CHAPTER 2

WHAT IS SOFTWARE QUALITY?

The question, "What is software quality?", is bound to generate many different answers, depending on whom you ask, under what circumstances, for what kind of software systems, and so on. An alternative question that is probably easier for us to get more informative answers is: "What are the characteristics for high-quality software?"

In this chapter, we attempt to define software quality by defining the expected characteristics or properties of high-quality software. In doing so, we need to examine the different perspectives and expectations of users as well as other people involved with the development, management, marketing, and maintenance of the software products. We also need to examine the individual characteristics associated with quality and their inter-relationship, and focus our attention on the critical characteristics of functional correctness. We conclude the chapter with a comparison of software quality with quality concepts for other (non-software) systems and the evolving place of quality within software engineering.

2.1 QUALITY: PERSPECTIVES AND EXPECTATIONS

We next examine the different views of quality in a systematic manner, based on the different roles, responsibilities, and quality expectations of different people, and zoom in on a small set of views and related properties to be consistently followed throughout this book. Five major views according to (Kitchenham and Pfleeger, 1996; Pfleeger et al., 2002) are: transcendental, user, manufacturing, product, and value-based views, as outlined below:

- In the *transcendental* view, quality is hard to define or describe in abstract terms, but can be recognized if it is present. It is generally associated with some intangible properties that delight users.
- In the *user* view, quality is fitness for purpose or meeting user's needs.
- In the *manufacturing* view, quality means conformance to process standards.
- In the *product* view, the focus is on inherent characteristics in the product itself in the hope that controlling these internal quality indicators (or the so-called product-internal metrics described in Chapter 18) will result in improved external product behavior (quality in use).
- In the *value-based* view, quality is the customers' willingness to pay for a software.

People's roles and responsibilities

When software quality is concerned, different people would have different views and expectations based on their roles and responsibilities. With the quality assurance (QA) and quality engineering focus of this book, we can divide the people into two broad groups:

- *Consumers* of software products or services, including customers and users, either internally or externally. Sometime we also make the distinction between the *customers*, who are responsible for the acquisition of software products or services, and the *users*, who use the software products or services for various purposes, although the dual roles of customers and users are quite common. We can also extend the concept of users to include such non-human or "invisible" users as other software, embedded hardware, and the overall operational environment that the software operates under and interacts with (Whittaker, 2001).
- *Producers* of software products, or anyone involved with the development, management, maintenance, marketing, and service of software products. We adopt a broad definition of producers, which also include third-party participants who may be involved in add-on products and services, software packaging, software certification, fulfilling independent verification and validation (IV&V) responsibilities, and so on.

Subgroups within the above groups may have different concerns, although there are many common concerns within each group. In the subsequent discussions, we use *external* view for the first group's perspective, who are more concerned with the observed or external behavior, rather than the internal details that lead to such behavior. Similarly, we use a generic label *internal* view for the second group's perspective, because they are typically familiar with or at least aware of various internal characteristic of the products. In other words, the external view mostly sees a software system as a black box, where one can observe its behavior but not see through inside; while the internal view mostly sees it as a white box, or more appropriately a clear box, where one can see what is inside and how it works.

Quality expectations on the consumer side

The basic quality expectations of a user are that a software system performs useful functions as it is specified. There are two basic elements to this expectation: First, it performs right functions as specified, which, hopefully fits the user's needs (fit for use). Second, it performs these specified functions correctly over repeated use or over a long period of time, or performs its functions *reliably*. These two elements are related to the validation and verification aspects of QA we introduced in the previous chapter, which will be expanded further in Chapter 4. Looking into the future, we can work towards meeting this basic expectation and beyond to *delight* customers and users by preventing unforeseen negative impacts and produce unexpected positive effects (Denning, 1992).

For many users of today's ubiquitous software and systems, ease of use, or usability, may be a more important quality expectation than reliability or other concerns. For example, the adoption of graphical user interfaces (GUI) in personal computers to replace text-based command interpreters often used in mainframes is primarily driven by the usability concerns for their massive user population. Similarly, ease of installation, is another major trend for software intended for the same population, to allow for painless (and nearly effortless) installation and operation, or the so-called "plug-and-play". However, different users of the same system may have different views and priorities, such as the importance of usability for novice users and the importance of reliability for sophisticated users of the web (Vatanasombut et al., 2004).

When we consider the extended definition of users beyond human users, the primary expectations for quality would be to ensure the smooth operation and interaction between the software and these non-human users in the form of better inter-operability and adaptability, so that the software can work well with others and within its surrounding environment.

The basic quality expectations of a customer are similar to that of a user, with the additional concern for the cost of the software or service. This additional concern can be reflected by the so-called value-based view of quality, that is, whether a customer is willing to pay for it. The competing interests of quality and other software engineering concerns, such as cost, schedule, functionality, and their trade-offs, are examined in Section 2.4.

Quality expectations on the producer side

For software producers, the most fundamental quality question is to fulfill their contractual obligations by producing software products that conform to product specifications or providing services that conform to service agreement. By extension, various product internal characteristics that make it easy to conform to product specifications, such as good designs that maintain conceptual integrity of product components and reduce coupling across different components, are also associated with good quality.

For product and service managers, adherence to pre-selected software process and relevant standards, proper choice of software methodologies, languages, and tools, as well as other factors, may be closely related to quality. They are also interested in managing and satisfying user's quality expectations, by translating such quality expectations into realistic quality goals that can be defined and managed internally, selecting appropriate and effective QA strategies, and seeing them through.

For other people on the producer side, their different concerns may also produce quality views and expectations different from the above. For example, usability and modifiability may be paramount for people involved with software service, maintainability for maintenance personnel, portability for third-party or software packaging service providers, and profitability and customer value for product marketing.

2.2 QUALITY FRAMEWORKS AND ISO-9126

Based on the different quality views and expectations outlined above, quality can be defined accordingly. In fact, we have already mentioned above various so-called "-ilities" connected to the term quality, such as reliability, usability, portability, maintainability, etc. Various models or frameworks have been proposed to accommodate these different quality views and expectations, and to define quality and related attributes, features, characteristics, and measurements. We next briefly describe ISO-9126 (ISO, 2001), the mostly influential one in the software engineering community today, and discuss various adaptations of such quality frameworks for specific application environments.

ISO-9126

ISO-9126 (ISO, 2001) provides a hierarchical framework for quality definition, organized into quality characteristics and sub-characteristics. There are six top-level quality characteristics, with each associated with its own exclusive (non-overlapping) sub-characteristics, as summarized below:

- Functionality: A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. The sub-characteristics include:
  - Suitability
  - Accuracy
  - Interoperability
  - Security
- Reliability: A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. The sub-characteristics include:
  - Maturity
  - Fault tolerance
  - Recoverability
- Usability: A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. The sub-characteristics include:
  - Understandability
  - Learnability
  - Operability
- Efficiency: A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. The sub-characteristics include:
  - Time behavior
  - Resource behavior

- Maintainability: A set of attributes that bear on the effort needed to make specified modifications. The sub-characteristics include:
  - Analyzability
  - Changeability
  - Stability
  - Testability
- Portability: A set of attributes that bear on the ability of software to be transferred from one environment to another. The sub-characteristics include:
  - Adaptability
  - Installability
  - Conformance
  - Replaceability

Alternative frameworks and focus on correctness

ISO-9126 offers a comprehensive framework to describe many attributes and properties we associate with quality. There is a strict hierarchy, where no sub-characteristics are shared among quality characteristics. However, certain product properties are linked to multiple quality characteristics or sub-characteristics (Dromey, 1995; Dromey, 1996). For example, various forms of redundancy affect both efficiency and maintainability. Consequently, various alternative quality frameworks have been proposed to allow for more flexible relations among the different quality attributes or factors, and to facilitate a smooth transition from specific quality concerns to specific product properties and metrics.

Many companies and communities associated with different application domains have adapted and customized existing quality frameworks to define quality for themselves, taking into consideration their specific business and market environment. One concrete example of this for companies is the quality attribute list CUPRIMDS (capability, usability, performance, reliability, installation, maintenance, documentation, and service) IBM used for their software products (Kan, 2002). CUPRIMDS is often used together with overall customer satisfaction (thus the acronym CUPRIMDSO) to characterize and measure software quality for IBM's software products.

Similarly, a set of quality attributes has been identified for web-based applications (Offutt, 2002), with the primary quality attributes as reliability, usability, and security, and the secondary quality attributes as availability, scalability, maintainability, and time to market. Such prioritized schemes are often used for specific application domains. For example, performance (or efficiency) and reliability would take precedence over usability and maintainability for real-time software products. On the contrary, it might be the other way round for mass market products for end users.

Among the software quality characteristics or attributes, some deal directly with the functional *correctness*, or the *conformance* to specifications as demonstrated by the absence of problems or instances of non-conformance. Other quality characteristics or attributes deal with usability, portability, etc. Correctness is typically related to several quality characteristics or sub-characteristics in quality frameworks described above. For example, in ISO-9126 it is related to both functionality, particularly its accuracy (in other words, conformance) sub-characteristics, and reliability.

Correctness is typically the most important aspect of quality for situations where daily life or business depends on the software, such as in managing corporate-wide computer networks, financial databases, and real-time control software. Even for market segments where new features and usability take priority, such as for web-based applications and software for personal use in the mass market, correctness is still a fundamental part of the users' expectations (Offutt, 2002; Prahalad and Krishnan, 1999). Therefore, we adopt the correctness-centered view of quality throughout this book. We will focus on correctness-related quality attributes and related ways to ensure and demonstrate quality defined as such.

2.3 CORRECTNESS AND DEFECTS: DEFINITIONS, PROPERTIES, AND MEASUREMENTS

When many people associate *quality* or high-quality with a software system, it is an indication that few, if any, software problems, are expected to occur during its operations. What is more, when problems do occur, the negative impact is expected to be minimal. Related issues are discussed in this section.

Definitions: Error, fault, failure, and defect

Key to the correctness aspect of software quality is the concept of defect, failure, fault, and error. The term "defect" generally refers to some problem with the software, either with its external behavior or with its internal characteristics. The IEEE Standard 610.12 (IEEE, 1990) defines the following terms related to defects:

- *Failure*: The inability of a system or component to perform its required functions within specified performance requirements.
- *Fault*: An incorrect step, process, or data definition in a computer program.
- *Error*: A human action that produces an incorrect result.

Therefore, the term *failure* refers to a behavioral deviation from the user requirement or the product specification; *fault* refers to an underlying condition within a software that causes certain failure(s) to occur; while *error* refers to a missing or incorrect human action resulting in certain fault(s) being injected into a software.

We also extend errors to include *error sources*, or the root causes for the missing or incorrect actions, such as human misconceptions, misunderstandings, etc. Failures, faults, and errors are collectively referred to as *defects* in literature. We will use the term defect in this book in this collective sense or when its derivatives are commonly used in literature, such as in defect handling.

Software problems or defects, are also commonly referred to as "bugs". However, the term bug is never precisely defined, such as the different aspects of defects defined as errors, faults, and failures above. Some people have also raised the moral or philosophical objection to the use of bug as evading responsibility for something people committed. Therefore, we try to avoid using the term "bug" in this book.

Similarly, we also try to avoid using the related terms "debug" or "debugging" for similar reasons. The term "debug" general means "get rid of the bugs". Sometimes, it also includes activities related to detecting the presence of bugs and dealing with them. In this book, we will use, in their place, the following terms:

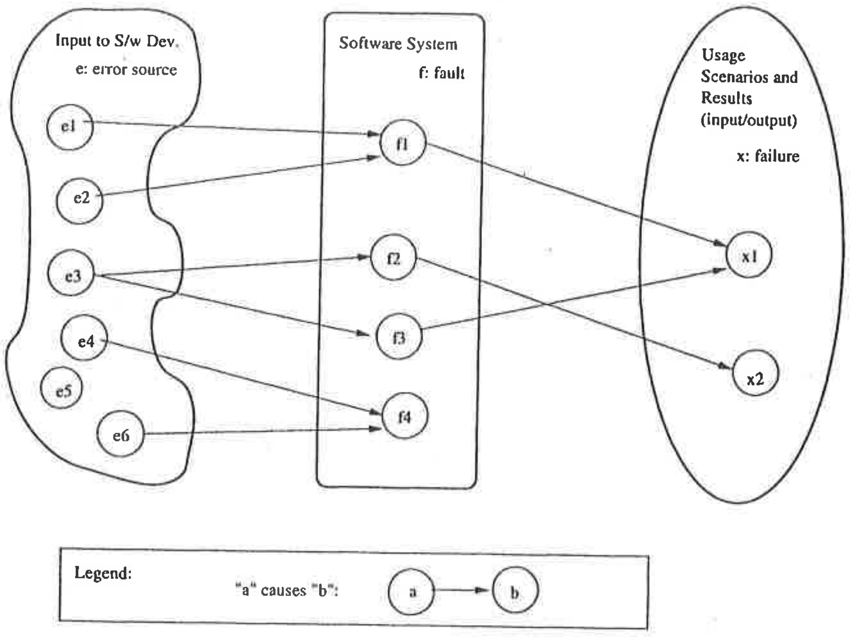

Figure 2.1 Defect related concepts and relations

- We use *defect detection and removal* for the overall concept and activities related to what many people commonly call "debugging".
- When specific activities related to "debugging" are involved, we point the specifics out using more precisely defined terms, including,
  - Specific activities related to defect discovery, including testing, inspection, etc.
  - Specific follow-up activities after defect discovery, including defect diagnosis, analysis, fixing, and re-verification.

All these specific terms will be more precisely defined in this book when they are introduced or when topics most closely related to them are covered.

Concepts and relations illustrated

The concepts of error (including error source), fault, failure, and defect can be placed into the context of software artifact, software development activities, and operational usage, as depicted in Figure 2.1. Some specific information illustrated include:

- The software system as represented by its artifacts is depicted in the middle box. The artifacts include mainly software code and sometime other artifacts such as designs, specifications, requirement documents, etc. The *faults* scattered among these artifacts are depicted as circled entities within the middle box.
- The input to the software development activities, depicted in the left box, include conceptual models and information, developers with certain knowledge and experience, reusable software components, etc. Various *error sources* are also depicted as circled entities within this left box.

- The *errors* as missing or incorrect human actions are not directly depicted within one box, but rather as actions leading to the injection of faults in the middle box because of some error sources in the left box.
- Usage scenarios and execution results, depicted in the right box, describe the input to software execution, its expected dynamic behavior and output, and the overall results. A subset of these behavior patterns or results can be classified as failures when they deviate from the expected behavior, and is depicted as the collection of circled failure instances.

With the above definitions and interpretations, we can see that failures, faults, and errors are different aspects of defects. A causal relation exists among these three aspects of defects:

$$\text{errors} \rightarrow \text{faults} \rightarrow \text{failures}$$

That is, errors may cause faults to be injected into the software, and faults may cause failures when the software is executed. However, this relationship is not necessarily 1-to-1: A single error may cause many faults, such as in the case that a wrong algorithm is applied in multiple modules and causes multiple faults, and a single fault may cause many failures in repeated executions. Conversely, the same failure may be caused by several faults, such as an interface or interaction failure involving multiple modules, and the same fault may be there due to different errors. Figure 2.1 also illustrates some of these situations, as described below:

- The error source *e3* causes multiple faults, *f2* and *f3*.
- The fault *f1* is caused by multiple error sources, *e1* and *e2*.
- Sometimes, an error source, such as *e5*, may not cause any fault injection, and a fault, such as *f4*, may not cause any failure, under the given scenarios or circumstances. Such faults are typically called *dormant* or *latent* faults, which may still cause problems under a different set of scenarios or circumstances.

Correctness-centered properties and measurements

With the correctness focus adopted in this book and the binary partition of people into consumer and producer groups, we can define quality and related properties according to these views (external views for producers vs. internal views for consumers) and attributes (correctness vs. others) in Table 2.1.

The correctness-centered quality from the external view, or from the view of consumers (users and customers) of a software product or service, can be defined and measured by various failure-related properties and measurement. To a user or a customer, the primary concern is that the software operates without failure, or with as few failures as possible. When such failures or undesirable events do occur, the impact should be as little as possible. These concerns can be captured by various properties and related measurements, as follows:

- *Failure properties and direct failure measurement*: Failure properties include information about the specific failures, what they are, how they occur, etc. These properties can be measured directly by examining failure count, distribution, density, etc. We will examine detailed failure properties and measurements in connection with defect classification and analysis in Chapter 20.

Table 2.1 Correctness-centered properties according to quality views and attributes

View	Attribute	
	Correctness	Others
Consumer/ External (user & customer)	Failure-related properties	Usability Maintainability Portability Performance Installability Readability etc. (-ilities)
Producer/ Internal (developer, manager, tester, etc.)	Fault-related properties	Design Size Change Complexity, etc.

- *Failure likelihood and reliability measurement*: How often or how likely a failure is going to occur is of critical concern to software users and customers. This likelihood is captured in various reliability measures, where *reliability* can be defined as the probability of failure-free operations for a specific time period or for a given set of input (Musa et al., 1987; Lyu, 1995a; Tian, 1998). We will discuss this topic in Chapter 22.

- *Failure severity measurement and safety assurance*: The failure impact is also a critical concern for users and customers of many software products and services, especially if the damage caused by failures could be substantial. *Accidents*, which are defined to be failures with severe consequences, need to be avoided, contained, or dealt with to ensure the safety for the personnel involved and to minimize other damages. We will discuss this topic in Chapter 16.

In contrast to the consumers' perspective of quality above, the producers of software systems see quality from a different perspectives in their interaction with software systems and related problems. They need to fix the problems or faults that caused the failures, as well as deal with the injection and activation of other faults that could potentially cause other failures that have not yet been observed.

Similar to the failure properties and related measurements discussed above, we need to examine various fault properties and related measurements from the internal view or the producers' view. We can collect and analyze information about individual faults, as well as do so collectively. Individual faults can be analyzed and examined according to their types, their relations to specific failures and accidents, their causes, the time and circumstances when they are injected, etc. Faults can be analyzed collectively according to their distribution and density over development phases and different software components. These topics will be covered in detail in Chapter 20 in connection with defect classification and analysis. Techniques to identify high-defect areas for focused quality improvement are covered in Chapter 21.

Defects in the context of QA and quality engineering

For most software development organizations, ensuring quality means dealing with defects. Three generic ways to deal with defects include: 1) defect prevention, 2) defect detection and removal, and 3) defect containment. These different ways of dealing with defects and the related activities and techniques for QA will be described in Chapter 3.

Various QA alternatives and related techniques can be used in a concerted effort to effectively and efficiently deal with defects and assure software quality. In the process of dealing with defects, various direct defect measurements and other indirect quality measurements (used as quality indicators) might be taken, often forming a multi-dimensional measurement space referred to as quality profile (Humphrey, 1998). These measurement results need to be analyzed using various models to provide quality assessment and feedback to the overall software development process. Part IV covers these topics.

By extension, quality engineering can also be viewed as defect management. In addition to the execution of the planned QA activities, quality engineering also includes:

- quality planning before specific QA activities are carried out,
- measurement, analysis, and feedback to monitor and control the QA activities.

In this respect, much of quality planning can be viewed as estimation and planning for anticipated defects. Much of the feedback is provided in terms of various defect related quality assessments and predictions. These topics are described in Chapter 5 and Part IV, respectively.

2.4 A HISTORICAL PERSPECTIVE OF QUALITY

We next examine people's views and perceptions of quality in a historical context, and trace the evolving role of software quality in software engineering.

Evolving perceptions of quality

Before software and information technology (IT) industries came into existence, quality has long been associated with physical objects or systems, such as cars, tools, radio and television receivers, etc. Under this traditional setting, QA is typically associated with the manufacturing process. The focus is on ensuring that the products conform to their specifications. What is more, these specifications often accompany the finished products, so that the buyers or users can check them for reference. For example, the user's guide for stereo equipments often lists their specifications in terms of physical dimensions, frequency responses, total harmonic distortion, and other relevant information.

Since many items in the product specifications are specified in terms of ranges and error tolerance, reducing variance in manufacturing has been the focal point of statistical quality control. Quality problems are synonymous to non-conformance to specifications or observed defects defined by the non-conformance. For example, the commonly used "initial quality" for automobiles by the industrial group J.D. Power and Associates (online at `www.jdpa.com`) is defined to be the average number of reported problems per 100 vehicle by owners during the first three years (they used to count only the first year) of their ownership based on actual survey results. Another commonly used quality measure for automobiles, reliability, is measured by the number of problems over a longer time for different stages of an automobile's lifetime. Therefore, it is usually treated as the most important quality measure for used vehicles.

With the development of service industries, an emerging view of quality is that business needs to adjust to the dynamically shifting expectations of customers, with the focus of quality control shifting from zero defect in products to zero defection of customers (Reichheld Jr. and Sasser, 1990). Customer loyalty due to their overall experience with the service is more important than just conforming to some prescribed specifications or standards.

According to (Prahalad and Krishnan, 1999), software industry has incorporated both the conformance and service views of quality, and high-quality software can be defined by three basic elements: conformance, adaptability, and innovation. This view generally agrees with the many facets of software quality we described so far. There are many reasons for this changing view of quality and the different QA focuses (Beizer, 1998). For example, the fundamental assumptions of physical constraints, continuity, quantifiability, composition/decomposition, etc., cannot be extended or mapped to the flexible software world. Therefore, different QA techniques covered in this book need to be used.

Quality in software engineering

Within software engineering, quality has been one of the several important factors, including cost, schedule, and functionality, which have been studied by researchers and practitioners (Blum, 1992; Humphrey, 1989; Ghezzi et al., 2003; von Mayrhauser, 1990). These factors determine the success or failure of a software product in evolving market environments, but may have varying importance for different time periods and different market segments.

In Musa and Everett (1990), these varying primary concerns were conveniently used to divide software engineering into four progressive stages:

1. In the *functional* stage, the focus was on providing the automated functions to replace what had been done manually before.
2. In the *schedule* stage, the focus was on introducing important features and new systems on a timely and orderly basis to satisfy urgent user needs.
3. In the *cost* stage, the focus was on reducing the price to stay competitive accompanied by the widespread use of personal computers.
4. In the *reliability* stage, the focus was managing users' quality expectations under the increased dependency on software and high cost or severe damages associated with software failures.

We can see a gradual increase in importance of quality within software engineering. This general characterization is in agreement with what we have discussed so far, namely, the importance of focusing on correctness-centered quality attributes in our software QA effort for modern software systems.

2.5 SO, WHAT IS SOFTWARE QUALITY?

To conclude this chapter, we can answer the opening question, "What is software quality?" as follows:

- Software quality may include many different attributes and may be defined and perceived differently based on people's different roles and responsibilities.

- We adopt in this book the correctness-centered view of quality, that is, high quality means none or few problems of limited damage to customers. These problems are encountered by software users and caused by internal software defects.

The answer to a related question, "How do you ensure quality as defined above?" include many software QA and quality engineering activities to be described in the rest of this book.

Problems

2.1 What is software quality?

2.2 What is your view of software quality? What is your company's definition of quality? What other views not mentioned in Section 2.1 can you think of?

2.3 What is the relationship between quality, correctness, defects, and other "-ilities" (quality attributes)?

2.4 Define the following terms and give some concrete examples: defect, error, fault, failure, accident. What is the relationship among them? What about (software) bugs?

2.5 What is the pre-industrial concept of quality, and what is the future concept of quality? (Notice that we started with manufacturing in our historical perspective on quality.)

2.6 What is the relationship between quality, quality assurance, and quality engineering? What about between testing and quality?