XI

Software Engineering

Steven A. Demurjian, Sr., Section Advisor

The University of Connecticut

Graphics and Visual Computing is the study and realization of complex process for representing physical and conceptual objects visually on a computer screen. Fundamental to all graphics applications are the processes of modeling objects abstractly and rendering them on a computer screen. Also important are object identification, light, color, shading, projection, and animation. The reconstruction of scanned images and the virtual simulation of reality are also of major research interest, as is the ultimate goal of simulating human vision itself.

101 Software Qualities and Principles *Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli* **101**-1
Classification of Software Qualities • Representative Software Qualities • Quality Assurance • Software Engineering Principles in Support of Software Quality • A Case Study in Compiler Construction • Summarizing Remarks

102 Software Process Models *Ian Sommerville* **102**-1
Introduction • Specification-Driven Models • Evolutionary Development Models • Iterative Models • Formal Transformation • The Cleanroom Process • Process Model Applicability • Research Issues and Summary

103 Traditional Software Design *Steven A. Demurjian, Sr.* **103**-1
Introduction • The High-Tech Supermarket System • Traditional Approaches to Design • Design by Encapsulation and Hiding • Mathematical and Analytical Design

104 Object-Oriented Software Design *Steven A. Demurjian, Sr. and Patricia J. Pia* ... **104**-1
Introduction • Object-Oriented Concepts and Terms • Choosing Classes • Inheritance: Motivation, Usage, and Costs • Categories of Classes and Design Flaws • The Unified Modeling Language • Design Patterns

105 Software Testing *Gregory M. Kapfhammer* **105**-1
Introduction • Underlying Principles • Best Practices • Conclusion

106 Formal Methods *Jonathan P. Bowen and Michael G. Hinchey* **106**-1
Introduction • Underlying Principles • Best Practices • A Case Study • Technology Transfer and Research Issues • Glossary of Z Notation

107 Verification and Validation *John D. Gannon* **107**-1
Introduction • Approaches to Verification • Verifying Specifications of Systems • Verifying Programs • Current Status

108 Development Strategies and Project Management *Roger S. Pressman* **108**-1
Development Strategies • The Management Spectrum • Software Project Management • Software Quality Assurance • Software Configuration Management • Summary

109 Software Architecture *Stephen B. Seidman* **109**-1
Introduction • Underlying Principles • Describing Individual Software Architectures • Architecture Description Languages • Describing Architectural Styles

110 Specialized System Development *Osama Eljabiri and Fadi P. Deek* **110**-1
Introduction • Principles of Specialized System Development • Application-Based Specialized Development • Research Issues and Summary

101

Software Qualities and Principles

101.1 Classification of Software Qualities **101**-2
Product and Process Qualities • External vs. Internal Qualities

101.2 Representative Software Qualities **101**-3
Correctness, Reliability, and Robustness • Performance • Usability • Verifiability • Security • Maintainability • Reusability • Portability • Understandability • Interoperability • Productivity • Timeliness • Visibility

101.3 Quality Assurance **101**-13

101.4 Software Engineering Principles in Support of Software Quality **101**-14
Rigor and Formality • Separation of Concerns • Modularity • Abstraction • Anticipation of Change • Generality • Incrementality

101.5 A Case Study in Compiler Construction **101**-21
Rigor and Formality • Separation of Concerns • Modularity • Abstraction • Anticipation of Change • Generality • Incrementality

101.6 Summarizing Remarks **101**-24

Carlo Ghezzi
Politecnico di Milano

Mehdi Jazayeri
Technical University of Vienna

Dino Mandrioli
Politecnico di Milano

The goal of any engineering activity is to build something — an artifact or 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 application or 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 other 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. Even though it is 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 never 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. Software's malleability sometimes leads people to think that it can be changed easily. In practice, it cannot.

We may be able to change code easily with a text editor, but meeting the need for which the change was intended is not necessarily done so easily. Indeed, we must treat software like other engineering products

1-58488-360-X/$0.00+$1.50

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 exploit the malleability property, but we must do so with discipline.

Another characteristic of software is that its creation is *human intensive*. It requires mostly engineering, rather than manufacturing, resources. In most other engineering disciplines, the manufacturing process determines the final cost of the product. Also, the process must be managed closely to ensure that defects are not introduced into the product, for example, through the use of faulty parts. 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 with manufacturing. This process must 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 dictate 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 when a convoy of trucks travels across it. In traditional engineering disciplines, the engineer has tools for describing the qualities of the product distinctly from those of the design of the product. In software engineering, the distinction is not so clear. The functional requirements of the software product are often intermixed in specifications with the qualities of the design.

To achieve the desired qualities, the construction of any nontrivial product must follow sound *design principles*. These principles apply to any engineering discipline, including software engineering. In applying such principles to software engineering, they must be customized to deal with the peculiar characteristics of software.

Software engineering principles deal with both the **process** of software engineering and the final **product**. The right process helps to produce the right product, but the desired product also affects the choice of which process to use.

In this chapter, we first examine the qualities that are relevant to software products and software production processes (Section 101.1 and Section 101.2) and the means to assess them (Section 101.3). In Section 101.4, we present general principles that may be applied throughout the process of software construction and management in order to achieve the desired qualities of software products. Finally, Section 101.5 presents a case study of the use of software engineering principles for compiler development.

Throughout the chapter, we assume that the software product to be developed is large and complex. The application of engineering principles is indispensable in the development of such products. In general, the choice of principles and techniques is determined by the software quality goals. Software for critical applications, where the effects of errors are serious, even disastrous, imposes stricter reliability requirements than noncritical applications.

Principles, however, are not sufficient. In fact, they are general and abstract statements describing desirable properties of software processes and products. To apply principles, the software engineer uses appropriate methods and specific techniques that help to incorporate the desired properties into processes and products. 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. Some important software design methods will be the issue of specific chapters of this Handbook. Many methodologies are supported by software tools.

101.1 Classification of Software Qualities

There are many desirable software qualities. Some 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, predictable, and easy to

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

101.1.1 Product and Process Qualities

We use a process to produce the software product. We distinguish between qualities that apply to products and those that apply to processes. For example, we may want the product to be user-friendly and we may want the process to be efficient. Often, however, process qualities are closely related to product qualities. For example, if the process requires careful planning of test data before any design and development of the system starts, product reliability will increase. Moreover, there are qualities, such as efficiency, that can refer both to the product and the process.

It is useful 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 produces a number of intermediate products in the course of the software process. The customer-visible product consists perhaps of the executable code and the user manual, but the developers produce a number of other artifacts, such as requirements and design documents, test data, etc. We refer to these intermediate products as *work products* or artifacts to distinguish them from the end product delivered to the customer. Work products are often subject to similar quality requirements as the end product. Given the existence of many work products, it is possible to deliver different subsets of them 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.

Process quality has received increasing attention over time, as its impact on product quality has been recognized and observed.

101.1.2 External vs. Internal Qualities

We can divide software qualities into external and internal qualities. *External qualities* are visible to the users of the system; *internal qualities* concern the developers of the system. In general, users of software care only about external qualities, but it is the internal qualities — which deals 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 closely related, and the distinction between internal and external may depend on the user and the delivered product. For instance, a well documented design is usually an internal quality; in some cases, however, users want the delivery of design documentation as an essential part of the product (e.g., for military products). In this case, this quality becomes external.

101.2 Representative Software 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 in Section 101.1. Several software quality models are proposed in the literature. Most of them share the essential aspects but differ in emphasis and organization. ISO 9126 states, "The maturity of the models, terms, and definitions does not yet allow them to be included in a standard" [ISO 9126]. Furthermore, some application areas emphasize certain qualities and may require more specialized qualities. We give some examples of such qualities when we describe specific qualities.

101.2.1 Correctness, Reliability, and Robustness

The terms **correctness**, **reliability**, and **robustness** are related and collectively characterize a quality of software which implies that the application performs its functions as expected. In this section, we define these three terms and discuss their relationships to one another.

101.2.1.1 Correctness

A program is written to provide functions specified in its functional requirements specifications. Often, there are other requirements — such as performance and scalablity — that do not pertain to the functions of the system. We call these kinds of requirements *nonfunctional requirements*. A program is *functionally correct* if it behaves according to its stated functional 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 shall follow this convention when the context is clear.

The definition of correctness assumes that specifications for the system are available and that it is possible to determine unambiguously whether a program meets the specifications. Such specifications rarely exist for most current software systems. If a specification does exist, it is usually written in an informal style using natural language. Therefore, it is likely to contain many ambiguities. Regardless of these difficulties with current specifications, however, the definition of correctness is useful because it captures a desirable goal for software systems.

Correctness 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 shall see in Chapter 113, we may assess the correctness of a program through a variety of methods, some based on an experimental approach (e.g., testing), others based on an analytic approach (e.g., inspections or formal verification of correctness). Correctness can be enhanced by using appropriate tools, such as high-level languages — particularly those supporting extensive static analysis. Likewise, correctness can be improved by using standard proven algorithms or libraries of standard modules, rather than inventing new ones. Finally, correctness can be enhanced by using proven methodologies and processes.

101.2.1.2 Reliability

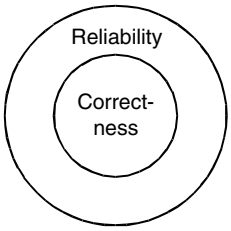
Informally, software is reliable if the user can depend on it; in fact, the attribute *dependable* is also used in such a case. The specialized literature on software reliability, however, defines reliability in terms of statistical behavior — the probability that the software will operate as expected over a specified time interval. According to this definition, reliability is a more specialized — and measurable — quality than the generic term dependability. For the purpose of this chapter, however, the informal definition is sufficient, and we consider the two terms as synonymous.

Correctness is an absolute quality: any deviation from its requirements makes a system incorrect, regardless of how minor or serious the consequence of the deviation is. 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 yet achieved this enviable status of reliability. 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.

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 with a warning not to lean over the railing. In the classic disciplines, design errors are extremely rare and generate news headlines. A collapsed bridge may even cause the designers to be prosecuted in court.

In contrast, software design errors are generally treated as unavoidable. Far from being surprised when we find software errors, we *expect* them. Instead of a guarantee of reliability, with software we get a disclaimer



Au: Figure caption OK?

FIGURE 101.1 Relationship between correctness and reliability in the ideal case.

that the manufacturer is not responsible for any damages due to product errors. Software engineering is still trying to achieve software reliability comparable to the reliability of other products.

Figure 101.1 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 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 101.1 represents an idealized situation wherein the requirements are themselves assumed to be correct; that is, they are a faithful representation of what the implementation must ensure in order to satisfy the needs of the expected users. Often, however, there are insurmountable obstacles to achieving this goal. The upshot is that we sometimes have correct applications 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 situation is discussed in Section 101.2.1.3.

Reliability has several subcomponents that assume importance depending on the application domain. Three qualities that contribute to reliability are fault-tolerance, availability, and safety. *Fault-tolerance* refers to the ability of the software to detect faults in the hardware environment, such as transient power failures, and continue operation in the face of them. *Availability* refers to the property of the system that ensures that the system's services are available to users close to 100% of the time. Thus, a highly available system must not only be able to deal with faults and errors; it also must allow system diagnostic and maintenance procedures to be carried out while the system is in normal operations. Systems such as telephone systems and banking systems must be both fault-tolerant and highly available. High availability and fault tolerance are usually accomplished by distributed architectures that use redundant resources.

In the case of highly critical systems, the term *safety* is often used to denote the absence of undesirable behaviors that can cause system hazards. Safety deals with requirements other than the primary mission of a system and requires that the system execute without causing unacceptable risk. Unlike functional requirements, which describe the intended correct behavior in terms of input–output relationships, safety requirements describe what should *never* happen while the system is executing. In some sense, they are negative requirements: they specify the states the system must never enter. For example, an X-ray medical system must observe the safety property that the radiation it applies never exceeds a certain threshold.

101.2.1.3 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 is not robust. It might be correct, though, if the requirements specification does not state what the program should do in the face of incorrect input. Obviously, robustness is difficult to define; 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 101.1).

Again, an analogy with bridges is instructive. Two bridges connecting two sides of the same river are both correct if each satisfies the stated requirements. If, however, during an unexpected, unprecedented earthquake, 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 resistance to earthquakes as a correctness requirement. In other words, as the phenomenon under study becomes better known, we approach the ideal case shown in Figure 101.1, where specifications capture the expected requirements exactly.

The means to achieve robustness depend on the application area. For example, a system written for novice computer users must be more prepared to deal with ill formatted input than an embedded system that receives its input from a sensor. This, of course, does not imply that embedded systems do not need to be robust. On the contrary, embedded systems often control critical devices and require extra robustness.

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 the requirement out of the specification, it may become an issue of robustness. The border between the two qualities is thus determined by the specification of the system. Finally, reliability comes in because not all incorrect behaviors signify equally serious problems; that is, some incorrect behaviors may be tolerable.

We may also use the terms correctness, robustness, and reliability in relation 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.

101.2.2 Performance

Any engineering product is expected to operate at a certain level of **performance**. Unlike other disciplines, software engineering often equates performance with efficiency, but they are not the same. Efficiency is an internal quality and refers to how economically the software utilizes the resources of the computer. Performance, on the other hand, is an external quality based on user requirements. For example, a customer may specify a performance requirement for a telephone switch to be able to process 10,000 calls per hour. An efficiency requirement for the same switch may state that processing a call must use a minimum amount of memory and take a minimum amount of time. Efficient use of resources often affects the performance of a system. For example, conserving memory may increase the response time of a switch and its ability to process a high number of calls per second.

Performance may affect the usability of the system. If a software system is too slow, it reduces the users' productivity, possibly to the point of not meeting their needs. Performance also affects the scalability of a software system. An algorithm that is quadratic may work on small inputs but not work at all on larger inputs.

There are three basic approaches to evaluating the performance of a system: measurement, analysis, and simulation. We can measure the actual performance of a system by means of hardware and software monitors that collect data while the system is running and thereby allow us to discover bottlenecks in the system. The second approach is to build a model of the product and mathematically analyze it. The third approach is to use the model to simulate the product.

In some application areas, performance criteria are standardized and may be used as the basis for comparing different products. For example, in telephone switching systems, number of calls processed per second and the maximum number of simultaneous calls processed are two common measures. For information systems, number of transactions per second is a common performance measure and is used as the basis for product selection.

In many software development projects, performance is addressed only after the initial version of the product is implemented. At that point, it is very difficult — sometimes even impossible — to achieve

significant improvements in performance without redesigning the software. Instead, even a simple model is useful for predicting a system's performance and guiding design choices so as to minimize the need for redesign.

In some complex projects, in which 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 to resolve such issues as whether a function should be provided by software or by a special-purpose hardware device.

The notion of performance also applies to a development process, in which case we call it *productivity*. Productivity is important enough to be treated as an independent quality and is discussed in Section 101.2.11.

101.2.3 Usability

A software system is usable — or user-friendly— if its human users find it easy to use. This definition reflects the subjective nature of **usability**.

The *user interface* is an important component of user-friendliness. Properties that make an application user-friendly to novices are different from those desired by expert users. For example, a software system that presents the novice user with a graphical interface is friendlier than one that requires the user to enter a set of one-letter commands. On the other hand, experienced users might prefer a set of commands that minimize the number of keystrokes, rather than a fancy graphical interface through which they must navigate to get to the command that they knew all along they wanted to execute.

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, usability 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 and predictability of its user and operator interfaces. Clearly, however, the other qualities mentioned — 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 a beautiful color.

Usability is also discussed under the subject of *human factors*. Human factors and usability engineering play a major role in many engineering disciplines. For example, automobile manufacturers devote significant effort to deciding the positions 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 after extensive study of user needs and attitudes by specialists in fields such as industrial design or psychology.

Interestingly, ease of use in many engineering disciplines is achieved through standardization of the human interface. Once a user knows how to use one television set, that user can operate almost any other television set. There is a clear trend in software applications toward more uniform and standard user interfaces, as seen, for example, in Web browsers. There are also usability labs that attempt to measure the usability of software products.

101.2.4 Verifiability

A software system is verifiable if its properties can be verified easily. For example, it is important to be able to verify the correctness or the performance of a software system. Verification can be performed by formal and informal analysis methods or through testing.

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.

101.2.5 Security

A system is secure if it provides its services only to its authorized users and protects the rights and information of those users. Although **security** is important in any system, it has gained importance as applications have become increasingly distributed and offered over public networks. Security is an important quality of information systems, in which users trust the safeguarding of their data to the system. For example, a banking system that maintains customer account data and provides access to that data through the Internet is required to be secure. Two qualities related to security are data integrity and privacy. *Data integrity* ensures that once the user data is committed to the system, it will not be modified or destroyed through system malfunction or unintended or malicious acts of other users. *Privacy* ensures that user transactions and user data are protected from unauthorized users and will be not be used for unauthorized purposes. For example, credit card numbers entered into an electronic commerce system must be used only for the purpose of the transaction for which they were entered.

The combination of security and verifiability enables security properties to be verified. Such a combination is relevant in financial and military systems.

101.2.6 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 that 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 captures the essence of this process better is *software evolution*. Second, in other engineering products, such as computer hardware, 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, the term is used widely, and we will continue using it here.

There is evidence that maintenance costs exceed 60 percent 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.

Corrective maintenance has to do with the removal of residual errors that are 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% of maintenance costs.

Adaptive maintenance, which accounts for nearly another 20% of maintenance costs, 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).

Finally, *perfective maintenance*, which absorbs over 50% of maintenance costs, 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.

The term *legacy software* refers to software that already exists in an organization and usually embodies much of the organization's processes and knowledge. Such software holds considerable value for the organization, represents past investments, and may not be replaced easily. On the other hand, because of its age, it is usually written in older languages and uses older software engineering technology. Legacy software is, therefore, difficult to maintain. For example, an old personnel system may embody an organization's operational procedures and personnel policies. Such legacy systems represent a challenge to

software evolution. *Reverse engineering* and *reengineering* techniques and technologies aim at uncovering the structure of legacy software and restructuring or in some way improving it.

Maintainability can be seen as two separate qualities: **reparability** and **evolvability**. Software is reparable if it allows the fixing of defects; it is evolvable if it allows changes that enable it to satisfy new or modified requirements.

The distinction between reparability and evolvability is not always clear. For example, if the requirements specifications are vague, it may not be clear whether a change is made to fix a defect or to satisfy a new requirement. In general, however, the distinction between the two qualities is useful.

Both reparability and evolvability are improved by suitable modularization in the software structure. As we shall see later, the right modularization may help to locate errors more easily. It may also help to encapsulate the changeable parts in a separate module, making it easier to apply changes.

101.2.7 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 the product — perhaps with minor changes — to build another product. Reusability may be applied at different levels of granularity — from whole applications to individual routines — but it appears to be more applicable to software components than to whole products.

A good example of a reusable product is the UNIX shell, which 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.

Numeric libraries were the first examples of reusable components. Several large FORTRAN libraries, now rewritten in other languages, have existed for many years. Users buy these libraries and use them to build their own products, without having to reinvent or recode well known algorithms. Several companies are devoted to producing just such libraries. Nowadays, reusable libraries exist for different areas, such as graphical user interfaces, simulation, etc. One of the goals of reusability researchers is to increase the granularity of components that may be reused. One of the goals of object-oriented programming (see Chapter 91) is to achieve both reusability and evolvability.

Reusability is difficult to achieve *a posteriori*. Reusable components must be designed with reusability as a primary design goal. Reusable components are abstractions of useful concepts, are general, and have clear and usable interfaces.

Languages that support generic components, such as C++ and Ada, enable higher levels of reusability in software components. Most C++ libraries commonly consist of template modules. The field of component-based software engineering (CBSE) has the goal of building applications by assembling a set of ready-made, reusable, off-the-shelf components. (See Chapter 117 on component-based computing.) Object oriented languages, such as C++ and Java, help in unifying the qualities of evolvability and reusability by extensive use of interfaces, inheritance, and polymorphism.

Reusability has broader applicability than just code. It may occur at different levels and may affect both product and process. In general, any of the artifacts of the software process, such as the requirements specification, may be reused. For example, 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. Clearly, the more modularly designed the work products are, the more likely it is that they, or parts of them, may be reused in the future.

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. Life-cycle models are also attempts at reusing higher-level processes.

Reusability of standard parts characterizes the maturity of an industrial field. We see high degrees of reuse in such mature areas as the automobile industry and consumer electronics. For example, a car is constructed by assembling many components that are highly standardized and used across many models produced by the same industry. Certainly, the designs are routinely reused from model to model. Finally, the manufacturing process is often reused. The level of reuse is increasing in software, but it still is short of that of other established engineering disciplines.

101.2.8 Portability

Software is portable if it can run in different environments. The term *environment* may refer to a hardware platform or a software environment, such as a particular operating system. **Portability** is economically important because it helps amortize the investment in the software system across different environments and different generations of the same environment. Many applications are independent of the actual hardware platform, because the operating system provides portability across hardware platforms. These days, the applications' dependencies are on operating systems and other software systems, such as databases and user interface systems. Portability may be achieved by modularizing the software so that dependencies on the environment are isolated in only a few, well designated modules. To port the software to a new environment, only these environment-dependent modules need to be modified. With the proliferation of networked systems, portability has taken on new importance because the execution environment is naturally heterogeneous, consisting of many different kinds of computers and operating systems. In addition, the delivery devices have become diverse. For example, Internet browsers must be able to run not only on workstations and personal computers, but also on palmtops and even mobile phones.

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 particular machine. Even in these cases, however, it is possible to achieve some level of portability. UNIX and its variant, Linux, are examples of an operating system that has been ported to many different hardware systems. Of course, the porting effort may require 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.

101.2.9 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. We can follow certain guidelines to produce more understandable designs and to write more understandable programs. Systematic documentation of both the design and the program is clearly very important. Furthermore, abstraction and modularity enhance a system's understandability.

The activity of software maintenance is dominated by the subactivity of *program understanding*. Maintenance engineers spend most of their time trying to uncover the logic of the application and a smaller portion of their time applying changes to the application.

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 a system understandable if it has predictable behavior. External understandability is a factor in a product's usability.

101.2.10 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 graphics package, 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. Interoperability can be seen as reusability at the application level.

Interoperability abounds in other engineering products. For example, stereo systems from various manufacturers work together and can be connected to television sets and video recorders. In fact, stereo systems produced decades ago accommodate new technologies such as compact discs! In contrast, early operating systems had to be modified — sometimes significantly — before they could work with new devices. The generation of plug-and-play operating systems attempts to solve this problem by automatically detecting and working with new devices.

The UNIX environment, with its standard interfaces, offers a limited example of interoperability within a single environment. UNIX encourages software engineers to design applications so that they have a simple, standard interface, which allows the output of one application to be used as the input to another. The UNIX standard interface is a primitive, character-oriented one. It falls short when one application needs to use structured data — say, a spreadsheet or an image — produced by another application.

With interoperability, a vendor can produce different products and allow the user to combine them if necessary. This makes it easier for the vendor to produce the products, and it gives the user more freedom in exactly what functions to pay for and to combine. Interoperability can be achieved through standardization of interfaces. An example of such interoperability is the Web browser application that provides plug-in interfaces for different applications. For example, a new audio player provided by one vendor may be added to the browser provided by another vendor.

A concept related to interoperability is that of an *open system* — an extensible collection of independently written applications that 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 application developer can then take advantage of these interfaces, some of which 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 (social) 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 efforts at standardization in this area. For example, the CORBA standard defines interfaces that support the development of components that may be used in open distributed systems.

101.2.11 Productivity

Productivity is a quality of the software production process, referring to its efficiency and performance. 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. Management tries to organize team members and adopt processes in such manner as to capitalize on the individual productivity of the 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 high productivity in producing a certain product, but not in producing a variety of products. Software reuse is a technique that increases the overall productivity of an organization in producing a collection of products, but the cost of developing reusable modules can be amortized only over many products.

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, for that matter — if we are to have any hope of comparing different processes in terms of their productivity. Early metrics, such as the number of lines of code produced, have many shortcomings.

As with other engineering disciplines, the efficiency of the process is affected strongly by automation. Many modern software engineering tools and environments help in achieving increases in productivity.

101.2.12 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. Today, due to increased competitive market pressures, software projects face even more stringent time-to-market challenges. Being late may sometimes preclude market opportunities. Although on-time delivery of a product that is lacking in other qualities, such as reliability or performance, may be pointless, some argue that the early delivery of a preliminary and still unstable version of a product may favor the later acceptance of the final product. The Internet has facilitated this approach. Vendors can place early versions of products on the Internet, enabling potential users to try the product, providing feedback to the vendor.

Timeliness requires careful scheduling, accurate estimation of work, and clearly specified and verifiable milestones. All engineering disciplines use standard project management techniques to achieve timeliness. They are sometimes difficult to apply in software engineering because of its human intensive nature. There are no objective or standard ways of defining, predicting, and measuring the amount of work required to produce a given piece of software, the productivity of software engineers, and the milestones in the development process.

One technique for achieving timeliness is through the *incremental delivery* of the product. Progressively larger subsets of the product are developed and delivered as new increments at each stage. Each increment provides additional functionality and becomes closer to the final product. Obviously, incremental delivery depends on the ability to break down the set of required system functions into subsets that can be delivered in increments. Incremental delivery allows (parts of) the product to become available earlier; and the use of the early increments helps in refining the requirements incrementally.

The biggest challenge to achieving timeliness is to ensure that other qualities, those of both product and process, are not jeopardized by focusing only on timeliness.

101.2.13 Visibility

A software development process has **visibility** if all of its steps and its current status are documented clearly. Another term used to characterize this property, also used in business processes and organizations, is *transparency*. 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, in 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 in opposing directions. The most common example of the latter situation is when the integration group has been testing a version of the software, assuming that the next version will involve fixing defects, while the engineering group decides to do a major redesign to add some functionality. This tension, created when one group is trying to stabilize the software while another group is destabilizing it, is common. The process must encourage a consistent view of the status and current goals among all participants.

One of the benefits of the *synch-and-stabilize* software build process, popularized by Microsoft, is that it adds some visibility to the process. In this approach, the software product is rebuilt (integrated) each day, exposing problems in components and their interactions as soon as they occur. These problems may be symptoms of deeper problems, but the process helps to uncover them early.

Visibility is not only an internal quality; it is also external. During the course of a long project, many requests arise about the status of the project. 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.

A difficulty in managing large projects is dealing with personnel turnover. With 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 often reduces the productivity of the whole project, as the folklore is being transferred slowly from the existing crew of engineers to the new engineers.

The preceding discussion points out that visibility of the process requires not only that all of its 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, as well. In other words, it must be understandable (see Section 101.2.9).

101.3 Quality Assurance

Once we have decided on the qualities that are the goals of software engineering, we need principles and techniques to help us achieve them. We also need to be able to measure a given quality. In software organizations, this activity is called **quality assurance**.

If we identify a quality as important, we must be ready to measure it to determine how well we are achieving it. This, in turn, requires that we define each quality precisely, so that it is clear what we should be measuring. Without measurements, any claims of improvement are without basis. But without defining a quality precisely, there is no hope that we can measure it precisely — let alone quantitatively.

The established engineering disciplines have standard techniques for measuring quality. For example, the reliability of many artifacts is commonly characterized by mean time to failure (MTTF), that is, the average time between two consecutive failures of the product. Although some software qualities, such as performance, are measured relatively easily, most software qualities unfortunately have no universally accepted metrics. For example, whether a given system will evolve more easily than another is usually determined subjectively. Nevertheless, metrics are needed, and indeed, much research work is currently under way for defining objective metrics.

The current state of practice of quality assurance is a collection of verification and monitoring methods. Some of them are based on objective evaluations, such as testing, and others are based on informal procedures, such as walkthroughs; a few, more ambitious methods aim at exploiting mathematical analysis during software verification. Chapters 111, 113, and 114 relate, to different extents and from different points of view, to the quality assurance problem.

Quality assurance for software processes has taken the form of process assessment procedures whose aim is to measure the ability of software organizations and their processes. For example, the *capability maturity model* (CMM) classifies software development organizations on the basis of detailed evaluation of their processes. The model defines five levels of maturity which may be used to characterize the software process quality of an organization, referred to as its *maturity level*. The maturity level of an organization is intended to reflect the ability of the organization to predict its costs and schedules. The levels are as follows:

1. **Initial** — The organization has no statistical control over its processes and no predictability.
2. **Repeatable** — The organization has a stable, repeatable process, supported by rigorous project management controls.
3. **Defined** — The organization has defined a base process that is applied consistently for managing different projects.
4. **Managed** — The process is systematically monitored and its performance measured with appropriate metrics.
5. **Optimizing** — The process measurements are used for continuous improvement of the process.

It turns out that most organizations fall into maturity levels 2 and 3. The CMM is being used by organizations to document their competence and by customers to qualify their vendors. Although the particular way of measuring process maturity and the appropriate metrics to be used are sometimes controversial, the impact of process quality on product quality is generally accepted to be significant.

An indispensable technology that supports disciplined processes and is a primary factor that differentiates mature from immature organizations is *configuration management,* which is concerned with controlled change management. In the software production process, configuration management is concerned with maintaining and controlling the relationship between all the work products of the various versions of a product. Configuration management tools allow the maintenance of families of products and their components. They help in controlling and managing changes to work products.

101.4 Software Engineering Principles in Support of Software Quality

So far, we have discussed a number of important software qualities. How can we achieve these qualities? In this section, we discuss seven general principles that help in achieving software quality. These principles may be applied throughout the software development process and are not limited to a particular phase of the process. The principles deal with rigor and formality, separation of concerns, modularity, abstraction, anticipation of change, generality, and incrementality. By its very nature, the list cannot be exhaustive, but it does cover the important areas of software engineering. The principles are, of course, strongly related and together form a set of guidelines for the engineer to follow.

101.4.1 Rigor and Formality

Software development is a creative activity. In any creative process, there is an inherent tendency to be neither precise nor accurate, but to follow the inspiration of the moment in an unstructured manner. *Rigor* — defined as precision and exactness — on the other hand, is a necessary complement to creativity in every engineering activity. It is only through a rigorous approach that we can repeatedly produce reliable products, control their costs, and increase our confidence in their reliability. Rigor need not constrain creativity. Rather, it can be used as a tool to enhance creativity. The engineer can be more confident of the results of a creative process after performing a rigorous assessment of those results.

Paradoxically, rigor is an intuitive principle 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 and precise 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 small bridge, to be used temporarily to connect the two sides of a creek. If the bridge were a large and permanent one, on the other hand, the engineer would instead use a mathematical model to verify whether the design was safe. An exceptionally long bridge or one built in an area of much seismic activity would require a more sophisticated mathematical model. In that case, the mathematical model would take into account 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. Textbooks on functional calculus 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; 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, in which the validity of some intermediate deduction is unclear, the mathematician may try to formalize the informal reasoning to assess its validity.

These examples show that the engineer (and the mathematician) must be able to identify and understand the level of rigor and formality that should be achieved, depending on the conceptual difficulty and criticality of the task. The level may even vary for different parts of the same system. For example, critical parts — such as the scheduler of a real-time operating systems kernel or the security component of an electronic commerce system — may merit a formal description of their intended functions and a formal approach to their assessment. Well understood and standard parts would require simpler approaches.

If we examine this issue in the context of software specifications, we see, for example, 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 instance, one may hope to use the formal description of the program to derive 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 in which 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 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 111 shows these concepts in action in the case of software specifications. Chapter 107 does the same for software verification. Chapter 106 is about formal methods.

Rigor and formality also apply to software processes. Rigorous documentation of a software process helps in reusing the process in other, similar projects. On the basis of such documentation, managers may foresee the steps through which a new project will evolve, assign appropriate resources as needed, etc. Similarly, rigorous documentation of the software process may help to maintain an existing product. If the various steps through which a project evolved are documented, one can modify an existing product, starting from the appropriate intermediate level of its derivation, not the final code. Finally, if the software process is specified rigorously, managers may monitor it accurately, in order to assess its timeliness and improve productivity.

101.4.2 Separation of Concerns

Separation of concerns allows us to deal with different aspects of a problem to dominate its complexity, so that we can concentrate on each aspect individually. Separation of concerns is a commonsense practice that we try to follow in our everyday lives to overcome the difficulties we encounter. The principle should also be applied to 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: functions to offer, expected reliability, efficiency with respect to space and time, the product's relationship with the environment (i.e., the special hardware or software resources required), user interfaces, etc. Others concern the development process: the development environment, the organization and structure of teams, 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). This, in turn, may affect the policy for error recovery. When different design decisions are strongly interconnected, it would be useful to take all the issues into account at the same time and by the same people, but this is not usually possible in practice.

There are various ways in which concerns may be separated. First, one can separate them in *time*. Temporal separation of concerns allows for the precise planning of activities and eliminates overhead that would arise through switching from one activity to another in an unconstrained way. In fact, it is the underlying motivation of the software life-cycle models, each of which defines a sequence of activities that should be followed in software production (see Chapter 110).

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 *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 113 and Chapter 114.

Another important type of separation of concerns allows different 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 flow of data from one activity to another in the system and the flow of control that governs how 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 parts of the same system separately; here, separation is in terms of size. This is a fundamental concept that we must master to dominate the complexity of software production. Indeed, it is so important that we prefer to detail it shortly as a separate point under modularity (see Section 101.4.3).

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. While this is true in principle, our ability to make optimized decisions in the face of complexity is rather limited. If we consider too many concerns simultaneously, we are likely to be overwhelmed by the amount of detail and complexity we face. Some of the most important decisions in design concern which aspects to consider together and which separately.

Perhaps the most important application of separation of concerns is to separate problem-domain concerns from implementation-domain concerns. Problem-domain properties hold in general, regardless of the implementation environment. For example, in designing a personnel-management system, we must separate issues that are true about employees in general from those which are a consequence of our implementation of the employee as a data structure or object. In the problem domain, we may speak of the relationship between employees, such as "employee A reports to employee B"; in the implementation domain we may speak of one object pointing to another. These concerns, unfortunately, are often intermingled in many projects.

As a final remark, we 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 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. Or, 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.

101.4.3 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)
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 executed in sequence, first concentrating on modules and then on their composition, then we say that the system is designed from the *bottom up*. The converse, when we decompose the system into modules first and then concentrate on individual module design, is *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. Although modularity is often discussed in the context of software design, it is not only a desirable design principle; it permeates the whole of software production. In particular, modularity provides four main benefits in practice:

The capability of decomposing a complex system into simpler pieces
The capability of composing a complex system from existing modules
The capability of understanding the system in terms of its pieces
The capability of modifying a system by modifying only a small number of its 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 then conquer them individually.

The *composability* of a system is based on starting from the bottom up with elementary components and combining them in steps toward finally producing a finished system. As an example, a system for office automation may be designed by assembling existing hardware components, such as personal workstations, a network, and peripherals; system software, such as the operating system; and productivity tools, such as document processors, databases, and spreadsheets.

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 with another that performs the same function, but differs in computational resource requirements.

Modularity supports the capability of understanding and modifying a system. If the entire system can be understood only in its entirety, modifications are likely to be difficult to apply, and the result will probably be unreliable. When it is necessary to repair a defect or enhance a feature, proper modularity helps to confine the search for the fault or enhancement to single components. Modularity thus forms the basis for software evolution.

To achieve modular composability, decomposability, understandability, and modifiability, the software engineer must design the modules with the goal of 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 a property about the internal structure 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

Au: Figure caption OK?

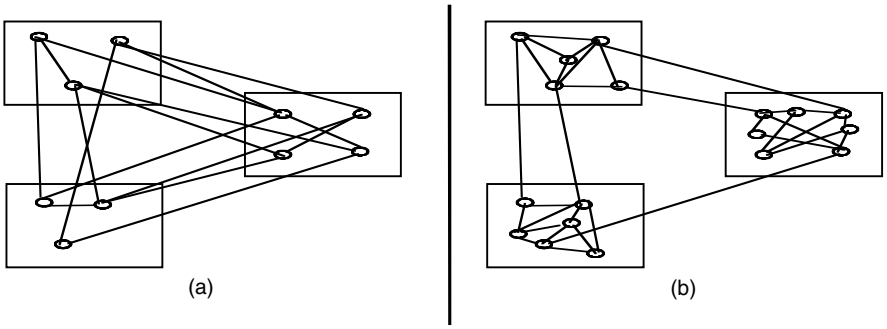


FIGURE 101.2 Graphical description of cohesion and coupling (a) A highly coupled structure. (b) A structure with high cohesion and low coupling.

modules depend on each other heavily, they have high coupling. Ideally, we would like modules in a system to exhibit *low coupling*, because it will then be possible to analyze, understand, modify, test, and reuse them separately. Figure 101.2 provides a graphical view of cohesion and coupling. The practical design guideline derived from the high-cohesion/low-coupling rule is that two units should be put in the same module if they are tightly dependent on each other; otherwise they should be placed in different modules. Indeed, this is achieved by object-oriented programming, which groups both the data and the routines that manipulate them in a single module.

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. In other words, modularity supports the application of the principle of separation of concerns.

101.4.4 Abstraction

Abstraction is a fundamental principle for understanding, designing, and analyzing complex problems. In applying abstraction, 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 less important 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 digital 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 modes of functioning 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 watch and performing 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.

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 a set of equations, is an idealized abstraction of a device. The equations are a simplified model that approximates the behavior of the real components. The equations often ignore details, such as the fact that there are no "pure" connectors between components and that connectors should also be modeled in terms of resistors, capacitors, etc. The designer ignores both of these facts, because the effects they describe are negligible in terms of the observed results.

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 we believe are relevant. The same holds true 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 shown in Chapter 111, 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 specific computer's addressing mechanism. This helps us to concentrate on the solution to the problem we are trying to solve, rather than on 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 of the manual procedure it replaces. It provides the *essence* of the manual procedure, not its exact details. When applied judiciously in design and programming, abstraction affects all software qualities in the product. For example, proper abstraction leads to modularity, which aids in achieving maintainability and reusability.

Abstraction is an important principle that applies to both software products and software processes. For example, the comments that we often use 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 use of the procedure by the other parts of the program.

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 is to identify some key factors of the new system — for example, the number of engineers on the project and the expected size of the final system — and to extrapolate from the cost profiles of previous similar systems. The key factors used to perform the analysis are an abstraction of the system for the purpose of cost estimation.

101.4.5 Anticipation of Change

Anticipation of change is perhaps the one principle that distinguishes software the most from other types of industrial productions. In fact, software undergoes changes constantly, and anticipation of change is a principle that we can use to achieve evolvability.

The ability of software to evolve does not happen by accident or sheer luck — it requires a special effort to anticipate how and where changes are likely to occur. Designers should try to identify likely future changes and take special care to make these changes easy to apply. Software should 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 into 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. In other words, anticipation of change should be the basis for our modularization strategy. (See Chapter 103 on software design).

In many cases, a software application is developed before its requirements are completely understood. Then, after being released, on the basis of feedback from the users, 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 known initially.

Anticipation of change also affects the management of the software process. For example, managers should anticipate the effects of personnel turnover. Also, when the life cycle of an application is designed, it is important to take maintenance into account. Depending on the anticipated changes, managers must

estimate costs and design the organizational structure that will support the evolution of the software. Finally, managers should decide whether it is worthwhile to invest time and effort in the production of reusable components, either as a by-product of a given software development project or as a parallel development effort.

101.4.6 Generality

The principle of generality may be stated as follows:

Every time you are asked to solve a problem, try to focus on the discovery of a more general problem that may be hidden behind the problem at hand. It may happen that the generalized problem is not more complex — indeed, it may even be simpler — than the original problem. Moreover, it is likely that the solution to the generalized problem has more potential for being reused. It may even happen that the solution is already provided by some off-the-shelf package. Also, it may happen that, by generalizing a problem, you end up designing a module that is invoked at more than one point of the application, rather than having several specialized solutions.

A generalized solution may of course be more costly, in terms of speed of execution, memory requirements, or development time, than the specialized solution that is tailored to the original problem. Thus, it is necessary to evaluate the trade-offs of generality with respect to cost and efficiency, in order to decide whether it is worthwhile to solve the generalized problem instead of the original problem.

Generality is a fundamental principle that allows us to develop software components for the market. Such general-purpose, off-the-shelf products represent a rather general trend in software. For every specific application area, general packages that provide standard solutions to common problems are increasingly available. If the problem at hand may be restated as an instance of a problem solved by a general package, it may be convenient to adopt the package instead of implementing a specialized solution. For example, we may use macros to specialize a spreadsheet application to be used as an expense-report application.

101.4.7 Incrementality

Incrementality characterizes a process that proceeds in a stepwise fashion, in *increments*. We try to achieve the desired goal by successively closer approximations to it. Each approximation is an increment over the previous one.

Incrementality applies to many engineering activities. When applied to software, it means that the desired application is produced as a result of an evolutionary process.

One way of applying the incrementality principle is to identify useful *early subsets* of an application that may be developed and delivered to customers, in order to get *early feedback*. This allows the application to evolve in a controlled manner in cases where the initial requirements are not stable or fully understood. The motivation for incrementality is that in most practical cases there is no way of getting all the requirements right before an application is developed. Rather, requirements emerge as the application — or some part of it — is available for practical experimentation. Consequently, the sooner we can receive feedback from the customer concerning the usefulness of the application, the easier it is to incorporate the required changes into the product. Thus, incrementality is intertwined with anticipation of change and is one of the cornerstones upon which evolvability may be based.

Incrementality applies to many of the software qualities discussed in the early part of this chapter. We may progressively add functions to the application being developed, starting from a kernel of functions that would still make the system useful, though incomplete. For example, in a business automation system, some functions would still be done manually, whereas others would be done automatically by the application.

We can also add performance in an incremental fashion. That is, the initial version of the application might emphasize user interfaces and reliability more than performance, and successive releases would then improve space and time efficiency.

When an application is developed incrementally, intermediate stages may constitute prototypes of the end product; that is, they are just an approximation of it. The idea of rapid **prototyping** is often advocated as a way of progressively developing an application hand in hand with an understanding of its requirements.

Obviously, a software life cycle based on prototyping is rather different from the typical waterfall model described earlier, wherein we first do a complete requirements analysis and specification and then start developing the application. Instead, prototyping is based on a more flexible and iterative development model. This difference affects not only the technical aspects of projects, but also the organizational and managerial issues. The unified process, presented in Chapter 110, is based on **incremental development**.

Evolutionary software development requires special care in the management of documents, programs, test data, etc., developed for the various versions of software. Each meaningful incremental step must be recorded, documentation must be easily retrieved, changes must be applied in a controlled way, and so on. If these are not done carefully, an intended evolutionary development may quickly turn into undisciplined software development, and all the potential advantages of evolvability will be lost.

101.5 A Case Study in Compiler Construction

In this section, we will show the application of the principles we have just presented to the practical case of compiler construction.

101.5.1 Rigor and Formality

There are many reasons that compiler designers should be rigorous and, possibly, formal. First, a compiler is a critical product. A compiler that generates incorrect code is as serious a problem as a processor that executes an instruction incorrectly. An incorrect compiler can generate incorrect applications, regardless of the quality of the application itself. Second, when a compiler is used to generate code for mass-produced software, such as databases or word processors, the effect of an error in the compiler is multiplied on a mass scale. Thus, in general, it is important to approach the development of a compiler rigorously, with the aim of producing a high-quality compiler.

Compiler construction is one of the fields in computer science where formality has been exploited well for a long time. In fact, formal languages and automata theory were largely motivated by the need for making compiler construction more effective and reliable. Nowadays, the syntax of programming languages is formally defined through Backus–Naur form (BNF) or an equivalent formalism. It is not by chance that, most often, problems associated with compiler correctness are related to the semantic aspects of the language, which are usually defined informally, rather than the syntactic aspects, which are well defined by BNF.

101.5.2 Separation of Concerns

As with most nontrivial engineering artifacts, the construction of a compiler involves several concerns. Correctness (i.e., producing an object code consistent with the source code and producing appropriate error messages in the case of erroneous source programs) is, as usual, a primary concern. Other important issues are efficiency and user-friendliness. Efficiency could be related to compile time (in which case it amounts to performing source code analysis and translation quickly or using little memory) or to run time (in which case it involves producing an object code that is itself efficient). User-friendliness also has several aspects, ranging from the precision, thoroughness, and helpfulness of the diagnostics to the ease of interacting with the human–computer interface (e.g., through well designed windows and other graphical aids).

These and other aspects of the compiler should be analyzed separately, as far as possible. For instance, there is no reason to worry about diagnostic messages while one is designing a sophisticated algorithm to optimize register allocation. This is not to say, as we already noted in general, that different concerns do not affect each other. Typically, in an attempt to make object code as efficient as possible, we might incorrectly overload some register. Also, attempts to produce good run-time diagnostics (e.g., checking that array indexes are within their bounds) may produce run-time inefficiencies.

Run-time diagnostics and efficiency are a typical example of when separation of concerns can and should be applied while keeping in mind the mutual dependencies between the different aspects. In this

case, in fact, the two concerns are often well separated by offering the user the option of enabling or disabling run-time checks. During the development and verification phases, when correctness is still being established and is a major concern, the user turns on run-time checks, making diagnostics the prevailing concern for the compiler. Once the program has been thoroughly checked, efficiency becomes the major concern for its user and, therefore, for the compiler, too; thus, the user could turn off the generation of run-time checks by the compiler.

101.5.3 Modularity

A compiler can be modularized in several ways. Here, we propose a fairly simplistic and traditional modularization based on the several "passes" performed by the compiler on the source code. Such a modular structure should be good enough for our initial purposes. According to this scheme, each pass performs a partial translation from an intermediate representation to another one, the last pass transforming the final intermediate representation to the object code.

The following are the usual compiler phases:

Lexical analysis — Analyzes program identifiers, replaces them with an internal representation, and builds a symbol table with their description. It also produces a first set of diagnostic messages if the source code contains lexical errors (e.g., ill formed identifiers).

Syntax analysis or parsing — Takes the output of the lexical analysis and builds a syntax tree, describing the syntactic structure of the original code. It also produces a second set of diagnostic messages related to the syntactic structure of the program (e.g., missing parentheses).

Code generation — Produces the object code. This last phase is usually done in several steps. For example, a machine-independent intermediate code is produced as a first step, followed by a translation into machine-oriented object code. Each of these partial translations may include an optimizing phase that rearranges the code to make it more efficient.

The foregoing description suggests a corresponding modular description of the structure of the compiler, depicted graphically in Figure 101.3a.

Despite the oversimplification present in the figure, we can already derive a few distinguishing features of modular design:

- System *modules* can be drawn naturally as boxes.
- Module *interfaces* can be drawn as directed lines connecting the boxes representing the modules.

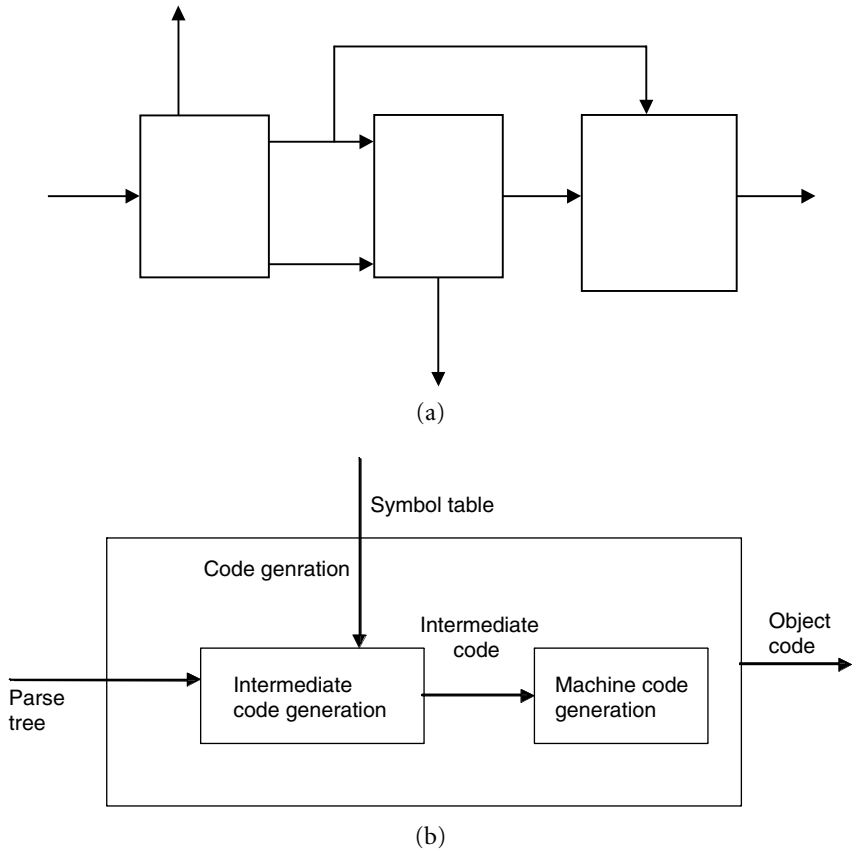
An interface is an item that somehow connects different modules; it represents anything that is communicated or shared by them. Notice that the graphical metaphor suggests that everything that is inside a box is hidden from the outside; modules interact with each other exclusively through interfaces. In the figure, it is convenient to represent interfaces with arrows to emphasize the fact that the item they describe is the output of some module and the input to another one. In other cases, the notion of an interface may be more symmetric (e.g., a shared data structure); in such cases, it is more convenient to represent the item with an *un*directed line.

Notice also that the lines representing the source code, the diagnostic messages, and the object code are the input or output of the whole "system." They are, therefore, drawn without source and target, respectively.

The modular structure of Figure 101.3a lends itself to a natural iteration of the decomposition process. For instance, according to the description of the code-generation phase, the box representing this pass can be refined as suggested in Figure 101.3b.

101.5.4 Abstraction

Abstraction can be applied in compiler design along several directions. From a syntactic point of view, it is fairly typical to distinguish between concrete and abstract syntax. Abstract syntax aims at focusing on the essential features of language constructs, neglecting details that do not affect a program's structure.



Au: Figure Caption OK?

FIGURE 101.3 (a) The modular structure of a compiler. (b) A further modularization of the code-generation module.

For instance, a conditional statement consists of a condition, a statement to be executed if the condition holds and, possibly, a statement to be executed if the condition does not hold. This description remains valid both if we include the keyword *then* before the positive statement, as it happens in Pascal, and if we do not, as it happens in C. Similar remarks apply to the use of the C-like pair {,} and of the Algol-like pair *begin-end* to bracket sequences of statements.

Another typical abstraction is often applied with respect to the target code: as we saw in the previous section, the first phase of code generation produces an intermediate code, which can be viewed as the code for an *abstract machine*. The second phase then translates the code of this abstract machine into code for the concrete target machine. In this way, a major part of the compiler construction abstracts away from the peculiarities of the particular processor that must run the object code. The Java language, indeed, defines a Java Virtual Machine, whose code (Java bytecode) can be executed by interpreting it on different concrete machines.

101.5.5 Anticipation of Change

Several changes may occur during the lifetime of a compiler:

New releases of the target processors may become available with new, more powerful, instructions.
New input–output (I/O) devices may be introduced, requiring new types of I/O statements.
Standardization committees may define changes and extensions to the source language.

The design of the compiler should anticipate such changes. For instance, the Pascal language tried to "freeze" I/O statements within a rigid language definition. This decision conflicted with typical machine

dependencies, and the result was often a number of dialects of the same language, differing mainly in the I/O part. Later, it was recognized that attempts to freeze language I/O were not effective. Thus, languages such as C and Java encapsulated I/O into standardized libraries, reducing the amount of work to be redone whenever I/O changes occurred.

Also, the more likely it is to want to adapt the compiler to different target machines, the higher are the benefits of separating the code-generation phase into two subphases, as we showed previously.

101.5.6 Generality

Like abstraction, generality can be pursued along several dimensions in compiler construction, depending on the overall goals of the project (e.g., producing a fairly wide family of products, as opposed to a highly specialized compiler).

It is useful to be parametric with respect to the target machine. The case of Java's bytecode is a striking example of general design and its benefits. In fact, bytecode is also independent of the source language, allowing it to be used as the target for compilers of languages other than Java. Generality (with respect to target machines) is therefore achieved via abstraction of individual machine architectures into just one virtual machine.

Sometimes, a compiler can be parametric even with respect to the source language. A fairly extreme example of such a generality is provided by so-called *compiler compilers*, which take as input the definition of the source — and possibly of the target — language and automatically produces a compiler translating the source language into the target one. Perhaps the most successful and widely known example of a compiler compiler is provided by the Unix Lex and Yacc programs, which are used to produce automatically the lexical and syntactic modules of a compiler.

Such generality can be achieved thanks to the formalization of the syntax of the language. Thus, the generality principle is exploited in conjunction with formality. Another fairly obvious relation exists between the principles of generality and design for change: we usually want to be parametric — general — with respect to those features which are most likely to change.

101.5.7 Incrementality

Incrementality, too, can be pursued in several ways. For instance, we can first deliver a kernel version of a compiler that recognizes only a subset of the source language and then follow that with subsequent releases that recognize increasingly larger subsets of the language. Alternatively, the initial release could offer just the very essentials: translation into a correct object code and a minimum of diagnostics. Then, we can add more diagnostics and better optimizations in further releases. The systematic use of libraries offers another natural way to exploit incrementality. It is quite common that the first release of a new compiler includes a very minimum of such libraries (e.g., for I/O and memory management); later, new or more powerful libraries are released (e.g., graphical and mathematical libraries).

101.6 Summarizing Remarks

In this chapter, we have discussed important software engineering qualities and principles. Qualities are the ultimate goal we want to achieve; principles provide the basis for concrete means to reach the goal.

Because of their general applicability, we have presented the principles separately, as the cornerstones of software engineering, rather than in the context of any specific phase of the software life cycle. We used a case study of compiler construction to show the application of the general principles. We emphasized the role of general principles without presenting specific methods, techniques, or tools. The reason is that software engineering — like any other branch of engineering — must be based on a sound set of principles. In turn, principles are the basis for the set of methods used in the discipline and for the specific techniques and tools used in everyday life.

As technology evolves, software engineering tools will evolve. As our knowledge about software engineering increases, methods and techniques will evolve, too — though less rapidly than tools. Principles, on the other hand, will remain more stable; they constitute the foundation upon which all the rest may be built. They form the basis for the concepts discussed in the remainder of this Handbook. For instance, Chapter 104 presents object-oriented design, a popular methodology that stresses the qualities of reusability and evolvability, as well as the principles of modularity, anticipation of change, generality, and incrementality.

Defining Terms

Cohesion: Property of a modular software system that measures the logical coherence of a module.

Correctness: Software is (functionally) correct if it behaves according to the specification of the functions it should provide.

Coupling: Property of a modular software system that measures the amount of mutual dependence among modules.

Evolvability: Ease of software evolution.

Incremental development: A software process that proceeds by producing progressively larger subsets of the desired product by delivering new increments at each stage. Each increment provides additional functionality and brings the currently available subset closer to the desired one.

Interoperability: Ability of a software system to coexist and cooperate with other systems.

Maintainability: Ease of maintaining software. It can be further decomposed into evolvability and reparability.

Methodology: A combination of methods and techniques promoting a disciplined approach to software development.

Performance: In software engineering, *performance* is a synonym for *efficiency*. It refers to how economically the software utilizes the resources of the computer.

Portability: Software is portable if it can run on different machines.

Productivity: Efficiency of the software process.

Prototyping: A development process in which early executable versions of the end product are delivered as prototypes, with the main purpose of verifying the adequacy of specifications and driving further development.

Quality assurance: The process of verifying whether a software product meets the required qualities.

Reliability: Software is reliable if the user can depend on it.

Reparability: A software system is reparable if it allows the correction of its defects with a limited amount of work.

Reusability: Ease of reusing software components in more than one product. Reusability can also refer to other artifacts (such as requirements, design, etc.).

Robustness: Software is robust if it behaves "reasonably," even in circumstances that were not anticipated in the requirements specification.

Security: A system is secure if it protects its data and services from unauthorized access and modification.

Software process: Activities through which a software product is developed and maintained.

Software product: All of the artifacts produced by a software process. This definition encompasses not only the executable code and user manuals that are delivered to the customer, but also requirements and design documents, source code, test data, etc.

Timeliness: A process-related quality meaning the ability to deliver a product on time.

Usability: A software system is usable — or user-friendly — if its human users find it easy to use. This definition reflects the subjective nature of usability.

Verifiability: A software system is verifiable if its properties can be verified easily.

Visibility: A process-related quality meaning that all steps and the current process status are documented clearly.

References

Boehm et al. [1978]. B.W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G. MacLeod, and M.J. Merritt, *Characteristics of Software Quality*, volume 1 of TRW Series on Software Technology, North-Holland, Amsterdam.

Fenton and Pfleeger [1998]. N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed., PWS Publishing, Boston, MA.

Garg and Jazayeri [1996]. P. Garg and M. Jazayeri,*Process-Centered Software Engineering Environments*, IEEE Computer Society Press.

Ghezzi et al. [2002]. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd edition. Prentice Hall.

Hoffman and Weiss [2001]. D.M. Hoffman and D.M. Weiss, Eds., *Software Fundamentals — Collected Papers by David L. Parnas*, Addison-Wesley, Reading, MA.

Humphrey [1989]. W.S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, MA.

ISO 9126 [1991]. ISO/IEC 9126, *Information Technology — Software Product Evaluation — Quality Characteristics and Guidelines for Their Use*, 1991-12-15.

Jazayeri [1995]. M. Jazayeri, "Component programming — a fresh look at software components," in *Proceedings of the 5th European Software Engineering Conference*, Lecture Notes in Computer Science 989, Springer-Verlag, pages 457–478.

Neumann [1995]. P.G. Neumann, *Computer-Related Risks*. Addison-Wesley, Reading, MA.

Parnas [1978]. D.L. Parnas, "Some software engineering principles," in *Structured Analysis and Design*, *State of the Art Report*, INFOTECH International, pages 237–247.

Further Information

This chapter is adapted from Chapters 2 and 3 of Ghezzi et al. [2003], which is a general textbook on software engineering.

A classification of software qualities is presented and discussed in detail by Boehm et al. [1978]. The international standard [ISO 9126] also provides a list and a discussion of major software qualities. Fenton and Pfleeger [1998] give a comprehensive study of software metrics for quality. Neumann [1995] illustrates the consequences of lack of quality in software. These real-life situations should concern all software professionals.

Humphrey [1985] defined the software process maturity model. The book is devoted to improving software quality through better processes and the assessment of the process. Garg and Jazayeri [1995] is a collection of articles on software environments that integrate and automate the software process.

Parnas's work on design methods is the major source of insight into the concepts of separation of concerns, modularity, abstraction, and anticipation of change. In particular, Parnas [1978] illustrates important software engineering principles. Thirty of Parnas's important papers have been collected in Hoffman and Weiss [2001], along with some updates and commentaries.

Jazayeri [1995] discusses and gives concrete examples of the power of the principle of generality in design and programming.