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:

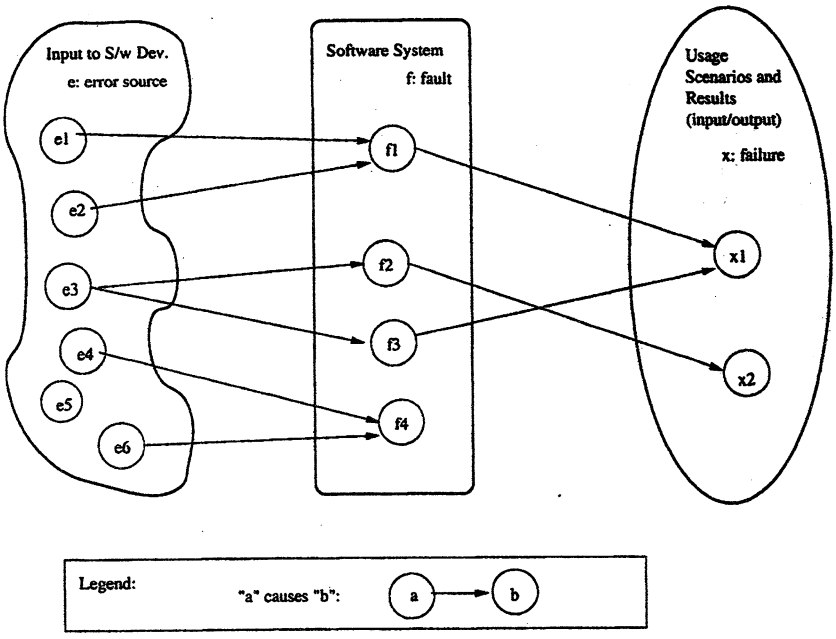

Figure 2.1 Defect related concepts and relations

- 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?

CHAPTER 3

QUALITY ASSURANCE

With the correctness-centered quality definitions adopted in the previous chapter for this book, the central activities for quality assurance (QA) can be viewed as to ensure that few, if any, defects remain in the software system when it is delivered to its customers or released to the market. Furthermore, we want to ensure that these remaining defects will cause minimal disruptions or damages. In this chapter, we survey existing QA alternatives and related techniques, and examine the specific ways they employ to deal with defects. Through this examination, we can abstract out several generic ways to deal with defects, which can then be used to classify these QA alternatives. Detailed descriptions and a general comparison of the related QA activities and techniques are presented in Part II and Part III.

3.1 CLASSIFICATION: QA AS DEALING WITH DEFECTS

A close examination of how different QA alternatives deal with defects can yield a generic classification scheme that can be used to help us better select, adapt and use different QA alternatives and related techniques for specific applications. We next describe a classification scheme initially proposed in Tian (2001) and illustrate it with examples.

A classification scheme

With the defect definitions given in the previous chapter, we can view different QA activities as attempting to prevent, eliminate, reduce, or contain various specific problems associated

with different aspects of defects. We can classify these QA alternatives into the following three generic categories:

- *Defect prevention through error blocking or error source removal*: These QA activities prevent certain types of faults from being injected into the software. Since errors are the missing or incorrect human actions that lead to the injection of faults into software systems, we can directly correct or block these actions, or remove the underlying causes for them. Therefore, defect prevention can be done in two generic ways:
  - *Eliminating certain error sources*, such as eliminating ambiguities or correcting human misconceptions, which are the root causes for the errors.
  - *Fault prevention or blocking* by directly correcting or blocking these missing or incorrect human actions. This group of techniques breaks the causal relation between error sources and faults through the use of certain tools and technologies, enforcement of certain process and product standards, etc.
- *Defect reduction through fault detection and removal*: These QA alternatives detect and remove certain faults once they have been injected into the software systems. In fact, most traditional QA activities fall into this category. For example,
  - Inspection directly detects and removes faults from the software code, design, etc.
  - Testing removes faults based on related failure observations during program execution.

  Various other means, based on either static analyses or observations of dynamic executions, can be applied to reduce the number of faults in a software system.
- *Defect containment through failure prevention and containment*: These containment measures focus on the failures by either containing them to local areas so that there are no global failures observable to users, or limiting the damage caused by software system failures. Therefore, defect containment can be done in two generic ways:
  - Some QA alternatives, such as the use of fault-tolerance techniques, break the causal relation between faults and failures so that local faults will not cause global failures, thus "tolerating" these local faults.
  - A related extension to fault-tolerance is containment measures to avoid catastrophic consequences, such as death, personal injury, and severe property or environmental damages, in case of failures. For example, failure containment for real-time control software used in nuclear reactors may include concrete walls to encircle and contain radioactive material in case of reactor melt-down due to software failures, in order to prevent damage to environment and people's health.

Dealing with pre-/post-release defects

Different QA alternatives can be viewed as a concerted effort to deal with errors, faults, or failures, in order to achieve the common goal of quality assurance and improvement. Defect prevention and defect reduction activities directly deal with the competing processes

of defect injection and removal during the software development process (Humphrey, 1995). They affect the defect contents, or the number of faults, in the finished software products by working to reduce the pre-release defect injections or to remove as many such defects as possible before product release. The faults left in the finished software products are often called "dormant defects", which may stay *dormant* for some time, but have the potential of causing problems to customers and users of the products — a situation that we would like to alleviate or avoid. Further analyses of different types of defects can be found in Chapter 20. Related techniques to identify high-risk areas for focused defect reduction and QA can be found in Chapter 21.

After product release, the failures observed and problems reported by customers and users also need to be fixed, which in turn, could lead to reduced defects and improved product quality. However, one cannot rely on these post-release problem reports and give up pre-release defect prevention and reduction activities, because the cost of fixing defects after product release is significantly higher than before product release due to the numerous installations. In addition, the damage to software vendors' reputation can be devastating. Controlled field testing, commonly referred to as "beta testing", and similar techniques discussed further in Chapter 12 have been suggested and used to complement pre-release QA activities. Related process issues are discussed in Chapter 4.

On the other hand, defect containment activities aim at minimizing the negative impact of these remaining faults during operational use after product release. However, most of the defect containment techniques involve redundancies or duplications, and require significantly more development effort to design and implement related features. Therefore, they are typically limited to the situations where in-field failures are associated with substantial damage, such as in corporate-wide database for critical data, global telecommunication networks, and various computer-controlled safety critical systems such as medical devices and nuclear reactors. The details about these issues can be found in Chapter 16.

Graphical depiction of the classification scheme

The above QA activity classification can be illustrated in Figure 3.1, forming a series of barriers represented by dotted broken lines. Each barrier removes or blocks defect sources, or prevents undesirable consequences. Specific information depicted includes:

- The barrier between the input to software development activities (left box) and the software system (middle box) represents defect prevention activities.
- The curved barrier between the software system (middle box) and the usage scenario and observed behavior (right box) represents defect or fault removal activities such as inspection and testing.
- The straight barrier to the right of and close to the above fault removal barrier represents failure prevention activities such as fault tolerance.
- The last barrier, surrounding selected failure instances, represents failure containment activities.

In Figure 3.1, faults are depicted as circled entities within the middle box for the software system. Error sources are depicted as circled entities within the left box for the input to the software development activities. Failures are depicted as the circled instances within the right box for usage scenarios and execution results. Figure 3.1 also shows the relationship

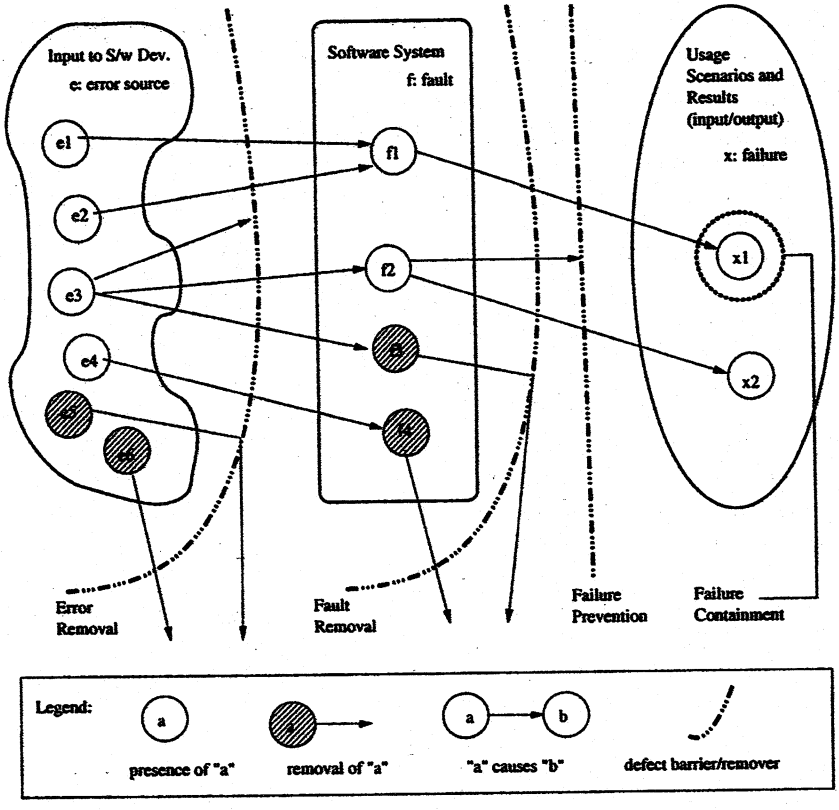

Figure 3.1 Generic ways to deal with defects

between these QA activities and related errors, faults, and failures through some specific examples, as follows:

- Some of the human conceptual errors, such as error source *e6*, are directly removed by error source removal activities, such as through better education to correct the specific human conceptual mistakes.

- Other incorrect actions or errors, such as some of those caused by error source *e3* and *e5*, are blocked. If an error source can be consistently blocked, such as *e5*, it is equivalent to being removed. On the other hand, if an error source is blocked sometimes, such as *e3*, additional or alternative defect prevention techniques need to be used, similar to the situation for other error sources such as *e1*, *e2*, and *e4*, where faults are likely to be injected into the software system because of these error sources.

- Some faults, such as *f4*, are detected directly through inspection or other static analysis and removed as a part of or as follow-up to these activities, without involving the observation of failures.

- Other faults, such as *f3*, are detected through testing or other execution-based QA alternatives by observing their dynamic behavior. If a failure is observed in these QA activities, the related faults are located by examining the execution record and

removed as a part of or as follow-up to these activities. Consequently, no operational failures after product release will be caused by these faults.

- Still other faults, such as *f2*, are blocked through fault tolerance for some execution instances. However, fault-tolerance techniques typically do not identify and fix the underlying faults. Therefore, these faults could still lead to operational failures under different dynamic environments, such as *f2* leading to *x2*.

- Among the failure instances, failure containment strategy may be applied for those with severe consequences. For example, *x1* is such an instance, where failure containment is applied to it, as shown by the surrounding dotted circle.

We next survey different QA alternatives, organized in the above classification scheme, and provide pointers to related chapters where they are described in detail.

3.2 DEFECT PREVENTION

The QA alternatives commonly referred to as defect prevention activities can be used for most software systems to reduce the chance for defect injections and the subsequent cost to deal with these injected defects. Most of the defect prevention activities assume that there are known error sources or missing/incorrect actions that result in fault injections, as follows:

- If human misconceptions are the error sources, education and training can help us remove these error sources.

- If imprecise designs and implementations that deviate from product specifications or design intentions are the causes for faults, formal methods can help us prevent such deviations.

- If non-conformance to selected processes or standards is the problem that leads to fault injections, then process conformance or standard enforcement can help use prevent the injection of related faults.

- If certain tools or technologies can reduce fault injections under similar environments, they should be adopted.

Therefore, root cause analyses described in Chapter 21 are needed to establish these preconditions, or *root causes*, for injected or potential faults, so that appropriate defect prevention activities can be applied to prevent injection of similar faults in the future. Once such causal relations are established, appropriate QA activities can then be selected and applied for defect prevention.

3.2.1 Education and training

Education and training provide people-based solutions for error source elimination. It has long been observed by software practitioners that the people factor is the most important factor that determines the quality and, ultimately, the success or failure of most software projects. Education and training of software professionals can help them control, manage, and improve the way they work. Such activities can also help ensure that they have few, if

any, misconceptions related to the product and the product development. The elimination of these human misconceptions will help prevent certain types of faults from being injected into software products. The education and training effort for error source elimination should focus on the following areas:

- *Product and domain specific knowledge.* If the people involved are not familiar with the product type or application domain, there is a good chance that wrong solutions will be implemented. For example, developers unfamiliar with embedded software may design software without considering its environmental constraints, thus leading to various interface and interaction problems between software and its physical surroundings.

- *Software development knowledge and expertise* plays an important role in developing high-quality software products. For example, lack of expertise with requirement analysis and product specification usually leads to many problems and rework in subsequent design, coding, and testing activities.

- *Knowledge about Development methodology, technology, and tools* also plays an important role in developing high-quality software products. For example, in an implementation of Cleanroom technology (Mills et al., 1987b), if the developers are not familiar with the key components of formal verification or statistical testing, there is little chance for producing high-quality products.

- *Development process knowledge.* If the project personnel do not have a good understanding of the development process involved, there is little chance that the process can be implemented correctly. For example, if the people involved in incremental software development do not know how the individual development efforts for different increments fit together, the uncoordinated development may lead to many interface or interaction problems.

3.2.2 Formal method

Formal methods provide a way to eliminate certain error sources and to verify the absence of related faults. Formal development methods, or formal methods in short, include formal specification and formal verification. Formal specification is concerned with producing an unambiguous set of product specifications so that customer requirements, as well as environmental constraints and design intentions, are correctly reflected, thus reducing the chances of accidental fault injections. Formal verification checks the conformance of software design or code against these formal specifications, thus ensuring that the software is fault-free with respect to its formal specifications.

Various techniques exist to specify and verify the "correctness" of software systems, namely, to answer the questions: "What is the correct behavior?", and "How to verify it?" We will describe some of these techniques in Chapter 15, with the basic ideas briefly introduced below.

- The oldest and most influential formal method is the so-call axiomatic approach (Hoare, 1969; Zelkowitz, 1993). In this approach, the "meaning" of a program element or the formal interpretation of the effect of its execution is abstracted into an axiom. Additional axioms and rules are used to connect different pieces together. A set of formal conditions describing the program state before the execution of a

program is called its *pre-conditions*, and the set after program execution the *post-conditions*. This approach verifies that a given program satisfies its prescribed pre- and post-conditions.

- Other influential formal verification techniques include the predicate transformer based on weakest precondition ideas (Dijkstra, 1975; Gries, 1987), and program calculus or functional approach heavily based on mathematical functions and symbolic executions (Mills et al., 1987a). The basic ideas are similar to the axiomatic approach, but the proof procedures are somewhat different.
- Various other limited scope or semi-formal techniques also exist, which check for certain properties instead of proving the full correctness of programs. For example, model checking techniques are gaining popularity in the software engineering research community (Ghezzi et al., 2003). Various semi-formal methods based on forms or tables, such as (Parnas and Madey, 1995), instead of formal logic or mathematical functions, have found important applications as well.

So far, the biggest obstacle to formal methods is the high cost associated with the difficult task of performing these human intensive activities correctly without adequate automated support. This fact also explains, to a degree, the increasing popularity of limited scope and semi-formal approaches.

3.2.3 Other defect prevention techniques

Other defect prevention techniques, to be described in Chapter 13, including those based on technologies, tools, processes, and standards, are briefly introduced below:

- Besides the formal methods surveyed above, appropriate use of other software methodologies or technologies can also help reduce the chances of fault injections. Many of the problems with low quality "fat software" could be addressed by disciplined methodologies and return to essentials for high-quality "lean software" (Wirth, 1995). Similarly, the use of the information hiding principle (Parnas, 1972) can help reduce the complexity of program interfaces and interactions among different components, thus reducing the possibility of related problems.
- A better managed process can also eliminate many systematic problems. For example, not having a defined process or not following it for system configuration management may lead to inconsistencies or interface problems among different software components. Therefore, ensuring appropriate process definition and conformance helps eliminate some such error sources. Similarly, enforcement of selected standards for certain types of products and development activities also reduces fault injections.
- Sometimes, specific software tools can also help reduce the chances of fault injections. For example, a syntax-directed editor that automatically balances out each open parenthesis, "{", with a close parenthesis, "}", can help reduce syntactical problems in programs written in the C language.

Additional work is needed to guide the selection of appropriate processes, standards, tools, and technologies, or to tailor existing ones to fit the specific application environment. Effective monitoring and enforcement systems are also needed to ensure that the selected processes or standards are followed, or the selected tools or technologies are used properly, to reduce the chance of fault injections.

3.3 DEFECT REDUCTION

For most large software systems in use today, it is unrealistic to expect the defect prevention activities surveyed above to be 100% effective in preventing accidental fault injections. Therefore, we need effective techniques to remove as many of the injected faults as possible under project constraints.

3.3.1 Inspection: Direct fault detection and removal

Software inspections are critical examinations of software artifacts by human inspectors aimed at discovering and fixing faults in the software systems. Inspection is a well-known QA alternative familiar to most experienced software quality professionals. The earliest and most influential work in software inspection is Fagan inspection (Fagan, 1976). Various other variations have been proposed and used to effectively conduct inspection under different environments. A detailed discussion about inspection processes and techniques, applications and results, and many related topics can be found in Chapter 14. The basic ideas of inspection are outlined below:

- Inspections are critical reading and analysis of software code or other software artifacts, such as designs, product specifications, test plans, etc.
- Inspections are typically conducted by multiple human inspectors, through some coordination process. Multiple inspection phases or sessions might be used.
- Faults are detected directly in inspection by human inspectors, either during their individual inspections or various types of group sessions.
- Identified faults need to be removed as a result of the inspection process, and their removal also needs to be verified.
- The inspection processes vary, but typically include some planning and follow-up activities in addition to the core inspection activity.
- The formality and structure of inspections may vary, from very informal reviews and walkthroughs, to fairly formal variations of Fagan inspection, to correctness inspections approaching the rigor and formality of formal methods.

Inspection is most commonly applied to code, but it could also be applied to requirement specifications, designs, test plans and test cases, user manuals, and other documents or software artifacts. Therefore, inspection can be used throughout the development process, particularly early in the software development before anything can be tested. Consequently, inspection can be an effective and economical QA alternative because of the much increased cost of fixing late defects as compared to fixing early ones.

Another important potential benefit of inspection is the opportunity to conduct causal analysis during the inspection process, for example, as an added step in Gilb inspection (Gilb and Graham, 1993). These causal analysis results can be used to guide defect prevention activities by removing identified error sources or correcting identified missing/incorrect human actions. These advantages of inspection will be covered in more detail in Chapter 14 and compared to other QA alternatives in Chapter 17.

3.3.2 Testing: Failure observation and fault removal

Testing is one of the most important parts of QA and the most commonly performed QA activity. Testing involves the execution of software and the observation of the program behavior or outcome. If a failure is observed, the execution record is then analyzed to locate and fix the fault(s) that caused the failure. As a major part of this book, various issues related to testing and commonly used testing techniques are covered in Part II (Chapters 6 through 12).

Individual testing activities and techniques can be classified using various criteria and examined accordingly, as discussed below. Here we pay special attention to how they deal with defects. A more comprehensive classification scheme is presented in Chapter 6.

When can a specific testing activity be performed and related faults be detected?

Because testing is an execution-based QA activity, a prerequisite to actual testing is the existence of the implemented software units, components, or system to be tested, although preparation for testing can be carried out in earlier phases of software development. As a result, actual testing can be divided into various sub-phases starting from the coding phase up to post-release product support, including: unit testing, component testing, integration testing, system testing, acceptance testing, beta testing, etc. The observation of failures can be associated with these individual sub-phases, and the identification and removal of related faults can be associated with corresponding individual units, components, or the complete system.

If software prototypes are used, such as in the spiral process, or if a software system is developed using an incremental or iterative process, testing can usually get started much earlier. Later on, integration testing plays a much more important role in detecting inter-operability problems among different software components. This issue is discussed further in Chapter 4, in connection to the distribution of QA activities in the software processes.

What to test, and what kind of faults are found?

Black-box (or functional) testing verifies the correct handling of the external functions provided by the software, or whether the observed behavior conforms to user expectations or product specifications. White-box (or structural) testing verifies the correct implementation of internal units, structures, and relations among them. Various techniques can be used to build models and generate test cases to perform systematic black-box or white-box testing.

When black-box testing is performed, failures related to specific external functions can be observed, leading to corresponding faults being detected and removed. The emphasis is on reducing the chances of encountering functional problems by target customers. On the other hand, when white-box testing is performed, failures related to internal implementations can be observed, leading to corresponding faults being detected and removed. The emphasis is on reducing internal faults so that there is less chance for failures later on no matter what kind of application environment the software is subjected to.

When, or at what defect level, to stop testing?

Most of the traditional testing techniques and testing sub-phases use some kind of coverage information as the stopping criteria, with the implicit assumption that higher coverage

means higher quality or lower levels of defects. For example, checklists are often used to make sure major functions and usage scenarios are tested before product release. Every statement or unit in a component must be covered before subsequent integration testing can proceed. More formal testing techniques include control flow testing that attempts to cover execution paths and domain testing that attempts to cover boundaries between different input sub-domains. Such formal coverage information can only be obtained by using expensive coverage analysis and testing tools. However, rough coverage measurement can be obtained easily by examining the proportion of tested items in various checklists.

On the other hand, product reliability goals can be used as a more objective criterion to stop testing. The use of this criterion requires the testing to be performed under an environment that resembles actual usage by target customers so that realistic reliability assessment can be obtained, resulting in the so-called usage-based statistical testing.

The coverage criterion ensures that certain types of faults are detected and removed, thus reducing the number of defects to a lower level, although quality is not directly assessed. The usage-based testing and the related reliability criterion ensure that the faults that are most likely to cause problems to customers are more likely to be detected and removed, and the reliability of the software reaches certain targets before testing stops.

3.3.3 Other techniques and risk identification

Inspection is the most commonly used static techniques for defect detection and removal. Various other static techniques are available, including various formal model based analyses such as algorithm analysis, decision table analysis, boundary value analysis, finite-state machine and Petri-net modeling, control and data flow analyses, software fault trees, etc.

Similarly, in addition to testing, other dynamic, execution-based, techniques also exist for fault detection and removal. For example, symbolic execution, simulation, and prototyping can help us detect and remove various defects early in the software development process, before large-scale testing becomes a viable alternative.

On the other hand, in-field measurement and related analyses, such as timing and performance analysis for real-time systems, and accident analysis and reconstruction using software fault trees and event trees for safety-critical systems, can also help us locate and remove related defects. Although these activities are an important part of product support, they are not generally considered as a part of the traditional QA activities because of the damages already done to the customers' applications and to the software vendors' reputation. As mentioned in Section 3.1, because of the benefits of dealing with problems before product release instead of after product release, the focus of these activities is to provide useful information for future QA activities.

A comprehensive survey of techniques for fault detection and removal can be found in Chapters 6 and 14, in connection with testing and inspection techniques. Related techniques for dealing with post-release defects are covered in Chapter 16 in connection with fault tolerance and failure containment techniques.

Fault distribution is highly uneven for most software products, regardless of their size, functionality, implementation language, and other characteristics. Much empirical evidence has accumulated over the years to support the so-called 80:20 rule, which states that 20% of the software components are responsible for 80% of the problems. These problematic components can generally be characterized by specific measurement properties about their design, size, complexity, change history, and other product or process characteristics. Because of the uneven fault distribution among software components, there is a great need for risk identification techniques to analyze these measurement data so that inspection, testing,

and other QA activities can be more effectively focused on those potentially high-defect components.

These risk identification techniques are described in Chapter 21, including: traditional statistical analysis techniques, principal component analysis and discriminant analysis, neural networks, tree-based modeling, pattern matching techniques, and learning algorithms. These techniques are compared according to several criteria, including: accuracy, simplicity, early availability and stability, ease of result interpretation, constructive information and guidance for quality improvement, and availability of tool support. Appropriate risk identification techniques can be selected to fit specific application environments in order to identify high-risk software components for focused inspection and testing.

3.4 DEFECT CONTAINMENT

Because of the large size and high complexity of most software systems in use today, the above defect reduction activities can only reduce the number of faults to a fairly low level, but not completely eliminate them. For software systems where failure impact is substantial, such as many real-time control software sub-systems used in medical, nuclear, transportation, and other embedded systems, this low defect level and failure risk may still be inadequate. Some additional QA alternatives are needed.

On the other hand, these few remaining faults may be triggered under rare conditions or unusual dynamic scenarios, making it unrealistic to attempt to generate the huge number of test cases to cover all these conditions or to perform exhaustive inspection based on all possible scenarios. Instead, some other means need to be used to prevent failures by breaking the causal relations between these faults and the resulting failures, thus "tolerating" these faults, or to contain the failures by reducing the resulting damage.

3.4.1 Software fault tolerance

Software fault tolerance ideas originate from fault tolerance designs in traditional hardware systems that require higher levels of reliability, availability, or dependability. In such systems, spare parts and backup units are commonly used to keep the systems in operational conditions, maybe at a reduced capability, at the presence of unit or part failures. The primary software fault tolerance techniques include recovery blocks, N-version programming (NVP), and their variations (Lyu, 1995b). We will describe these techniques and examine how they deal with failures and related faults in Chapter 16, with the basic ideas summarized below:

- Recovery blocks use repeated executions (or redundancy over time) as the basic mechanism for fault tolerance. If dynamic failures in some local areas are detected, a portion of the latest execution is repeated, in the hope that this repeated execution will not lead to the same failure. Therefore, local failures will not propagate to global failures, although some time-delay may be involved.

- NVP uses parallel redundancy, where N copies, each of a different version, of programs fulfilling the same functionality are running in parallel. The decision algorithm in NVP makes sure that local failures in limited number of these parallel versions will not compromise global execution results.

One fact worth noting is that in most fault tolerance techniques, faults are not typically identified, therefore not removed, but only tolerated dynamically. This is in sharp contrast to defect detection and removal activities such as inspection and testing.

3.4.2 Safety assurance and failure containment

For safety critical systems, the primary concern is our ability to prevent accidents from happening, where an accident is a failure with a severe consequence. Even low failure probabilities for software are not tolerable in such systems if these failures may still likely lead to accidents. Therefore, in addition to the above QA techniques, various specific techniques are also used for safety critical systems based on analysis of hazards, or logical pre-conditions for accidents (Leveson, 1995). These safety assurance and improvement techniques are covered in Chapter 16. A brief analysis of how each of them deals with defects is given below:

- *Hazard elimination* through substitution, simplification, decoupling, elimination of specific human errors, and reduction of hazardous materials or conditions. These techniques reduce certain defect injections or substitute non-hazardous ones for hazardous ones. The general approach is similar to the defect prevention and defect reduction techniques surveyed earlier, but with a focus on those problems involved in hazardous situations.
- *Hazard reduction* through design for controllability (for example, automatic pressure release in boilers), use of locking devices (for example, hardware/software interlocks), and failure minimization using safety margins and redundancy. These techniques are similar to the fault tolerance techniques surveyed above, where local failures are contained without leading to system failures.
- *Hazard control* through reducing exposure, isolation and containment (for example, barriers between the system and the environment), protection systems (active protection activated in case of hazard), and fail-safe design (passive protection, fail in a safe state without causing further damages). These techniques reduce the severity of failures, therefore weakening the link between failures and accidents.
- *Damage control* through escape routes, safe abandonment of products and materials, and devices for limiting physical damages to equipments or people. These techniques reduce the severity of accidents, thus limiting the damage caused by these accidents and related software failures.

Notice that both hazard control and damage control above are post-failure activities that attempt to "contain" the failures so that they will not lead to accidents or the accident damage can be controlled or minimized. These activities are specific to safety critical systems, which are not generally covered in the QA activities for other systems. On the other hand, many techniques for defect prevention, reduction, and tolerance can also be used in safety-critical systems for hazard elimination and reductions through focused activities on safety-critical product components or features.

3.5 CONCLUDING REMARKS

According to the different ways different QA alternatives deal with defects, they can be classified into three general categories:

- *Defect prevention* through error source elimination and error blocking activities, such as education and training, formal specification and verification, and proper selection and application of appropriate technologies, tools, processes, or standards. The detailed descriptions of these specific techniques and related activities are given in Chapter 15 for formal verification techniques and in Chapter 13 for the rest.
- *Defect reduction* through inspection, testing, and other static analyses or dynamic activities, to detect and remove faults from software. As one of the most important and widely used alternatives, testing is described in Part II (Chapters 6 through 12). Related dynamic analysis is also described in Chapter 12. The other important alternative, inspection, is described in Chapter 14, where a brief description of related static analysis techniques is also included.
- *Defect containment* through fault tolerance, failure prevention, or failure impact minimization, to assure software reliability and safety. The detailed description of these specific techniques and related activities is given in Chapter 16.

Existing software quality literature generally covers defect reduction techniques such as testing and inspection in more details than defect prevention activities, while largely ignore the role of defect containment in QA. This chapter brings together information from diverse sources to offer a common starting point and information base for software quality professionals and software engineering students. Follow-up chapters describe each specific alternative in much more detail and offer a comprehensive coverage of important techniques for QA as well as integration of QA activities into the overall software development and maintenance process.

Problems

3.1 What is quality assurance?

3.2 What are the different types of QA activities? Do you know any classification other than the one described in this chapter based on how they deal with defects?

3.3 For the product your are working on, which QA strategy is used? What other QA strategies and techniques might be applicable or effective?

3.4 Can you use the QA strategies and techniques described in this chapter to deal with other problems, not necessarily defect-related problems, such as usability, performance, modifiability? In addition, can you generalize the QA activities described in this chapter to deal with defects related to things other than software?

3.5 Formal methods are related to both defect prevention and defect detection/removal. Can you think of other QA activities that cut across multiple categories in our classification of QA activities into defect prevention, reduction, and containment.

3.6 What are the similarities and differences between items in the following pairs:

a) software testing and hardware testing

b) software inspection and inspection of other things (for example, car inspection, house inspection, inspection for weapons-of-mass-destruction)

c) quality assurance and safety assurance

CHAPTER 5

QUALITY ENGINEERING

In this chapter, we enlarge the scope of our discussion to include other major activities associated with quality assurance (QA) for software systems, primarily in the areas of setting quality goals, planning for QA, monitoring QA activities, and providing feedback for project management and quality improvement.

5.1 QUALITY ENGINEERING: ACTIVITIES AND PROCESS

As stated in Chapter 2, different customers and users have different quality expectations under different market environments. Therefore, we need to move beyond just performing QA activities toward quality engineering by managing these quality expectations as an engineering problem: Our goal is to meet or exceed these quality expectations through the selection and execution of appropriate QA activities while minimizing the cost and other project risks under the project constraints.

In order to ensure that these quality goals are met through the selected QA activities, various measurements need to be taken parallel to the QA activities themselves. Post-mortem data often need to be collected as well. Both in-process and post-mortem data need to be analyzed using various models to provide an objective quality assessment. Such quality assessments not only help us determine if the preset quality goals have been achieved, but also provide us with information to improve the overall product quality.

To summarize, there are three major groups of activities in the quality engineering process, as depicted in Figure 5.1. They are labeled in roughly chronological order as pre-QA activities, in-QA activities, and post-QA activities:

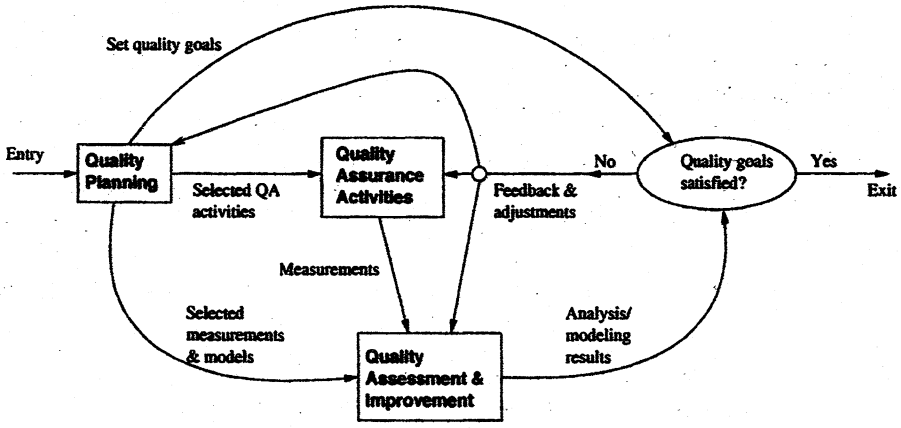

Figure 5.1 Quality engineering process

1. *Pre-QA activities: Quality planning.* These are the activities that should be carried out before carrying out the regular QA activities. There are two major types of pre-QA activities in quality planning, including:

   (a) Set specific quality goals.

   (b) Form an overall QA strategy, which includes two sub-activities:

      i. Select appropriate QA activities to perform.

      ii. Choose appropriate quality measurements and models to provide feedback, quality assessment and improvement.

   A detailed description of these pre-QA activities is presented in Section 5.2.

2. *In-QA activities: Executing planned QA activities and handling discovered defects.* In addition to performing selected QA activities, an important part of this normal execution is to deal with the discovered problems. These activities were described in the previous two chapters.

3. *Post-QA activities: Quality measurement, assessment and improvement* These are the activities that are carried out after normal QA activities have started but not as part of these normal activities. The primary purpose of these activities is to provide quality assessment and feedback so that various management decisions can be made and possible quality improvement initiatives can be carried out. These activities are described in Section 5.3.

Notice here that "post-QA" does not mean after the finish of QA activities. In fact, many of the measurement and analysis activities are carried out parallel to QA activities after they are started. In addition, pre-QA activities may overlap with the normal QA activities as well.

Pre-QA quality planning activities play a leading role in this quality engineering process, although the execution of selected QA activities usually consumes the most resources. Quality goals need to be set so that we can manage the QA activities and stop them when the quality goals are met. QA strategies need to be selected, before we can carry out specific QA activities, collect data, perform analysis, and provide feedback.

There are two kinds of feedback in this quality engineering process, both the short term direct feedback to the QA activities and the long-term feedback to the overall quality engineering process. The short term feedback to QA activities typically provides information for progress tracking, activity scheduling, and identification of areas that need special attentions. For example, various models and tools were used to provide test effort tracking, reliability monitoring, and identification of low-reliability areas for various software products developed in the IBM Software Solutions Toronto Lab to manage their testing process (Tian, 1996).

The long-term feedback to the overall quality engineering process comes in two forms:

- Feedback to quality planning so that necessary adjustment can be made to quality goals and QA strategies. For example, if the current quality goals are unachievable, alternative goals need to be negotiated. If the selected QA strategy is inappropriate, a new or modified strategy needs to be selected. Similarly, such adjustments may also be applied to future projects instead of the current project.
- Feedback to the quality assessment and improvement activities. For example, the modeling results may be highly unstable, which may well be an indication of the model inappropriateness. In this case, new or modified models need to be used, probably on screened or pre-processed data.

Quality engineering and QIP

In the TAME project and related work (Basili and Rombach, 1988; Oivo and Basili, 1992; Basili, 1995; van Solingen and Berghout, 1999), quality improvement was achieved through measurement, analysis, feedback, and organizational support. The overall framework is called QIP, or quality improvement paradigm. QIP includes three interconnected steps: understanding, assessing, and packaging, which form a feedback and improvement loop, as briefly described below:

1. The first step is to *understand* the baseline so that improvement opportunities can be identified and clear, measurable goals can be set. All future process changes are measured against this baseline.
2. The second step is to introduce process changes through experiments, pilot projects, *assess* their impact, and fine tune these process changes.
3. The last step is to *package* baseline data, experiment results, local experience, and updated process as the way to infuse the findings of the improvement program into the development organization.

QIP and related work on measurement selection and organizational support are described further in connection to defect prevention in Chapter 13 and in connection to quality assessment and improvement in Part IV.

Our approach to quality engineering can be considered as an adaptation of QIP to assure and measure quality, and to manage quality expectations of target customers. Some specific correspondences are noted below:

- Our pre-QA activities roughly correspond to the *understand* step in QIP.
- The execution of our selected QA strategies correspond to the "changes" introduced in the *assess* step in QIP. However, we are focusing on the execution of normal QA

activities and the related measurement activities selected previously in our planning step, instead of specific changes.

- Our analysis and feedback (or post-QA) activities overlap with both the *assess* and *package* steps in QIP, with the analysis part roughly corresponding to the QIP-assess step and the longer term feedback roughly corresponding to the QIP-package step.

5.2 QUALITY PLANNING: GOAL SETTING AND STRATEGY FORMATION

As mentioned above, pre-QA quality planning includes setting quality goals and forming a QA strategy. The general steps include:

1. Setting quality goals by matching customer's quality expectations with what can be economically achieved by the software development organizations in the following sub-steps:
   (a) Identify quality views and attributes meaningful to target customers and users.
   (b) Select direct quality measures that can be used to measure the selected quality attributes from customer's perspective.
   (c) Quantify these quality measures to set quality goals while considering the market environment and the cost of achieving different quality goals.
2. In forming a QA strategy, we need to plan for its two basic elements:
   (a) Map the above quality views, attributes, and quantitative goals to select a specific set of QA alternatives.
   (b) Map the above external direct quality measures into internal indirect ones via selected quality models. This step selects indirect quality measures as well as usable models for quality assessment and analysis.

We next examine these steps and associated pre-QA activities in detail.

Setting quality goals

One important fact in managing customer's quality expectations is that different quality attributes may have different levels of importance to different customers and users. Relevant quality views and attributes need to be identified first. For example, reliability is typically the primary concern for various business and commercial software systems because of people's reliance on such systems and the substantial financial loss if they are malfunctioning. Similarly, if a software is used in various real-time control situations, such as air traffic control software and embedded software in automobile, medical devices, etc., accidents due to failures may be catastrophic. Therefore, safety is the major concern. On the other hand, for mass market software packages, such as various auxiliary utilities for personal computers, usability, instead of reliability or safety, is the primary concern.

Even in the narrower interpretation of quality we adopted in this book to be the correctness-centered quality attributes associated with errors, faults, failures, and accidents, there are different types of problems and defects that may mean different things to different customers. For example, for a software product that is intended for diverse operational environments, inter-operability problems may be a major concern to its customers and users; while the

same problems may not be a major concern for software products with a standard operational environment. Therefore, specific quality expectations by the customers require us to identify relevant quality views and attributes prior to setting appropriate quality goals. This needs to be done in close consultation with the customers and users, or those who represents their interests, such as requirement analysts, marketing personnel, etc.

Once we obtained qualitative knowledge about customers' quality expectations, we need to quantify these quality expectations to set appropriate quality goals in two steps:

1. *We need to select or define the quality measurements and models* commonly accepted by the customers and in the software engineering community. For example, as pointed out in Chapter 2, reliability and safety are examples of correctness-centered quality measures that are meaningful to customers and users, which can be related to various internal measures of faults commonly used within software development organizations.

2. *We need to find out the expected values* or ranges of the corresponding quality measurements. For example, different market segments might have different reliability expectations. Such quality expectations are also influenced by the general market conditions and competitive pressure.

Software vendors not only compete on quality alone, but also on cost, schedule, innovation, flexibility, overall user experience, and other features and properties as well. Zero defect is not an achievable goal under most circumstances, and should not be the goal. Instead, zero defection and positive flow of new customers and users based on quality expectation management should be a goal (Reichheld Jr. and Sasser, 1990). In a sense, this activity determines to a large extent the product positioning vs. competitors in the marketplace and potential customers and users.

Another practical concern with the proper setting of quality goals is the cost associated with different levels of quality. This cost can be divided into two major components, the failure cost and the development cost. The customers typically care more about the total failure cost, C_f, which can be estimated by the average single failure cost, c_f, and failure probability, p_f, over a pre-defined duration of operation as:

$$C_f = c_f \times p_f.$$

As we will see later in Chapter 22, this failure probability can be expressed in terms of reliability, R, as $p_f = 1 - R$, where R is defined to be the probability of failure-free operations for a specific period of given set of input.

To minimize C_f, one can either try to minimize c_f or p_f. However, c_f is typically determined by the nature of software applications and the overall environment the software is used in. Consequently, not much can be done about c_f reduction without incurring substantial amount of other cost. One exception to this is in the safety critical systems, where much additional cost was incurred to establish barriers and containment in order to reduce failure impact, as described in Chapter 16. On the other hand, minimizing p_f, or improving reliability, typically requires additional development cost, in the form of additional testing time, use of additional QA techniques, etc.

Therefore, an engineering decision need to be made to match the quantified customer's quality expectations above with their willingness to pay for the quality. Such quantitative cost-of-quality analyses should help us reach a set of quality goals.

Forming a QA strategy

Once specific quality goals were set, we can select appropriate QA alternatives as part of a QA strategy to achieve these goals. Several important factors need to be considered:

- *The influence of quality perspectives and attributes:* For different kinds of customers, users, and market segments, different QA alternatives might be appropriate, because they focus on the assurance of quality attributes based on this specific perspective. For example, various usability testing techniques may be useful for ensuring the usability of a software product, but may not be effective for ensuring its functional correctness.

- *The influence of different quality levels:* Quantitative quality levels as specified in the quality goals may also affect the choice of appropriate QA techniques. For example, systems with various software fault tolerance features may incur substantially more additional cost than the ones without them. Therefore, they may be usable for highly dependable systems or safety critical systems, where large business operations and people's lives may depend on the correct operations of software systems, but may not be suitable for less critical software systems that only provide non-essential information to the users.

Notice that in dealing with both of the above factors, we assume that there is a certain relationship between these factors and specific QA alternatives. Therefore, specific QA alternatives need to be selected to fulfill specific quality goals based on the quality perspectives and attributes of concern to the customers and users.

Implicitly assumed in this selection process is a good understanding of the advantages and disadvantages of different QA alternatives under different application environments. These comparative advantages and disadvantages are the *other factors* that also need to be considered in selecting different QA alternatives and related techniques and activities. These factors include cost, applicability to different environments, effectiveness in dealing with different kinds of problems, etc. discussed in Chapter 17.

In order to achieve the quality goals, we also need to know where we are and how far away we are from the preset quality goals. To gain this knowledge, objective assessment using some quality models on collected data from the QA activities is necessary. As we will discuss in more detail in Chapter 18, there are direct quality measures and indirect quality measures. The direct quality measures need to be defined as part of the activities to set quality goals, when such goals are quantified.

Under many situations, direct quality measures cannot be obtained until it is already too late. For example, for safety critical systems, post-accident measurements provide a direct measure of safety. But due to the enormous damage associated with such accidents, we are trying to do everything to avoid such accidents. To control and monitor these safety assurance activities, various indirect measurements and indicators can be used. For all software systems there is also an increasing cost of fixing problems late instead of doing so early in general, because a hidden problem may lead to other related problems, and the longer it stays undiscovered in the system, the further removed it is from its root causes, thus making the discovery of it even more difficult. Therefore, there is a strong incentive for early indicators of quality that usually measure quality indirectly.

Indirect quality measures are those which can be used in various quality models to assess and predict quality, through their established relations to direct quality measures based on historical data or data from other sources. Therefore, we also need to choose appropriate measurements, both direct and indirect quality measurement, and models to provide quality

assessment and feedback. The actual measurement and analysis activities and related usage of analysis results are discussed in Chapter 18.

5.3 QUALITY ASSESSMENT AND IMPROVEMENT

Various parallel and post-QA activities are carried out to close the quality engineering loop. The primary purpose of these activities is to provide quality assessment and feedback so that various management decisions, such as product release, can be made and possible quality and process improvement initiatives can be carried out. The major activities in this category include:

- *Measurement:* Besides defect measurements collected during defect handling, which is typically carried out as part of the normal QA activities, various other measurements are typically needed for us to track the QA activities as well as for project management and various other purposes. These measurements provide the data input to subsequent analysis and modeling activities that provide feedback and useful information to manage software project and quality.
- *Analysis and modeling:* These activities analyze measurement data from software projects and fit them to analytical models that provide quantitative assessment of selected quality characteristics or sub-characteristics. Such models can help us obtain an objective assessment of the current product quality, accurate prediction of the future quality, and some models can also help us identify problematic areas.
- *Providing feedback and identifying improvement potentials:* Results from the above analysis and modeling activities can provide feedback to the quality engineering process to help us make project scheduling, resource allocation, and other management decisions. When problematic areas are identified by related models, appropriate remedial actions can be applied for quality and process improvement.
- *Follow-up activities:* Besides the immediate use of analysis and modeling results described above, various follow-up activities can be carried out to affect the long-term quality and organizational performance. For example, if major changes are suggested for the quality engineering process or the software development process, they typically need to wait until the current process is finished to avoid unnecessary disturbance and risk to the current project.

The details about these activities are described in Part IV.

5.4 QUALITY ENGINEERING IN SOFTWARE PROCESSES

The quality engineering process forms an integral part of the overall software engineering process, where other concerns, such as cost and schedule, are also considered and managed. As described in Chapter 4, individual QA activities can be carried out and integrated into the software process. When we broaden our scope to quality engineering, it also covers pre-QA quality planning as well as the post-QA measurement and analysis activities carried out parallel to and after QA activities to provide feedback and other useful information. All these activities and the quality engineering process can be integrated into the overall software process as well, as described below.

Activity distribution and integration

Pre-QA quality planning can be an integral part of any project planning. For example, in the waterfall process, this is typically carried out in the phase for market analysis, requirement gathering, and product specification. Such activities also provide us with valuable information about quality expectations by target customers and users in the specific market segment a software vendor is prepared to compete in. Quality goals can be planned and set accordingly. Project planning typically includes decisions on languages, tools, and technologies to be used for the intended software product. It should be expanded to include 1) choices of specific QA strategies and 2) measurement and models to be used for monitoring the project progress and for providing feedback.

In alternative software processes other than waterfall, such as in incremental, iterative, spiral, and extreme programming processes, pre-QA activities play an even more active role, because they are not only carried out at the beginning of the whole project, but also at the beginning of each subpart or iteration due to the nature that each subpart includes more or less all the elements in the waterfall phases. Therefore, we need to set specific quality goals for each subpart, and choose appropriate QA activities, techniques, measurement, and models for each subpart. The overall quality goal may evolve from these sub-goals in an iterative fashion.

For normal project monitoring and management under any process, appropriate measurement activities need to be carried out to collect or extract data from the software process and related artifacts; analyses need to be performed on these data; and management decision can be made accordingly. On the one hand, the measurement activity cannot be carried out without the involvement of the software development team, either as part of the normal defect handling and project tracking activities, or as added activity to provide specific input to related analysis and modeling. Therefore, the measurement activities have to be handled "on-line" during the software development process, with some additional activities in information or measurement extraction carried out after the data collection and recording are completed.

On the other hand, much of the analysis and modeling activities could be done "off-line", to minimize the possible disruption or disturbance to the normal software development process. However, timely feedback based on the results from such analyses and models is needed to make adjustments to the QA and to the development activities. Consequently, even such "off-line" activities need to be carried out in a timely fashion, but may be at a lower frequency. For example, in the implementation of testing tracking, measurement, reliability analysis, and feedback for IBM's software products (Tian, 1996), dedicated quality analyst performed such analyses and modeling and provided weekly feedback to the testing team, while the data measurement and recording were carried out on a daily basis.

The specific analysis, feedback, and follow-up activities in the software quality engineering process fit well into the normal software management activities. Therefore, they can be considered as an integral part of software project management. Of course, the focus of these quality engineering activities is on the quality management, as compared to the overall project management that also includes managing project features, cost, schedule, and so on.

The integration of the quality engineering process into the waterfall software development process can be illustrated by Figure 5.2. The horizontal activities roughly illustrate the timeline correspondence to software development activities. For example, quality planning starts right at the start of the requirement analysis phase, followed by the execution of the selected QA activities, and finally followed by the measurement and analysis activities.

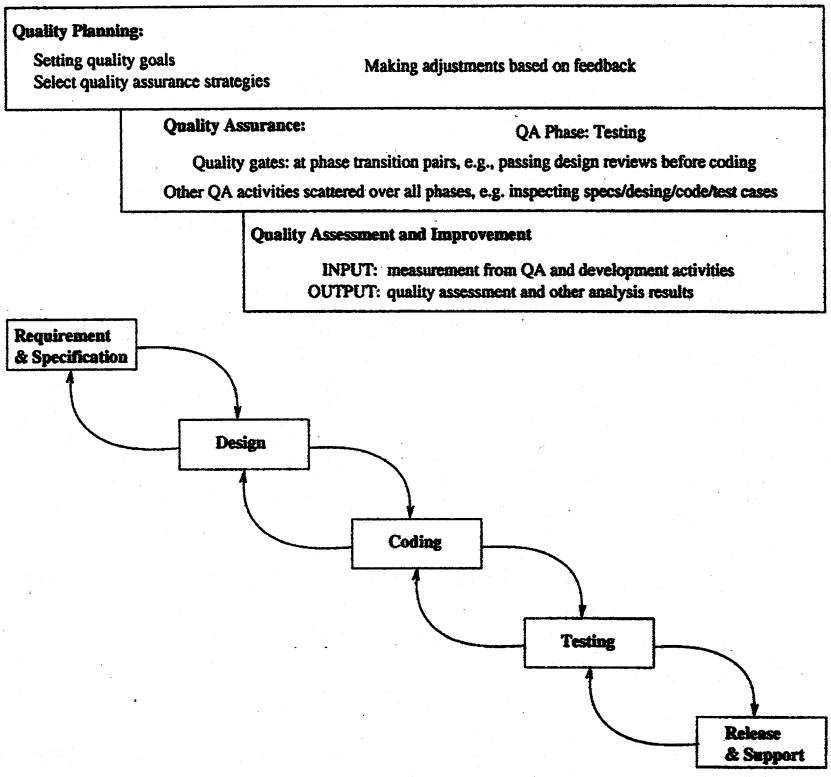

Figure 5.2 Quality engineering in the waterfall process

All these activities typically last over the whole development process, with different sub-activities carried out in different phases. This is particularly true for the QA activities, with testing in the test phase, various reviews or inspections at the transition from one phase to its successor phase, and other QA activities scattered over other phases.

Minor modifications are needed to integrate quality engineering activities into other development processes. However, the distribution of these activities and related effort is by no means uniform over the activities or over time, which is examined next.

Effort profile

Among the three major types of activities in the quality engineering process, the execution of specific QA activities is central to dealing with defects and assuring quality for the software products. Therefore, they should and normally do consume the most resources in terms of human effort as well as utilization of computing and other related resources. However, the effort distribution among the three is not constant over time because of the process characteristics described above and the shifting focus over time. Some key factors that affect and characterize the effort profile, or the effort distribution over time, include:

- Quality planning drives and should precede the other two groups of activities. Therefore, at the beginning part of product development, quality planning should be the dominant part of quality engineering activities. Thereafter, occasional adjustments

to the quality goals and selected quality strategies might be applied, but only a small share of effort is needed.

- The collective effort of selected QA activities generally demonstrates the following pattern:
  - There is a gradual build-up process for individual QA activities, and for them collectively.
  - The collective effort normally peaks off a little bit before product release, when development activities wind down and testing activities are the dominant activities.
  - Around product release and thereafter, the effort tapers off, typically with a sudden drop at product release.

  Of course, the specific mix of selected QA activities as well as the specific development process used would affect the shape of this effort profile as well. But the general pattern is expected to hold.

- Measurement and quality assessment activities start after selected QA activities are well under way. Typically, at the early part of the development process, small amounts of such activities are carried out to monitor quality progress. But they are not expected to be used to make major management decisions such as product release. These activities peak off right before or at the product release, and lower gradually after that. In the overall shape and pattern, the effort profile for these activities follows that for the collective QA activities above, but with a time delay and a heavier load at the tail-end.

One common adjustment to the above pattern is the time period after product release. Immediately after product release or after a time delay for market penetration, the initial wave of operational use by customers is typically accompanied by many user-reported problems, which include both legitimate failures and user errors. Consequently, there is typically an upswing of overall QA effort. New data and models are also called for, resulting in an upswing of measurement and analysis activities as well. The main reason for this upswing is the difference between the environment where the product is tested under and the actual operational environment the product is subjected to. The use of usage-based testing described in Chapters 8 and 10 would help make this bump smoother.

This general profile can be graphically illustrated in Figure 5.3. The overall quality engineering effort over time is divided into three parts:

- The bottom part represents the share of total effort by quality planning activities;
- The middle part represents the share of total effort for the execution of selected QA activities;
- The upper part represents the share of total effort for the measurement and quality assessment activities.

Notice that this figure is for illustration purposes only. The exact profile based on real data would not be as smooth and would naturally show large amount of variability, with many small peaks and valleys. But the general shape and pattern should preserve.

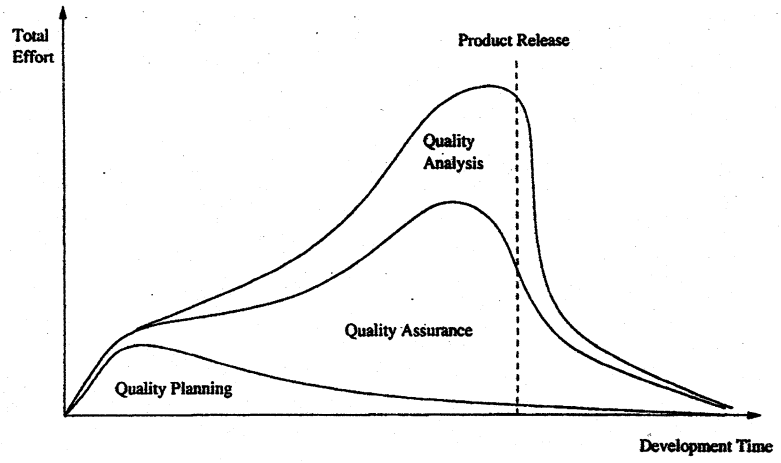

Figure 5.3 Quality engineering effort profile: The share of different activities as part of the total effort

In addition, the general shape and pattern of the profile such as in Figure 5.3 should preserve regardless of the specific development process used. Waterfall process would see more dominance of quality planning in the beginning, and dominance of testing near product release, and measurement and quality assessment activities peak right before product release.

Other development processes, such as incremental, iterative, spiral, and extreme programming processes, would be associated with curves that vary less between the peaks and valleys. QA is spread out more evenly in these processes than in the waterfall process, although it is still expected to peak a little bit before product release. Similarly, measurement and analysis activities are also spread out more evenly to monitor and assess each part or increment, with the cumulative modeling results used in product release decisions. There are also more adjustments and small-scale planning activities involved in quality planning, which also makes the corresponding profiles less variable as well.

5.5 CONCLUDING REMARKS

To manage the quality assurance (QA) activities and to provide realistic opportunities of quantifiable quality improvement, we need to go beyond QA to perform the following:

- *Quality planning* before specific QA activities are carried out, in the so-called pre-QA activities in software quality engineering. We need to set the overall quality goal by managing customer's quality expectations under the project cost and budgetary constraints. We also need to select specific QA alternatives and techniques to implement as well as measurement and models to provide project monitoring and qualitative feedback.

- *Quality quantification and improvement* through measurement, analysis, feedback, and follow-up activities. These activities need to be carried out after the start of specific QA activities, in the so-called post-QA activities in software quality engineering. The analyses would provide us with quantitative assessment of product quality, and

identification of improvement opportunities. The follow-up actions would implement these quality and process improvement initiatives and help us achieve quantifiable quality improvement.

The integration of these activities with the QA activities forms our software quality engineering process depicted in Figure 5.1, which can also be integrated into the overall software development and maintenance process. Following this general framework and with a detailed description of pre-QA quality planning in this chapter, we can start our examination of the specific QA techniques and post-QA activities in the rest of this book.

Problems

5.1 What is the difference between quality assurance and quality engineering?

5.2 Why is quantification of quality goals important?

5.3 What can you do if certain quality goals are hard to quantify? Can you give some concrete examples of such situations and practical suggestions?

5.4 There are some studies on the cost-of-quality in literature, but the results are generally hard to apply to specific projects. Do you have some suggestions on how to assess the cost-of-quality for your own project? Would it be convincing enough to be relied upon in negotiating quality goals with your customers?

5.5 As mentioned in this chapter, the quality engineering effort profile would be somewhat different from that in Figure 5.3 if processes other than waterfall are used. Can you assess, qualitatively or quantitatively, the differences when other development processes are used?

5.6 Based on some project data you can access, build your own quality engineering effort profile and compare it to that in Figure 5.3. Pay special attention to development process used and the division of planning, QA, and analysis/follow-up activities.