware engineering, there is a subspecialty called repairability, availability, and serviceability (RAS).

In other engineering fields, as the cost of a product decreases and the product assumes the status of a commodity, the need for repairability decreases: it is cheaper to replace the whole thing, or at least major parts of it, than to repair it. For example, in early television sets, you could replace a single vacuum tube. Today, a whole board has to be replaced.

In fact, a common technique for achieving repairability in such products is to use standard parts that can be replaced easily. But software parts do not deteriorate. Thus, while the use of standard parts can reduce the cost of software production, the concept of replaceable parts does not seem to apply to software repairability. Software is also
Sec. 2.2 Representative Qualities

different in this regard because the cost of software is determined, not by tangible parts, but by human design activity.

Repairability is also affected by the number of parts in a product. For example, it is harder to repair a defect in a monolithic automobile body than if the body were made of several regularly shaped parts. In the latter case, we could replace a single part more easily than the whole body. Of course, if the body consisted of too many parts, it would require too many connections among the parts, leading to the probability that the connections themselves might need repair.

An analogous situation applies to software: a software product that consists of well-designed modules is much easier to analyze and repair than a monolithic one. Merely increasing the number of modules, however, does not make a more repairable product. We have to choose the right module structure with the right module interfaces to reduce the need for module interconnections. The right modularization promotes repairability by allowing errors to be confined to few modules, making it easier to locate and remove them. In Chapter 4, we will examine several modularization techniques, including information hiding and abstract data types, in detail.

Repairability can be improved through the use of proper tools. For example, using a high-level language rather than an assembly language leads to better repairability. Also, tools such as debuggers can help in isolating and repairing errors.

A product's repairability affects its reliability. On the other hand, the need for repairability decreases as reliability increases.

2.2.5.2 Evolvability

Like other engineering products, software products are modified over time to provide new functions or to change existing functions. Indeed, the fact that software is so malleable makes modifications extremely easy to apply to an implementation. There is, however, a major difference between software modification and modification of other engineering products. In the case of other engineering products, modifications start at the design level and then proceed to the implementation of the product. For example, if one decides to add a second story to a house, first one must do a feasibility study to check whether this can be done safely. Then one is required to do a design, based on the original design of the house. Then the design must be approved, after assessing that it does not violate the existing regulations. And, finally, the construction of the new part may be commissioned.

In the case of software, unfortunately, people seldom proceed in such an organized fashion. Although the change might be a radical change in the application, too often the implementation is started without doing any feasibility study, let alone a change in the original design. Still worse, after the change is accomplished, the modification is not even documented a posteriori; i.e., the specifications are not updated to reflect the change. This makes future changes more and more difficult to apply.

On the other hand, successful software products are quite long lived. Their first release is the beginning of a long lifetime and each successive release is the next step in the evolution of the system. If the software is designed with care, and if each modification is thought out carefully, then it can evolve gracefully.
As the cost of software production and the complexity of applications grow, the evolvability of software assumes more and more importance. One reason for this is the need to leverage the investment made in the software as the hardware technology advances. Some of the earliest large systems developed in the 1960s are today taking advantage of new hardware, device, and network technologies. For example, the American Airlines SABRE reservation system, initially developed in the middle 1960s, is still evolving with new functionality. This is an amazing feat considering the increasing performance demands on the system.

Most software systems start out being evolvable, but after years of evolution they reach a state where any major modification runs the risk of “breaking” existing features. In fact, evolvability is achieved by modularization and successive changes tend to reduce the modularity of the original system. This is even worse if modifications are applied without careful study of the original design and without precise description of changes in both the design and the requirements specification.

Indeed, studies of large software systems show that evolvability decreases with each release of a software product. Each release complicates the structure of the software, so that future modifications become more difficult. To overcome this problem, the initial design of the product, as well as any succeeding changes, must be done with evolvability in mind. Evolvability is one of the most important software qualities, and the principles we present in the next chapter will help achieve it. In Chapter 4, we present special concepts, such as program families, which are intended exactly for the purpose of fostering evolvability.

Evolvability is both a product- and process-related quality. In terms of the latter, the process must be able to accommodate new management and organizational techniques, changes in engineering education, etc.

2.2.6 Reusability

Reusability is akin to evolvability. In product evolution, we modify a product to build a new version of that same product; in product reuse, we use it—perhaps with minor changes—to build another product. Reusability appears to be more applicable to software components than to whole products but it certainly seems possible to build products that are reusable.

A good example of a reusable product is the UNIX shell. The UNIX shell is a command language interpreter; that is, it accepts user commands and executes them. But it is designed to be used both interactively and in “batch.” The ability to start a new shell with a file containing a list of shell commands allows us to write programs—scripts—in the shell command language. We can view the program as a new product that uses the shell as a component. By encouraging standard interfaces, the UNIX environment in fact supports the reuse of any of its commands, as well as the shell, in building powerful utilities.

Scientific libraries are the best known reusable components. Several large FORTRAN libraries have existed for many years. Users can buy these and use them to build their own products, without having to reinvent or recode well-known algorithms. Indeed, several companies are devoted to producing just such libraries.
Sec. 2.2 Representative Qualities

Another successful example of reusable packages is the recent development of windowing systems such as X windows or Motif, for the development of user interfaces. We will discuss these in Chapter 9.

Unfortunately, while reusability is clearly an important tool for reducing software production costs, examples of software reuse in practice are rather rare.

Reusability is difficult to achieve a posteriori, therefore, one should strive for reusability when software components are developed. In the next two chapters, we will examine some principles and techniques for achieving reusability. One of the more promising techniques is the use of object-oriented design, which can unify the qualities of evolvability and reusability.

So far, we have discussed reusability in the framework of reusable components, but the concept has broader applicability: it may occur at different levels and may affect both product and process. A simple and widely practiced type of reusability consists of the reuse of people, i.e., reusing their specific knowledge of an application domain, of a development or target environment, and so on. This level of reuse is unsatisfactory, partially due to the turnover of software engineers: knowledge goes away with people and never becomes a permanent asset.

Another level of reuse may occur at the requirements level. When a new application is conceived, we may try to identify parts that are similar to parts used in a previous application. Thus, we may reuse parts of the previous requirements specification instead of developing an entirely new one.

As discussed above, further levels of reuse may occur when the application is designed, or even at the code level. In the latter case, we might be provided with software components that are reused from a previous application. Some software experts claim that in the future new applications will be produced by assembling together a set of ready-made, off-the-shelf components. Software companies will invest in the development of their own catalogues of reusable components so that the knowledge acquired in developing applications will not disappear as people leave, but will progressively accumulate in the catalogues. Other companies will invest their efforts in the production of generalized reusable components to be put on the marketplace for use by other software producers.

Reusability applies to the software process as well. Indeed, the various software methodologies can be viewed as attempts to reuse the same process for building different products. The various life cycle models are also attempts at reusing higher level processes. Another example of reusability in a process is the "replay" approach to software maintenance. In this approach, the entire process is repeated when making a modification. That is, first the requirements are modified, and then the subsequent steps are followed as in the initial product development. More details on this will be given in Chapter 7.

Reusability is a key factor that characterizes the maturity of an industrial field. We see high degrees of reusability in such mature areas as the automobile industry and consumer electronics. For example, in the automobile industry, the engine is often reused from model to model. Moreover, a car is constructed by assembling together many components that are highly standardized and used across many models produced by the same industry. Finally, the manufacturing process is often reused. The low degree of
reusability in software is a clear indication that the field must evolve to achieve the status of a well-established engineering discipline.

Exercise

2.2 Discuss how reusability may affect the reliability of products.

2.2.7 Portability

Software is portable if it can run in different environments. The term “environment” can refer to a hardware platform or a software environment such as a particular operating system. With the proliferation of different processors and operating systems, portability has become an important issue for software engineers.

Even within one processor family, portability can be important because of the variations in memory capacity and additional instructions. One way to achieve portability within one machine architecture is to have the software system assume a minimum configuration as far as memory capacity is concerned and use a subset of the machine facilities that are guaranteed to be available on all models of the architecture (such as machine instructions and operating system facilities). But this penalizes the larger models because, presumably, the system can perform better on these models if it does not make such restrictive assumptions. Accordingly, we need to use techniques that allow the software to determine the capabilities of the hardware and to adapt to them. One good example of this approach is the way that UNIX allows programs to interact with many different terminals without explicit assumptions in the programs about the terminals they are using. The X Windows system extends this capability to allow applications to run on any bit-mapped display.

More generally, portability refers to the ability to run a system on different hardware platforms. As the ratio of money spent on software versus hardware increases, portability gains more importance.

Some software systems are inherently machine specific. For example, an operating system is written to control a specific computer, and a compiler produces code for a specific machine. Even in these cases, however, it is possible to achieve some level of portability. Again, UNIX is an example of an operating system that has been ported to many different hardware systems. Of course, the porting effort requires months of work. Still, we can call the software portable because writing the system from scratch for the new environment would require much more effort than porting it.

For many applications, it is important to be portable across operating systems. Or, looked at another way, the operating system provides portability across hardware platforms.

Exercise

2.3 Discuss portability as a special case of reusability.
2.2.8 Understandability

Some software systems are easier to understand than others. Of course, some tasks are inherently more complex than others. For example, a system that does weather forecasting, no matter how well it is written, will be harder to understand than one that prints a mailing list. Given tasks of inherently similar difficulty, we can follow certain guidelines to produce more understandable designs and to write more understandable programs.

Understandability is an internal product quality, and it helps in achieving many of the other qualities such as evolvability and verifiability. From an external point of view, the user considers the system understandable if it has predictable behavior. External understandability is a component of user friendliness.

2.2.9 Interoperability

Interoperability refers to the ability of a system to coexist and cooperate with other systems—for example, a word-processor’s ability to incorporate a chart produced by a graphing package, or the graphics package’s ability to graph the data produced by a spreadsheet, or the spreadsheet’s ability to process an image scanned by a scanner.

While rare in software products, interoperability abounds in other engineering products. For example, stereo systems from various manufacturers work together and can be connected to television sets and recorders. In fact, stereo systems produced decades ago accommodate new technologies such as compact discs, while virtually every operating system has to be modified—sometimes significantly—before it can work with the new optical disks.

Once again, the UNIX environment, with its standard interfaces, offers a limited example of interoperability within a single environment: UNIX encourages applications to have a simple, standard interface, which allows the output of one application to be used as the input to another.

The UNIX example also illustrates the limitations of interoperability in current systems: the UNIX standard interface is a primitive, character-oriented one. It is not easy for one application to use structured data—say, a spreadsheet or an image—produced by another application. Also, the UNIX system itself cannot operate in conjunction with other operating systems.

Another example of the limitations of interoperability in current software is illustrated by most personal computer software. Many vendors produce “integrated” products, meaning products that include several different functions. With better interoperability, the vendor could produce different products and allow the user to combine them if necessary. This would make it easier for the vendor to produce the products, and it would give the user more freedom in exactly what functions to pay for and to combine. In fact, in many instances, the vendor of the “integrated product” also has several independent products, each supporting a single function, but these products

1 There are exceptions in the Macintosh environment, where applications are achieving increasingly higher levels of interoperability.
do not work together. Interoperability can be achieved through standardization of interfaces.

A concept related to interoperability is that of an open system. An open system is an extensible collection of independently-written applications that cooperate to function as an integrated system. An open system allows the addition of new functionality by independent organizations, after the system is delivered. This can be achieved, for example, by releasing the system together with a specification of its "open" interfaces. Any applications developer can then take advantage of these interfaces. Some of the interfaces may be used for communication between different applications or systems. Open systems allow different applications, written by different organizations, to interoperate.

An interesting requirement of open systems is that new functionality may be added without taking the system down. An open system is analogous to a growing organization that evolves over time, adapting to changes in the environment. The importance of interoperability has sparked a growing interest in open systems, producing some recent standardization efforts in this area.

**Exercise**

2.4 Discuss the relationship between evolvability and open systems.

### 2.2.10 Productivity

Productivity is a quality of the software production process; it measures the efficiency of the process and, as we said before, is the performance quality applied to the process. An efficient process results in faster delivery of the product.

Individual engineers produce software at a certain rate, although there are great variations among individuals of different ability. When individuals are part of a team, the productivity of the team is some function of the productivity of the individuals. Very often, the combined productivity is much less than the sum of the parts. Process organizations and techniques are attempts at capitalizing on the individual productivity of team members.

Productivity offers many trade-offs in the choice of a process. For example, a process that requires specialization of individual team members may lead to productivity in producing a certain product, but not in producing a variety of products. Software reuse is a technique that leads to the overall productivity of an organization that is involved in developing many products, but developing reusable modules is harder than developing modules for one's own use, thus reducing the productivity of the group that is developing reusable modules as part of their product development.

While software productivity is of great interest due to the increasing cost of software, it is difficult to measure. Clearly, we need a metric for measuring productivity—or any other quality—if we are to have any hope of comparing different processes in terms of productivity. Early metrics such as the number of lines of code produced have many shortcomings. In Chapter 8, we will discuss metrics for measuring productivity and team
organizations for improving productivity. As in other engineering disciplines, we will see that efficiency of the process is affected strongly by automation. Modern software engineering tools and environments lead to increases in productivity. These tools will be discussed in Chapter 9.

Exercise

2.5 Critically evaluate number of lines of code as a productivity measure. (This issue will be analyzed in depth in Chapter 8.)

2.2.11 Timeliness

Timeliness is a process-related quality that refers to the ability to deliver a product on time. Historically, timeliness has been lacking in software production processes, leading to the "software crisis," which in turn led to the need for--and birth of--software engineering itself. Even now, many current processes fail to result in a timely product.

The following (real) example is typical of current (circa 1988) industry practice. The first release of an Ada compiler was promised by a computer manufacturer for a certain date. When the date arrived, the customers who had ordered the product received, instead of the product, a letter stating that since the product still contained many defects, the manufacturer had decided that it would be better to delay delivery rather than deliver a product containing defects. The product was promised for three months later.

In four months, the product arrived, along with a letter stating that many, but not all, of the defects had been corrected. But this time, the manufacturer had decided that it was better to let customers receive the Ada compiler, even though it contained several serious defects, so that the customers could start their own product development using Ada. The value of early delivery at this time had outweighed the cost of delivering a defective product, in the opinion of the manufacturer. So, in the end, what was delivered was late and defective.

Timeliness by itself is not a useful quality, although being late may sometimes preclude market opportunities. Delivering on time a product that is lacking in other qualities, such as reliability or performance, is pointless.

Timeliness requires careful scheduling, accurate work estimation, and clearly specified and verifiable milestones. All other engineering disciplines use standard project management techniques to achieve timeliness. There are even many computer-supported project management tools.

Standard project management techniques are difficult to apply in software engineering because of the difficulty in measuring the amount of work required for producing a given piece of software, the difficulty in measuring the productivity of engineers--or even having a dependable metric for productivity--and the use of imprecise and unverifiable milestones.

Another reason for the difficulty in achieving timeliness in the software process is continuously changing user requirements. Figure 2.2 plots user requirements against actual system capabilities and can show why most current software developments fail.
(The units of scale are not shown and can be assumed to be nonuniform.) At time $t_0$, the need for a software system is recognized and development starts with rather incomplete knowledge of the requirements. As a result, the initial product delivered at time $t_1$ does satisfy neither the initial requirements of time $t_0$, nor the user’s requirements of time $t_1$. Between time $t_1$ and time $t_3$, the product is “maintained,” in order to get closer to the user’s needs. Eventually, it matches the original user’s requirements at time $t_2$. For the reasons we have seen in Section 2.2.5.2, at time $t_3$ the cost of maintenance is so high that the software developer decides to do a major redesign. The new release becomes available at time $t_4$, but the gap with respect to the user’s needs at that point is even greater than before.

![Diagram](image)

**Figure 2.2** Software timeliness shortfall. (From Davis [1988], ©1988 IEEE, reprinted by permission of IEEE.)

One technique for achieving timeliness is through *incremental delivery* of the product. This technique is illustrated in the following—more successful—example of the delivery of an Ada compiler by a different (real) company from the one described before. This company delivered, very early on, a compiler that supported a very small subset of the Ada language—basically, a subset that was equivalent to Pascal with “packages.” The compiler did not support any of the novel features of the language, such as tasking and exception handling. The result was the early delivery of a reliable product. As a consequence, the users started experimenting with the new language and the company took more time to understand the subtleties of the new features of Ada. Over several releases, which took a period of two years, a full Ada compiler was delivered.

Incremental delivery allows the product to become available earlier; and the use of the product helps in refining the requirements incrementally. Outside of software engineering, a classic example of the difficulty in dealing with the requirements of complex systems is offered by modern weapons systems. In several well-publicized cases, the weapons have been obsolete by the time they have been delivered, or they have not met the requirements, or, in many cases, both. But after ten years of development, it
Sec. 2.2 Representative Qualities

is difficult to decide what to do with a product that does not meet a requirement stated ten years ago. The problem is exacerbated by the fact that requirements cannot be formulated precisely in these cases because the need is for the most advanced system possible at the time of delivery, not at the time the requirements are defined.

Obviously, incremental delivery depends on the ability to break down the set of required system functions into subsets that can be delivered in increments. If such subsets cannot be defined, no process can make the product available incrementally. But a nonincremental process prevents the production of product subsets even if such subsets can be identified. Timeliness can be achieved by a product that can be broken down into subsets and an incremental process.

Incremental delivery of useless subsets, of course, is not of value. Timeliness must be combined with other software qualities. Chapter 4 will discuss many techniques for achieving product subsets, and Chapter 7 will discuss techniques for achieving incremental processes.

2.2.12 Visibility

A software development process is visible if all of its steps and its current status are documented clearly. The term that describes this quality best is the Russian word glasnost. Other terms are “transparency” or “openness.” The idea is that the steps and the status of the project are available and easily accessible for external examination.

In many software projects, most engineers and even managers are unaware of the exact status of the project. Some may be designing, others coding, and still others testing, all at the same time. This, by itself, is not bad. Yet, if an engineer starts to redesign a major part of the code just before the software is supposed to be delivered for integration testing, the risk of serious problems and delays will be high.

Visibility allows engineers to weigh the impact of their actions and thus guides them in making decisions. It allows the members of the team to work in the same direction, rather than, as is often the case currently, in cross directions. The most common example of the latter situation is, as mentioned above, when the integration test group has been testing a version of the software assuming that the next version will involve fixing defects and will be only minimally different from the current version, while an engineer decides to do a major redesign to correct a minor defect. The tension between one group trying to stabilize the software while another person or group is destabilizing it—unintentionally, of course—is common. The process must encourage a consistent view of the status and current goals among all participants.

Visibility is not only an internal quality; it is also external. During the course of a long project, there are many requests about the status of the project. Sometimes these require formal presentations on the status, and at other times the requests are informal. Sometimes the requests come from the organization's management for future planning, and at other times they come from the outside, perhaps from the customer. If the software development process has low visibility, either these status reports will not be accurate, or they will require a lot of effort to prepare each time.

One of the difficulties of managing large projects is dealing with personnel turnover. In many software projects, critical information about the software requirements and design has the form of “folklore,” known only to people who have been with the project
either from the beginning or for a sufficiently long time. In such situations, recovering from the loss of a key engineer or adding new engineers to the project is very difficult. In fact, adding new engineers will often reduce the productivity of the whole project while the “folklore” is being transferred slowly from the existing crew of engineers to the new engineers.

The above points out that visibility of the process requires not only that all process steps be documented, but also that the current status of the intermediate products, such as requirements specifications and design specifications, be maintained accurately; that is, visibility of the product is required also. Intuitively, a product is visible if it is clearly structured as a collection of modules, with clearly understandable functions and easily accessible documentation.

2.3 QUALITY REQUIREMENTS IN DIFFERENT APPLICATION AREAS

The qualities we have described above are generic in the sense that they apply to any software system. But software systems are built to automate a particular application and therefore we can characterize a software system based on the requirements of the application area. In this section, we identify four major application areas of software systems and examine their additional requirements. We also show how they stress in different ways, some of the general qualities that we have discussed previously.

2.3.1 Information Systems

One of the largest and fastest growing application areas for computers is in the storage and retrieval of data. We call this class of systems “information-based systems” or simply “information systems” because the primary purpose of the system is managing information. Examples of information systems are banking systems, library-cataloguing systems, and personnel systems. At the heart of such systems is a data base against which we apply transactions to create, retrieve, update, or delete items.

Information systems have gained in importance because of the increasing value of information as a resource. The data that these systems manage is often the most valuable resource of an enterprise. Such data concern both the processes and the resources internal to the enterprise—plants, goods, people, etc.—and also information on external sources—competitors, suppliers, clients, etc. Increasingly, corporations are computerizing their manual procedures and trusting their data to computers. Their goal is to make their procedures more efficient and their data available on-line.

Information systems are data oriented and can be characterized on the basis of the way they treat data. Some of the qualities that characterize information systems are the following:

- **Data integrity.** Under what circumstances will the data be corrupted when the system malfunctions?

- **Security.** To what extent does the system protect the data from unauthorized access?
Chapter 3

Software Engineering Principles

In this chapter, we discuss some important and general principles that are central to successful software development. These principles deal with both the process of software engineering and the final product. The right process will help produce the right product, but the desired product will also affect the choice of which process to use. A traditional problem in software engineering has been the emphasis on either the process or the product to the exclusion of the other. Both are important.

The principles we develop are general enough to be applicable throughout the process of software construction and management. Principles, however, are not sufficient to drive software development. In fact, they are general and abstract statements describing desirable properties of software processes and products. But, to apply principles, the software engineer should be equipped with appropriate methods and specific techniques that help incorporate the desired properties into processes and products.

In principle, we should distinguish between methods and techniques. Methods are general guidelines that govern the execution of some activity; they are rigorous, systematic, and disciplined approaches. Techniques are more technical and mechanical than methods; often, they also have more restricted applicability. In general, however, the difference between the two is not sharp. We will therefore use the two terms interchangeably.

Sometimes, methods and techniques are packaged together to form a methodology. The purpose of a methodology is to promote a certain approach to solving a problem by preselecting the methods and techniques to be used. Tools, in turn, are developed to support the application of techniques, methods, and methodologies.
Figure 3.1 shows the relationship between principles, methods, methodologies, and tools. Each layer in the figure is based on the layer(s) below it and is more susceptible to change, due to passage of time. This figure shows clearly that principles are the basis of all methods, techniques, methodologies, and tools. The figure can also be used to explain the structure of this book. In this chapter, we present essential software engineering principles. In Chapters 4, 5, and 6, we present methods and techniques based on the principles of this chapter. Chapter 7 presents some methodologies, and Chapter 9 discusses tools and environments.

![Diagram showing the relationship between principles, methodologies, methods, and techniques]

**Figure 3.1** Relationship between principles, techniques, methodologies, and tools.

In our discussion of principles, we try to be general enough to cover every type of application. The same applies to the specific methods and techniques we develop in the chapters that follow. The emphasis we place on some principles and the particular methods and techniques we have selected, however, are deliberate choices. Among the qualities that were discussed in the previous chapter, we stress reliability and evolvability; and this choice, in turn, affects the emphasis on principles, methods, and techniques.

As mentioned in Chapter 1, we consider the case where the software to be developed is not just an experiment to be run a few times, maybe only by its own developer. Most likely, its expected users will have little or even no knowledge of computers and software. Or it might be required to support a critical application, where the effects of errors are serious, perhaps even disastrous. For these and other reasons, the application must be reliable.

Also, we assume that the application is sufficiently large and complex that special effort is required to decompose it into manageable parts. This is especially true in the likely case where the project is done by a team. But it is also true in the case of a single software engineer doing the job. In both cases, there is a need for an approach to software development that helps to overcome its complexity.

In all the above circumstances, which represent typical situations in software development, reliability and evolvability play a special role. Clearly, if the software does not have reliability and evolvability requirements, the need for software engineering principles and techniques diminishes greatly. In general, the choice of principles and techniques is determined by the software quality goals.

In this chapter, we discuss seven general and important principles that apply throughout the software development process: rigor and formality, separation of
concerns, modularity, abstraction, anticipation of change, generality, and incrementality. The list, by its very nature, cannot be exhaustive, but it does cover the important areas of software engineering. Although often the principles appear to be strongly related, we prefer to describe each of them separately below in quite general terms. They will be taken up in more concrete, detailed, and specific terms in the chapters that follow. In particular, the principle of modularity will be presented in Chapter 4 as the cornerstone of software design.

3.1 RIGOR AND FORMALITY

Software development is a creative activity. There is an inherent tendency in any creative process to be neither precise nor accurate, but rather to follow the inspiration of the moment in an unstructured manner. Rigor, on the other hand, is a necessary complement to creativity in every engineering activity: it is only through a rigorous approach that we can produce more reliable products, control their costs, and increase our confidence in their reliability. Rigor does not need to constrain creativity. Rather, it enhances creativity by improving the engineer's confidence in creative results, once they are critically analyzed in the light of a rigorous assessment.

Paradoxically, rigor is an intuitive quality that cannot be defined in a rigorous way. Also, various degrees of rigor can be achieved. The highest degree is what we call formality. Thus, formality is a stronger requirement than rigor: it requires the software process to be driven and evaluated by mathematical laws. Of course, formality implies rigor, but the converse is not true: one can be rigorous even in an informal setting.

In every engineering field, the design process proceeds as a sequence of well-defined, precisely stated, and supposedly sound steps. In each step, the engineer follows some method or applies some technique. The methods and techniques applied may be based on some combination of theoretical results derived by some formal modeling of reality, empirical adjustments that take care of phenomena not dealt with by the model, and rules of thumb that depend on past experience. The blend of these factors results in a rigorous and systematic approach—the methodology—that can be easily explained and applied time and again.

There is no need to be always formal during design, but the engineer must know how and when to be formal, should the need arise. For example, the engineer can rely on past experience and rules of thumb to design a short bridge, to be used temporarily to connect the two sides of a creek. She would instead use a mathematical model to verify whether the design is safe if the bridge were a long one that is supposed to stand permanently. She would use a more sophisticated mathematical model if the bridge were exceptionally long, or if it were built in a seismic area. In this case, the mathematical model would consider factors that could be ignored in the previous case.

Another—perhaps striking—example of the interplay between rigor and formality may be observed in mathematics. For example, textbooks on the calculus of functions are rigorous, but seldom formal: proofs of theorems are done in a very careful way, as sequences of intermediate deductions that lead to the final statement, where each deductive step relies on an intuitive justification that should convince the reader of its validity. Almost never, however, is the derivation of a proof stated in a formal way, in
terms of mathematical logic. This means that very often the mathematician is satisfied with a rigorous description of the derivation of a proof, without formalizing it completely. In critical cases, however, where the validity of some intermediate deduction is unclear, the mathematician may try to formalize the informal reasoning to assess its validity or falsity.

These examples show that the engineer (and the mathematician) must be able to understand the level of rigor and formality that should be achieved, depending on the conceptual difficulty of the task and its criticality. The level may even vary for different parts of the same system. For example, critical parts may deserve a formal description of their intended functions and a formal approach to their assessment. Well-understood and standard parts would require simpler approaches.

The same happens in the case of software engineering. Chapter 5 will go deeply into this issue in the context of software specifications. We will show that the description of what a program does may be given in a rigorous way by using natural language; it can also be given formally by providing a formal description in a language of logical statements. The advantage of formality over rigor is that formality may be the basis of mechanization of the process. For example, one may hope to use the formal description of the program to create the program (if the program does not yet exist) or to show that the program corresponds to the formal description (if the program and its formal specification exist).

Traditionally, there is only one phase of software development where a formal approach is used: programming. In fact, programs are formal objects: they are written in a language whose syntax and semantics are fully defined. Programs are formal descriptions that may be automatically manipulated by compilers: they are checked for formal correctness, transformed into an equivalent form in another language (assembly or machine language), "pretty-printed" so as to improve their appearance, etc. These mechanical operations, which are made possible by the use of formality in programming, can effectively improve the reliability and verifiability of software products.

Rigor and formality are not restricted to programming: they should be applied throughout the software process. Chapter 4 shows these concepts in action in the case of software design. Chapter 5 describes rigorous and formal approaches to software specification. Chapter 6 does the same for software verification.

So far, our discussion has emphasized the influence of rigor and formality on the reliability and verifiability of software products. Rigor and formality also have beneficial effects on maintainability, reusability, portability, understandability, and interoperability. For example, a rigorous, or even formal, software documentation can improve all of these qualities over informal documentation, which is often ambiguous, inconsistent, and incomplete.

Rigor and formality also apply to software processes. Rigorous documentation of a software process helps in reusing the process in other similar projects. Based on this documentation, managers may foresee the steps through which the new project will evolve, assign appropriate resources as needed, etc. Similarly, rigorous documentation of the software process may help maintain an existing product. If the various steps through which the project evolved are documented, one can modify an existing product starting from the appropriate intermediate level of its derivation, not the final code. More will be said on this crucial point in the following chapters. Finally, if the software process is
Sec. 3.2 Separation of Concerns

specified rigorously, managers may monitor it accurately, in order to assess its timeliness and improve productivity.

3.2 SEPARATION OF CONCERNS

Separation of concerns allows us to deal with different individual aspects of a problem, so that we can concentrate on each separately. Separation of concerns is a commonsense practice that we try to follow in our everyday life to master the difficulties we encounter. The principle should be applied also in the case of software development, to master its inherent complexity.

More specifically, there are many decisions that must be made in the development of a software product. Some of them concern features of the product as such: functions to offer, expected reliability, space and time efficiency, relationship with the environment (special hardware or software resources required), user interfaces, etc. Others concern the development process: development environment, team organization and structure, scheduling, control procedures, design strategies, error recovery mechanisms, etc. Still others concern economic and financial matters. These different decisions may be unrelated to one another. In such a case, it is obvious that they should be treated separately.

Very often, however, many decisions are strongly related and interdependent. For instance, a design decision (e.g., swapping some data from main memory to disk) may depend on the size of the memory of the selected target machine (and hence, the cost of the machine), and this, in turn, may affect the policy for error recovery. When different design decisions are strongly interconnected, it is practically impossible to take all the issues into account at the same time or by the same people.

The only way to master the complexity of the project is to separate the different concerns. First of all, one should try to isolate issues that are less intimately related to the others. Also, when issues are taken into account separately, related issues should not be considered in all details, but only insofar as they have an impact on the main issue under consideration.

There are various ways in which concerns may be separated. First of all, one can separate them in time. As an everyday life example, consider the case of a university professor who decides to deal with teaching activities by concentrating classes, seminars, office hours, and department meetings from 9 a.m. to 2 p.m. Monday through Thursday and leaving the rest of the time to research, except for Friday, which is devoted to consulting. Such temporal separation of concerns allows for precise planning of activities and eliminates overhead that would arise through switching from one activity to another in an unconstrained way. As we saw in Chapter 1 and will see in more detail in Chapter 7, separation of concerns in terms of time is the underlying motivation of the software life cycle, a rational model of the sequence of activities that should be followed in software production.

Another type of separation of concerns is in terms of qualities that should be treated separately. For example, in the case of software, we might wish to deal separately with the efficiency and the correctness of a given program. One might decide first to design software in such a careful and structured way that its correctness is expected to be
guaranteed \textit{a priori} and then to restructure the program partially to improve its efficiency. Similarly, in the verification phase, one might first check the functional correctness of the program and then its performance. Both activities can be done rigorously, applying some systematic procedures, or even formally, i.e., using formal correctness proofs and complexity analysis. Verification of program qualities is the subject of Chapter 6.

Another important type of separation of concerns allows different \textit{views} of the software to be analyzed separately. For example, when we analyze the requirements of an application, it may be helpful to concentrate separately on the data that flow from one activity to another in the system and the flow of control that governs the way different activities are synchronized. Both views help us understand the system we are working on better, although neither one gives a complete view of it.

Still another type of separation of concerns allows us to deal with \textit{parts} of the same system separately; here separation is in terms of size. This is a fundamental concept that we need to master to dominate the complexity of software production. It is so important that we prefer to detail it below as a separate point under modularity.

One may suspect that there is an inherent disadvantage in separation of concerns: by separating two or more issues, we might miss some global optimization that would be possible by tackling them together. Often, however, this objection overemphasizes our ability to make "optimized" decisions. By combining concerns, we are likely to make wrong decisions because we are unable to overcome complexity. Instead, the complexity of the global problem can be overcome much better by concentrating on different aspects separately, even at the expense of missing some potential optimizations.

Note that if two issues of one problem are intrinsically intertwined (i.e., the problem is not immediately decomposable into separate issues), it is often possible to make some overall design decisions first and then effectively separate the concern on the different issues. For example, consider a system where on-line transactions access a database concurrently. In a first implementation of the system, each transaction is supported by the underlying machine by locking the entire database at the start of the transaction and unlocking it at the end. Suppose now that a preliminary performance analysis shows that some transaction, say \( t_i \) (which might print some complex report extracting many data from the database), takes longer than we can afford to have database unavailable to other transactions. Thus, the problem is to revise the implementation in order to improve its performance yet maintain the overall correctness of the system. Clearly, the two issues—functional correctness and performance—are strongly related. Thus, a first design decision must concern both of them: \( t_i \) is no longer implemented as an atomic transaction, but is split into several subtransactions \( t_{i1}, t_{i2}, \ldots, t_{in} \), each being atomic. The new implementation may affect the correctness of the system, because of the interleaving that may occur between execution of any two subtransactions. Now, however, the two concerns of checking the functional correctness and analyzing performance have been separated, and two independent analyses can be made, maybe even by two different designers with different expertise.

As a final remark, note that separation of concerns may result in separation of responsibilities in dealing with separate issues. Thus, the principle is the basis for dividing the work on a complex problem into specific work assignments, possibly for different people with different skills. For example, by separating managerial and technical issues in the software process, we allow two types of people to cooperate in a
software project. As another example, having separated requirements analysis and specification from other activities in a software life cycle, we may hire specialized analysts with expertise in the application domain, instead of relying on internal resources. The analyst, in turn, may concentrate separately on functional and nonfunctional system requirements.

**Exercise**

3.1 Write a simple program in which you show that you can deal separately with correctness and efficiency.

### 3.3 MODULARITY

A complex system may be divided into simpler pieces called modules. A system that is composed of modules is called modular. The main benefit of modularity is that it allows the principle of separation of concerns to be applied in two phases: when dealing with the details of each module in isolation (and ignoring details of other modules) and when dealing with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system. If the two phases are temporarily executed in the order mentioned, then we say that the system is designed bottom up; the converse denotes top-down design.

Modularity is an important property of most engineering processes and products. For example, in the automobile industry, the construction of cars proceeds by assembling building blocks that are designed and built separately. Furthermore, parts are often reused from model to model, perhaps after minor changes. Most industrial processes are essentially modular, made out of work packages that are combined in simple ways (sequentially or overlapping) to achieve the desired result.

**Exercise**

3.2 Describe the work packages involved in building a house, and show how they are organized sequentially and in parallel.

We will emphasize modularity in the context of software design in the next chapter. Modularity, however, not only is a desirable design principle, but permeates the whole of software production. In particular, there are three goals that modularity tries to achieve in practice: capability of decomposing a complex system, of composing it from existing modules, and of understanding the system in pieces.

The decomposability of a system is based on dividing the original problem top down into subproblems and then applying the decomposition to each subproblem recursively. This procedure reflects the well-known Latin motto divide et impera (divide and
conquer), which describes the philosophy followed by the ancient Romans to dominate other nations: divide and isolate them first and conquer them individually.

The composability of a system is based on starting bottom up from elementary components and proceeding to the finished system. As an example, a system for office automation may be designed by assembling together existing hardware components such as personal workstations, a network, and peripherals; systems software such as the operating system; and productivity tools such as document processors, data bases, and spreadsheets. A car is another obvious example of a system that is built by assembling components. Consider first the main subsystems into which a car may be decomposed: the body, the electrical system, the power system, the transmission system, etc. Each of them, in turn, is made of standard parts; for example, the battery, fuses, cables, etc., form the electrical system. When something goes wrong, defective components may be replaced by new ones.

Ideally, in software production we would like to be able to assemble new applications by taking modules from a library and combining them to form the required product. Such modules should be designed with the express goal of being reusable. By using reusable components, we may speed up both the initial system construction and its fine-tuning. For example, it would be possible to replace a component by another that performs the same function but differs in computational resource requirements.

![Graphical description of cohesion and coupling](image)

(a) Graphical description of cohesion and coupling
(b) A highly coupled structure.
(c) A structure with high cohesion and low coupling.

The capability of understanding each part of a system separately aids in modifying a system. The evolutionary nature of software is such that the software engineer is often required to go back to previous work to modify it. If the entire system can be understood only in its entirety, modifications are likely to be difficult to apply, and the result unreliable. When the need for repair arises, proper modularity helps confine the search for the source of malfunction to single components.
Sec. 3.4 Abstraction

To achieve modular composability, decomposability, and understanding, modules must have high cohesion and low coupling.

A module has high cohesion if all of its elements are related strongly. Elements of a module (e.g., statements, procedures, and declarations) are grouped together in the same module for a logical reason, not just by chance; they cooperate to achieve a common goal, which is the function of the module.

Whereas cohesion is an internal property of a module, coupling characterizes a module’s relationship to other modules. Coupling measures the interdependence of two modules (e.g., module A calls a routine provided by module B or accesses a variable declared by module B). If two modules depend on each other heavily, they have high coupling. Ideally, we would like modules in a system to exhibit low coupling, because if two modules are highly coupled, it will be difficult to analyze, understand, modify, test, or reuse them separately. Figure 3.2 provides a graphical view of cohesion and coupling.

A good example of a system that has high cohesion and low coupling is the electric subsystem of a house. Because it is made out of a set of appliances with clearly definable functions and interconnected by simple wires, the system has low coupling. Because each appliance’s internal components are there exactly to provide the service the appliance is supposed to provide, the system has high cohesion.

Modular structures with high cohesion and low coupling allow us to see modules as black boxes when the overall structure of a system is described and then deal with each module separately when the module’s functionality is described or analyzed. This is just another example of the principle of separation of concerns.

Exercises

3.3 Suppose you decide to modularize the description of a car by splitting it into small cubes 15 inches on a side. Discuss this modularization in terms of cohesion and coupling. Propose a better way of modularizing the description, if any. Draw general conclusions about how one should modularize a complex system.

3.4 Explain some of the causes of and remedies for low cohesion in a software module.

3.5 Explain some of the causes of and remedies for high coupling between two software modules.

3.4 ABSTRACTION

Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details. Thus, abstraction is a special case of separation of concerns wherein we separate the concern of the important aspects from the concern of the unimportant details.

What we abstract away and consider as a detail that may be ignored depends on the purpose of the abstraction. For example, consider a quartz watch. A useful abstraction for the owner is a description of the effects of pushing its various buttons, which allow the watch to enter various functioning modes and react differently to sequences of commands. A useful abstraction for the person in charge of maintaining the watch is a
box that can be opened in order to replace the battery. Still other abstractions of the
device are useful for understanding the quartz watch and mastering the activities that are
needed to repair it (let alone design it). Thus, there may be many different abstractions of
the same reality, each providing a view of the reality and serving some specific purpose.

Exercise

3.6 Different people interacting with a software application may require different abstractions.
Comment briefly on what types of abstractions are useful for the end user, the designer,
and the maintainer.

Abstraction is a powerful technique practiced by engineers of all fields for mastering
complexity. For example, the representation of an electrical circuit in terms of resistors,
capacitors, etc., each characterized by some model in terms of equations, is an idealized
abstraction of a device. On the one hand, the equations are a simplified model that
approximates the behavior of the real components; on the other, the model we build often
ignores details such as the fact that there are no "pure" connectors between components
and that these connectors should also be modeled in terms of resistors, capacitors, etc.
Both facts can be ignored by the designer because the effects they describe are negligible
in terms of the results that we wish to observe.

This example illustrates an important general idea: the models we build of
phenomena—such as the equations for describing devices—are an abstraction from reality,
ignoring certain facts and concentrating on others that are deemed relevant. The same
holds for the models built and analyzed by software engineers. For example, when the
requirements for a new application are analyzed and specified, software engineers build a
model of the proposed application. As we will see in Chapter 5, this model may be
expressed in various forms, depending on the required degree of rigor and formality. No
matter what language we use for expressing requirements—be it natural language or the
formal language of mathematical formulas—what we provide is a model that abstracts
away from a number of details that we decide can be ignored safely.

Abstraction permeates the whole of programming. The programming languages that
we use are abstractions built on top of the hardware: they provide us with useful and
powerful constructs so that we can write (most) programs ignoring such details as the
number of bits that are used to represent numbers or the addressing mechanism. This
helps us concentrate on the problem to solve rather than the way to instruct the machine
on how to solve it. The programs we write are themselves abstractions. For example, a
computerized payroll procedure is an abstraction over the manual procedure it replaces: it
provides the essence of the manual procedure, not its exact details.

Abstraction is an important principle that applies to both software products and
processes. For example, the comments that appear in the header of a procedure are an
abstraction that describes the effect of the procedure. When the documentation of the
program is analyzed, such comments are supposed to provide all the information that is
needed to understand the other parts of the program that use the procedure.

As an example of the use of abstraction in software processes, consider the case of
cost estimation for a new application. One possible way of doing cost estimation consists*
Sec. 3.5 Anticipation of Change

of identifying some key factors of the new system and extrapolating from the cost profiles of previous similar systems. The key factors used to perform the analysis are abstractions of the system.

Exercises

3.7 Variables appearing in a programming language may be viewed as abstractions of memory locations. What details are abstracted away by programming language variables? What are the advantages of using the abstraction? What are the disadvantages?

3.8 A software life cycle model, such as the waterfall model outlined in Chapter 1, is an abstraction of a software process. Why?

3.5 ANTICIPATION OF CHANGE

Software undergoes changes constantly. As we saw in Chapter 2, changes are due both to the need for repairing the software—eliminating errors that were not detected before releasing the application—and to the need for supporting evolution of the application as new requirements arise or old requirements change. This is why we identified maintainability as a major software quality.

The ability of software to evolve does not come for free—it requires special effort to anticipate how and where the changes are likely to occur. When likely changes are identified, special care must be taken to proceed in a way that will make future changes easy to apply. We will see this important point in action in Chapter 4 in the case of design. We will show how software can be designed such that likely changes that we anticipate in the requirements, or modifications that are planned as part of the design strategy, may be incorporated in the application smoothly and safely. Basically, likely changes should be isolated in specific portions of the software in such a way that changes will be restricted to such small portions.

Anticipation of change is perhaps the one principle that distinguishes software the most from other types of industrial productions. In many cases, a software application is developed while its requirements are not entirely understood. Then, after being released, based on feedback from the user, the application must evolve as new requirements are discovered or old requirements are updated. In addition, applications are often embedded in an environment, such as an organizational structure. The environment is affected by the introduction of the application, and this generates new requirements that were not present initially. Thus, anticipation of changes is a principle that we can use to achieve evolvability.

Reusability is another software quality that is strongly affected by anticipation of change. As we saw, a component is reusable if it is directly usable to produce a new product. More realistically, it might undergo slight changes before it can be reused. As such, reusability may be viewed as low-grain evolvability, i.e., evolvability at the component level. If we can anticipate the changes of context in which a software