CHAPTER 17

Software Quality

Quality is never an accident; it is always the result of intelligent effort.
— *John Ruskin*

17.1 FIVE VIEWS OF SOFTWARE QUALITY

In the early days of computers, software developers mainly focused on product functionalities, and most of the end users were highly qualified professionals, such as mathematicians, scientists, and engineers. Development of personal computers and advances in computer networks, the World Wide Web, and graphical user interface made computer software highly accessible to all kinds of users. These days there is widespread computerization of many processes that used to be done by hand. For example, until the late 1990s taxpayers used to file returns on paper, but these days there are numerous web-based tax filing systems. There has been increasing customer expectations in terms of better quality in software products, and developers are under tremendous pressure to deliver high-quality products at a lower cost. Even though competing products deliver the same functionalities, it is the lower cost products with better quality attributes that survive in the competitive market. Therefore, all stakeholders—users, customers, developers, testers, and managers—in a product must have a broad understanding of the overall concept of software quality.

A number of factors influence the making and buying of software products. These factors are user's needs and expectations, the manufacturer's considerations, the inherent characteristics of a product, and the perceived value of a product. To be able to capture the quality concept, it is important to study quality from a broader perspective. This is because the concept of quality predates software development. In a much cited paper published in the *Sloan Management Review* [1], Garvin has analyzed how quality is perceived in different manners in different domains, namely, philosophy, economics, marketing, and management:

Transcendental View: In the transcendental view quality is something that can be recognized through experience but is not defined in some tractable

Software Testing and Quality Assurance: Theory and Practice, Edited by Kshirasagar Naik and Priyadarshi Tripathy
Copyright © 2008 John Wiley & Sons, Inc.

form. Quality is viewed to be something ideal, which is too complex to lend itself to be precisely defined. However, a good-quality object stands out, and it is easily recognized. Because of the philosophical nature of the transcendental view, no effort is made to express it using concrete measures.

User View: The user view concerns the extent to which a product meets user needs and expectations. Quality is not just viewed in terms of what a product can deliver, but it is also influenced by the service provisions in the sales contract. In this view, a user is concerned with whether or not a product is fit for use. This view is highly personalized in nature. The idea of *operational profile*, discussed in Chapter 15, plays an important role in this view. Because of the personalized nature of the product view, a product is considered to be of good quality if it satisfies the needs of a large number of customers. It is useful to identify what product attributes users consider to be important. The reader may note that the user view can encompass many subjective elements apart from the expected functionalities central to user satisfaction. Examples of subjective elements are *usability*, *reliability*, *testability*, and *efficiency*.

Manufacturing View: The manufacturing view has its genesis in the manufacturing sectors, such as the automobile and electronics sectors. In this view, quality is seen as conforming to requirements. Any deviation from the stated requirements is seen as reducing the quality of the product. The concept of *process* plays a key role in the manufacturing view. Products are to be manufactured "right the first time" so that development cost and maintenance cost are reduced. However, there is no guarantee that conforming to process standards will lead to good products. Some criticize this view with an argument that conformance to a process can only lead to *uniformity* in the products, and, therefore, it is possible to manufacture bad-quality products in a consistent manner. However, product quality can be incrementally enhanced by continuously improving the process. Development of the *capability maturity model* (CMM) [2] and ISO 9001 [3] are based on the manufacturing view.

Product View: The central hypothesis in the product view is this: *If a product is manufactured with good internal properties, then it will have good external qualities*. The product view is attractive because it gives rise to an opportunity to explore causal relationships between *internal properties* and *external qualities* of a product. In this view, the current quality level of a product indicates the presence or absence of measurable product properties. The product view of quality can be assessed in an objective manner. An example of the product view of software quality is that high degree of modularity, which is an internal property, makes a software testable and maintainable.

Value-Based View: The value-based view represents a merger of two independent concepts: *excellence* and *worth*. Quality is a measure of excellence, and value is a measure of worth. The central idea in the value-based view is how much a customer is willing to pay for a certain level of quality. The reality is that quality is meaningless if a product does not make economic sense. Essentially, the value-based view represents a trade-off between cost and quality.

Measuring Quality The five viewpoints help us in understanding different aspects of the quality concept. On the other hand, measurement allows us to have a quantitative view of the quality concept. In the following, we explain the reasons for developing a quantitative view of a software system [4]:

- Measurement allows us to establish baselines for qualities. Developers must know the minimum level of quality they must deliver for a product to be acceptable.
- Organizations make continuous improvements in their process models—and an improvement has a cost associated with it. Organizations need to know how much improvement in quality is achieved at a certain cost incurred due to process improvement. This causal relationship is useful in making management decisions concerning process improvement. Sometimes it may be worth investing more in process improvement, whereas some other time the return may not be significant.
- The present level of quality of a product needs to be evaluated so the need for improvements can be investigated.

Measurement of User's View The user's view encompasses a number of quality factors, such as functionality, reliability, and usability. It is easy to measure how much of the functionalities a software product delivers by designing at least one test case for each functionality. A product may require multiple test cases for the same functionality if the functionality is to be performed in different execution environments. Then, the ratio of the number of passed test cases to the total number of test cases designed to verify the functionalities is a measure of the functionalities delivered by the product. Among the qualities that reflect the user's view, the concept of *reliability* has drawn the most attention of researchers.

In the ISO 9126 quality model, *usability* has been broken down into three subcharacteristics, namely, *learnability*, *understandability*, and *operability*. Learnability can be specified as the average elapsed time for a typical user to gain a certain level of competence in using the product. Similarly, understandability can be quantified as the average time needed by a typical user to gain a certain level of understanding of the product. One can quantify operability in a similar manner. The basic idea of breaking down usability into learnability, understandability, and operability can be seen in light of Gilb's technique [5]: *The quality concept is broken down into component parts until each can be stated in terms of directly measurable attributes*. Gilb's technique is a general one to be applicable to a wide variety of user-level qualities.

Measurement of Manufacturer's View Manufacturers are interested in obtaining measures of the following two different quantities:

- **Defect Count:** How many defects have been detected?
- **Rework Cost:** How much does it cost to fix the known defects?

Defect count represents the number of all the defects that have been detected so far. If a product is in operation, this count includes the defects detected during development and operation. A defect count reflects the quality of work produced. Merely counting the defects is of not much use unless something can be done to improve the development process to reduce the defect count in subsequent projects. One can analyze the defects as follows:

- For each defect identify the development phase in which it was introduced and the phase in which it was discovered. Let us assume that a large fraction of the defects are introduced in the requirements gathering phase, and those are discovered during system testing. Then, we can conclude that requirement analysis was not adequately performed. We can also conclude that work done subsequently, such as design verification and unit testing, were not of high standard. If a large number of defects are found during system operation, one can say that system testing was not rigorously performed.
- Categorize the defects based on modules. Assuming that a module is a cohesive entity performing a well-defined task, by identifying the modules containing most of the defects we can identify where things are going wrong. This information can be used in managing resources. For example, if a large number of defects are found in a communication module in a distributed application, more resource could be allocated to train developers in the details of the communication system.
- To compare defects across modules and products in a meaningful way, normalize the defect count by product size. By normalizing defect count by product size in terms of the number of LOC, we can obtain a measure, called *defect density*. Intuitively, defect density is expressed as the *number of defects found per thousand lines of code*.
- Separate the defects found during operation from the ones found during development. The ratio of the number of defects found during operation to the total number of defects is a measure of the effectiveness of the entire gamut of test activities. If the ratio is close to zero, we can say that testing was highly effective. On the other hand, if the ratio is farther from the ideal value of zero, say, 0.2, it is apparent that all the testing activities detected only 80% of the defects.

After defects are detected, the developers make an effort to fix them. Ultimately, it costs some money to fix defects—this is apart from the "reputation" cost to an organization from defects discovered during operation. The rework cost includes all the additional cost associated with defect-related activities, such as fixing documents. Rework is an additional cost that is incurred due to work being done in a less than perfect manner the first time it was done. It is obvious that organizations strive to reduce the total cost of software development, including the rework cost. The rework cost can be split into two parts as follows:

- **Development Rework Cost:** This is the rework cost incurred *before* a product is released to the customers.
- **Operation Rework Cost:** This is the rework cost incurred *when* a product is in operation.

On the one hand, the development rework cost is a measure of development efficiency. In other words, if the development rework cost is zero, then the development efficiency is very high. On the other hand, the operation rework cost is a measure of the delivered quality of the product in operation. If the development rework cost is zero, then the delivered quality of the product in operation is very high. This is because the customers have not encountered any defect and, consequently, the development team is not spending any resource on defect fixing.

17.2 MCCALL'S QUALITY FACTORS AND CRITERIA

The concept of software quality and the efforts to understand it in terms of measurable quantities date back to the mid-1970s. McCall, Richards, and Walters [6] were the first to study the concept of software quality in terms of quality factors and quality criteria.

17.2.1 Quality Factors

A *quality factor* represents a behavioral characteristic of a system. Some examples of high-level quality factors are *correctness*, *reliability*, *efficiency*, *testability*, *portability*, and *reusability*. A full list of the quality factors will be given in a later part of this section. As the examples show, quality factors are external attributes of a software system. Customers, software developers, and quality assurance engineers are interested in different quality factors to a different extent. For example, customers may want an efficient and reliable software with less concern for portability. The developers strive to meet customer needs by making their system efficient and reliable, at the same time making the product portable and reusable to reduce the cost of software development. The software quality assurance team is more interested in the testability of a system so that some other factors, such as correctness, reliability, and efficiency, can be easily verified through testing. The testability factor is important to developers and customers as well: (i) Developers want to test their product before delivering it to the software quality assurance team and (ii) customers want to perform acceptance tests before taking delivery of a product. In Table 17.1, we list the quality factors as defined by McCall et al. [6]. Now we explain the 11 quality factors in more detail:

Correctness: A software system is expected to meet the explicitly specified functional requirements and the implicitly expected nonfunctional requirements. If a software system satisfies all the functional requirements, the

TABLE 17.1 McCall's Quality Factors

Quality Factors	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives
Reliability	Extent to which a program can be expected to perform its intended function with required precision
Efficiency	Amount of computing resources and code required by a program to perform a function
Integrity	Extent to which access to software or data by unauthorized persons can be controlled
Usability	Effort required to learn, operate, prepare input, and interpret output of a program
Maintainability	Effort required to locate and fix a defect in an operational program
Testability	Effort required to test a program to ensure that it performs its intended functions
Flexibility	Effort required to modify an operational program
Portability	Effort required to transfer a program from one hardware and/or software environment to another
Reusability	Extent to which parts of a software system can be reused in other applications
Interoperability	Effort required to couple one system with another

Source: From ref. 6.

system is said to be correct. However, a correct software system may still be unacceptable to customers if the system fails to meet unstated requirements, such as stability, performance, and scalability. On the other hand, even an incorrect system may be accepted by users.

Reliability: It is difficult to construct large software systems which are correct. A few functions may not work in all execution scenarios, and, therefore, the software is considered to be incorrect. However, the software may still be acceptable to customers because the execution scenarios causing the system to fail may not frequently occur when the system is deployed. Moreover, customers may accept software failures once in a while. Customers may still consider an incorrect system to be reliable if the failure rate is very small and it does not adversely affect their mission objectives. Reliability is a customer perception, and an incorrect software can still be considered to be reliable.

Efficiency: Efficiency concerns to what extent a software system utilizes resources, such as computing power, memory, disk space, communication bandwidth, and energy. A software system must utilize as little resources as possible to perform its functionalities. For example, by utilizing less communication bandwidth a base station in a cellular telephone network can support more users.

Integrity: A system's integrity refers to its ability to withstand attacks to its security. In other words, integrity refers to the extent to which access to software or data by unauthorized persons or programs can be controlled. Integrity has assumed a prominent role in today's network-based applications. Integrity is also an issue in multiuser systems.

Usability: A software system is considered to be usable if human users find it easy to use. Users put much emphasis on the user interface of software systems. Without a good user interface a software system may fizzle out even if it possesses many desired qualities. However, it must be remembered that a good user interface alone cannot make a product successful—the product must also be reliable, for example. If a software fails too often, no good user interface can keep it in the market.

Maintainability: In general, maintenance refers to the upkeep of products in response to deterioration of their components due to continued use of the products. Maintainability refers to how easily and inexpensively the maintenance tasks can be performed. For software products, there are three categories of maintenance activities: corrective, adaptive, and perfective. Corrective maintenance is a postrelease activity, and it refers to the removal of defects existing in an in-service software. The existing defects might have been known at the time of release of the product or might have been introduced during maintenance. Adaptive maintenance concerns adjusting software systems to changes in the execution environment. Perfective maintenance concerns modifying a software system to improve some of its qualities.

Testability: It is important to be able to verify every requirement, both explicitly stated and simply expected. Testability means the ability to verify requirements. At every stage of software development, it is necessary to consider the testability aspect of a product. Specifically, for each requirement we try to answer the question: What procedure should one use to test the requirement, and how easily can one verify it? To make a product testable, designers may have to instrument a design with functionalities not available to the customer.

Flexibility: Flexibility is reflected in the cost of modifying an operational system. As more and more changes are effected in a system throughout its operational phase, subsequent changes may cost more and more. If the initial design is not flexible, it is highly likely that subsequent changes are very expensive. In order to measure the flexibility of a system, one has to find an answer to the question: How easily can one add a new feature to a system?

Portability: Portability of a software system refers to how easily it can be adapted to run in a different execution environment. An execution environment is a broad term encompassing hardware platform, operating system, distributedness, and heterogeneity of the hardware system, to name a few. Portability is important for developers because a minor adaptation of a system can increase its market potential. Moreover, portability gives customers an option to easily move from one execution environment to another to best utilize emerging technologies in furthering their business. Good design principles such as modularity facilitate portability. For example, all environment-related computations can be localized in a few modules so that those can be easily identified and modified to port the system to another environment.

Reusability: Reusability means if a significant portion of one product can be reused, maybe with minor modification, in another product. It may not be economically viable to reuse small components. Reusability saves the cost and time to develop and test the component being reused. In the field of scientific computing, mathematical libraries are commonly reused. Reusability is not just limited to product parts, rather it can be applied to processes as well. For example, we are very much interested in developing good processes that are largely repeatable.

Interoperability: In this age of computer networking, isolated software systems are turning into a rarity. Today's software systems are coupled at the input–output level with other software systems. Intuitively, interoperability means whether or not the output of one system is acceptable as input to another system; it is likely that the two systems run on different computers interconnected by a network. When we consider Internet-based applications and wireless applications, the need for interoperability is simply overriding. For example, users of document processing packages, such as LaTex and Microsoft Word, want to import a variety of images produced by different graphics packages. Therefore, the graphics packages and the document processing packages must be interoperable. Another example of interoperability is the ability to roam from one cellular phone network in one country to another cellular network in another country.

The 11 quality factors defined in Table 17.1 have been grouped into three broad categories as follows:

- Product operation
- Product revision
- Product transition

The elements of each of the three broad categories are identified and further explained in Table 17.2. It may be noted that the above three categories relate more to postdevelopment activities expectations and less to in-development activities. In other words, McCall's quality factors emphasize more on the quality levels of a product delivered by an organization and the quality levels of a delivered product relevant to product maintenance. Quality factors in the product operation category refer to delivered quality. Testability is an important quality factor that is of much significance to developers during both product development and maintenance. Maintainability, flexibility, and portability are desired quality factor sought

TABLE 17.2 Categorization of McCall's Quality Factors

Quality Categories	Quality Factors	Broad Objectives
Product operation	Correctness	Does it do what the customer wants?
	Reliability	Does it do it accurately all of the time?
	Efficiency	Does it quickly solve the intended problem?
	Integrity	Is it secure?
	Usability	Can I run it?
Product revision	Maintainability	Can it be fixed?
	Testability	Can it be tested?
	Flexibility	Can it be changed?
Product transition	Portability	Can it be used on another machine?
	Reusability	Can parts of it be reused?
	Interoperability	Can it interface with another system?

Source: From ref. 6.

in a product so that the task of supporting the product after delivery is less expensive. Reusability is a quality factor that has the potential to reduce the development cost of a project by allowing developers to reuse some of the components from an existing product. Interoperability allows a product to coexist with other products, systems, and features.

17.2.2 Quality Criteria

A *quality criterion* is an attribute of a quality factor that is related to software development. For example, modularity is an attribute of the architecture of a software system. A highly modular software allows designers to put cohesive components in one module, thereby increasing the maintainability of the system. Similarly, traceability of a user requirement allows developers to accurately map the requirement to a subset of the modules, thereby increasing the correctness of the system. Some quality criteria relate to products and some to personnel. For example, modularity is a product-related quality criterion, whereas training concerns development and software quality assurance personnel. In Table 17.3, we list the 23 quality criteria defined by McCall et al. [6].

17.2.3 Relationship between Quality Factors and Criteria

The relationship between quality factors and quality criteria is shown in Figure 17.1. An arrow from a quality criterion to a quality factor means that the quality criterion has a positive impact on the quality factor. For example, traceability has a positive impact on correctness. Similarly, the quality criterion simplicity positively impacts reliability, usability, and testability.

Though it is desirable to improve all the quality factors, doing so may not be possible. This is because, in general, quality factors are not completely independent. Thus, we note two characteristics of the relationship as follows:

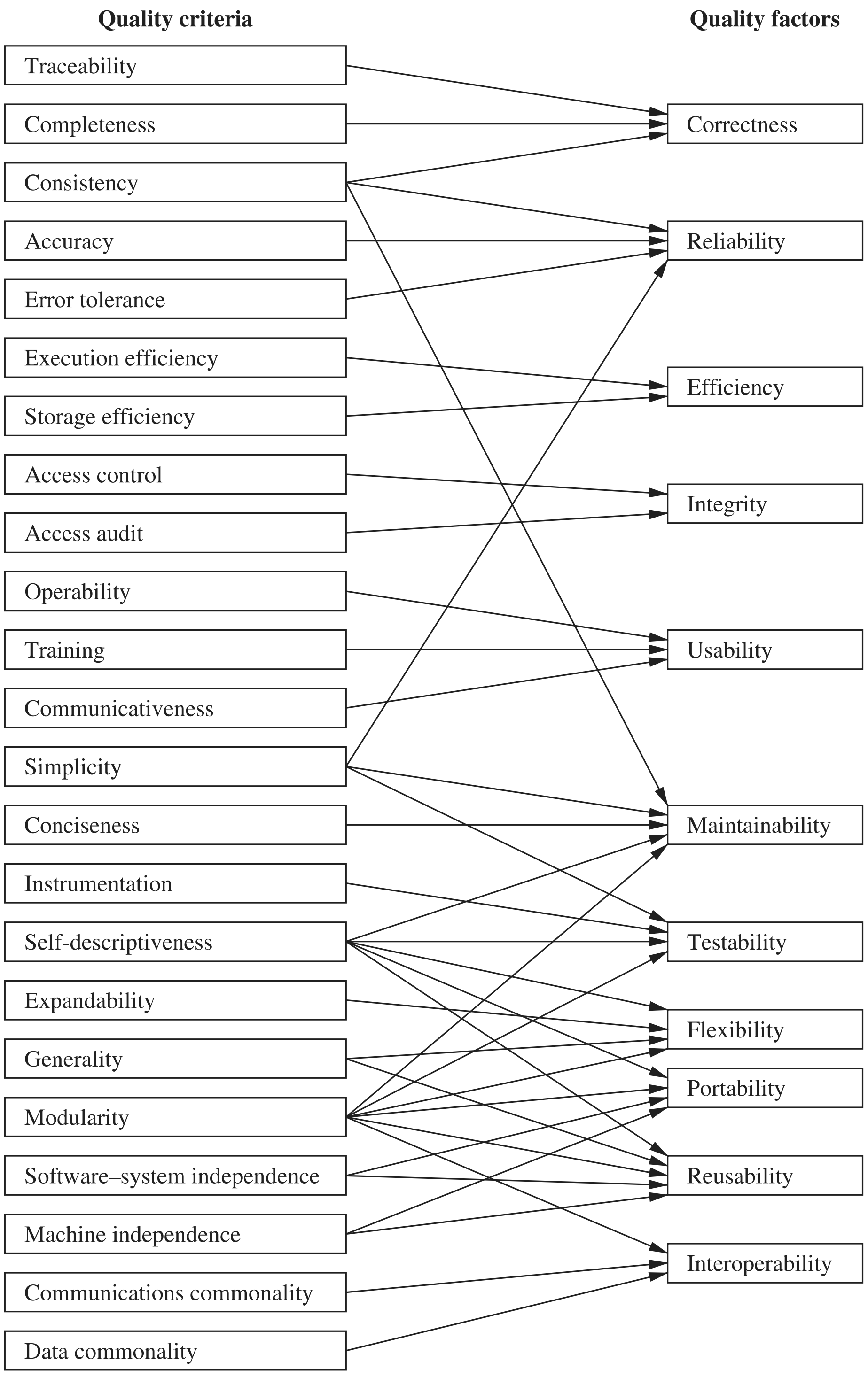

Figure 17.1 Relation between quality factors and quality criteria [6].

- If an effort is made to improve one quality factor, another quality factor may be degraded. For example, if an effort is made to make a software product testable, the efficiency of the software is likely to go down. To make code testable, programmers may not be able to write compact code. Moreover, if we are interested in making a product portable, the code must

TABLE 17.3 McCall's Quality Criteria

Quality Criteria	Definition
Access audit	Ease with which software and data can be checked for compliance with standards or other requirements
Access control	Provisions for control and protection of the software and data
Accuracy	Precision of computations and output
Communication commonality	Degree to which standard protocols and interfaces are used
Completeness	Degree to which a full implementation of the required functionalities has been achieved
Communicativeness	Ease with which inputs and outputs can be assimilated
Conciseness	Compactness of the source code, in terms of lines of code
Consistency	Use of uniform design and implementation techniques and notation throughout a project
Data commonality	Use of standard data representations
Error tolerance	Degree to which continuity of operation is ensured under adverse conditions
Execution efficiency	Run time efficiency of the software
Expandability	Degree to which storage requirements or software functions can be expanded
Generality	Breadth of the potential application of software components
Hardware independence	Degree to which the software is dependent on the underlying hardware
Instrumentation	Degree to which the software provides for measurement of its use or identification of errors
Modularity	Provision of highly independent modules
Operability	Ease of operation of the software
Self-documentation	Provision of in-line documentation that explains implementation of components
Simplicity	Ease with which the software can be understood.
Software system independence	Degree to which the software is independent of its software environment—nonstandard language constructs, operating system, libraries, database management system, etc.
Software efficiency	Run time storage requirements of the software
Traceability	Ability to link software components to requirements
Training	Ease with which new users can use the system

Source: From ref. 6.

be written in such a manner that it is easily understandable, and, hence, code need not be in a compact form. An effort to make code portable is likely to reduce its efficiency. In fact, attempts to improve integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability will reduce the efficiency of a software system.

- Some quality factors positively impact others. For example, an effort to enhance the correctness of a system will increase its reliability. As another example, an effort to enhance the testability of a system will improve its maintainability.

17.2.4 Quality Metrics

The high-level quality factors cannot be measured directly. For example, we cannot directly measure the testability of a software system. Neither can testability be expressed in "yes" or "no" terms. Instead, the degree of testability can be assessed by associating with testability a few quality metrics, namely, *simplicity*, *instrumentation*, *self-descriptiveness*, and *modularity*. A *quality metric* is a measure that captures some aspect of a quality criterion. One or more quality metrics should be associated with each criterion. The metrics can be derived as follows:

- Formulate a set of relevant questions concerning the quality criteria and seek a "yes" or "no" answer for each question.
- Divide the number of "yes" answers by the number of questions to obtain a value in the range of 0 to 1. The resulting number represents the intended quality metric.

For example, we can ask the following question concerning the *self-descriptiveness* of a product: *Is all documentation written clearly and simply such that procedures, functions, and algorithms can be easily understood?* Another question concerning self-descriptiveness is: *Is the design rationale behind a module clearly understood?* Different questions can have different degrees of importance in the computation of a metric, and, therefore, individual "yes" answers can be differently weighted in the above computation.

The above way of computing the value of a metric is highly subjective. The degree of subjectivity varies significantly from question to question in spite of the fact that all the responses are treated equally. It is difficult to combine different metrics to get a measure of a higher level quality factor. In addition, for some questions it is more meaningful to consider a response on a richer measurement scale. For example, the question "Is the design of a software system simple?" needs to be answered on a multiple ordinal scale to reflect a variety of possible answers, rather than a yes-or-no answer.

Similarly, one cannot directly measure the reliability of a system. However, the number of distinct failures observed so far is a measure of the initial reliability of the system. Moreover, the time gap between observed failures is treated as a measure of the reliability of a system.

17.3 ISO 9126 QUALITY CHARACTERISTICS

There has been international collaboration among experts to define a general framework for software quality. An expert group, under the aegis of the ISO, standardized a software quality document, namely, ISO 9126, which defines six broad, independent categories of quality characteristics as follows: