# CHAPTER 1

# OVERVIEW

Computers and software systems are becoming ubiquitous in modern society. Worldwide users rely on individual and interconnected computers, as well as the global information infrastructure, such as the Internet and the World Wide Web (WWW), to fulfill their needs for information processing, storage, search, and retrieval. All these needs are met with the support of the underlying software. This reliance requires the software to function correctly over a long time, to be easy to use, and so on. In general, such requirements for high *quality* need to be satisfied by the people involved in the development and support of these software systems through various quality assurance activities, and the claims for high quality need to be supported by evidence based on concrete measurements and analyses. This chapter introduces various concepts related to quality, quality assurance (QA), and quality engineering, and outlines the contents of this book.

## 1.1 MEETING PEOPLE'S QUALITY EXPECTATIONS

In general, people's quality expectations for software systems they use and rely upon are two-fold:

1. The software systems must do what they are supposed to do. In other words, they must *do the right things*.
2. They must perform these specific tasks correctly or satisfactorily. In other words, they must *do the things right*.

The former requires that the software be the "right software", or perform the right functions. For example, an airline reservation system is supposed to handle reservations, not intended to fly airplanes automatically. The focus of the related activities is to *validate* the required software functions under their intended operational environment. The latter requires that the software systems perform their intended functions without problems. In the airline reservation system example, the system should help travel agents or individual travelers make valid reservations within a pre-specified time limit, instead of making invalid ones, taking too long to make a reservation, or refusing to make reservations without proper justification. The focus of the related activities is to *verify* that the implemented software functions operate as specified.

### Main tasks for software quality engineering

As the main topics of this book, the tasks for software QA and quality engineering are to ensure software quality through the related validation and verification activities. These activities need to be carried out by the people and organizations responsible for developing and supporting these software systems in an overall quality engineering process that includes:

- quality planning;
- execution of selected QA or software validation and verification activities;
- measurement and analysis to provide convincing evidence to demonstrate software quality to all parties involved.

In particular, customers and users need to have the assurance that their quality expectations are satisfied by the delivered software systems. The overall experience and lessons learned in delivering such high-quality software systems can be packaged into the software quality engineering process for quantifiable quality improvement in future development projects or to provide better product support.

When viewed from a different angle, the negative impact of software problems is also increasing, accompanying the pervasive use of and reliance on software systems in modern society. The problems could be associated with performing wrong functions, or performing intended functions incorrectly, thus causing unintended consequences. We would like to see such negative impact be eliminated, if possible. However, due to the increasing demand for automation, additional functionality and convenience by modern society to the computer and software systems, and due to the ubiquitous nature of modern computer, software, and information infrastructure, the size and complexity of modern software systems have also increased steadily. This increase in size and complexity also has unintended consequences in terms of causing quality problems.

### Quality problems in large software systems

Many software systems nowadays are highly complex and contain millions of lines of source code. Examples of such large software systems can be found in virtually every product segment or every application domain, from various operating systems, such as commonly used versions of the Microsoft Windows and UNIX operations systems, to commercial software products, such as database products, to aviation and in-flight entertainment

software used on Boeing 777, to defense related software systems, such as various command/communication/control (CCC) systems.

Such large and complex systems typically involve hundreds or even thousands of people in their development over months or even years, and the systems are often to be operated under diverse, and sometimes unanticipated, application environments. One may argue that some systems are unnecessarily large and complex. According to (Wirth, 1995), such "fat software" may be caused by indiscriminately adding non-essential features, poor design, improper choices of languages and methodologies, which could be addressed by disciplined methodologies and return to essentials for "lean software". Various QA techniques, including many of those covered in this book, can help produce high-quality, lean software.

However, there is no "silver bullet", or an all powerful and effective solution to the size, complexity, quality, and other software engineering problems, due to the fundamental requirements and constraints that a software system must satisfy (Brooks, 1987). Accompanying the size and complexity problems are the many chances for other problems to be introduced into the software systems. Therefore, dealing with problems that may impact customers and users negatively and trying to manage and improve software quality are a fact of life for people involved in the development, management, marketing, and operational support of most modern software systems.

### Testing, quality assurance (QA), and quality engineering

The above factors make it virtually impossible or practically infeasible to achieve the complete prevention or elimination of software problems and related negative impact. Consequently, various software QA activities are carried out to prevent or eliminate certain classes of problems that lead to such negative impact, or to reduce the likelihood or severity of such negative impact when it is unavoidable. This book systematically describes topics and issues related to these software QA activities, with an emphasis on the technical aspects.

Software testing plays a central role among the software QA activities. By running the software system or executing its prescribed functions, testers can determine if the observed system behavior conforms to its specifications or requirements. If discrepancies exist between the two, follow-up actions can be carried out to locate and remove the related problems in software code, which may also include modifying the software design. Therefore, the detection and removal of defects through testing help reduce the number of defects in delivered software products, thus helping to achieve the quality goals. Even if no discrepancy is observed, the specific instances can be accumulated as evidence to demonstrate that the software performs as specified. Consequently, testing is the most frequently used means to assure and to demonstrate software quality. A substantial part of this book is devoted to software testing, with an emphasis on commonly used techniques that have proven to be effective in various practical application environments.

Beyond testing, there are many other QA alternatives supported by related techniques and activities, such as inspection, formal verification, defect prevention, and fault tolerance. Inspection is a critical examination of software code or other artifacts by human inspectors to identify and remove problems directly, without resorting to execution. Fault tolerance prevents global system failures even if local problems exist, through various redundancies strategically designed and implemented into the software systems. Other QA techniques employ specific means to assure software quality. This book also provides a comprehensive coverage of these topics.

In addition, all these QA activities need to be managed in an engineering process we call the software quality engineering process, with quality goals set early in the product

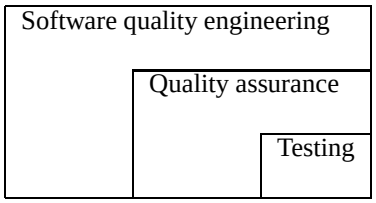


**Figure 1.1** Scope and content hierarchy: Testing, quality assurance (QA), and software quality engineering

development, and strategies for QA selected, carried out, and monitored to achieve these preset quality goals. As part of this overall process, data collected during the QA activities, as well as from the overall development activities, can be analyzed to provide feedback to the software development process for decision making, project management, and quantifiable quality improvement. This book also provides a comprehensive coverage of these topics.

## 1.2 BOOK ORGANIZATION AND CHAPTER OVERVIEW

Figure 1.1 illustrates the general scope of the topics introduced above: Testing is an important subset of QA activities; and QA is an important subset of quality engineering activities. This diagram also explains our book title: "Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement". This book is organized in four major parts and 22 chapters, with the main topics outlined below.

### Part I: Overview and Basics

Part I gives a general introduction and overview of the topics covered in the book, and presents the basic concepts and definitions related to quality, QA, testing, quality engineering, etc. Specific questions answered include:

- About this book: What is it? How to use it? How is it organized? In addition, what background knowledge is needed to have a thorough understanding of the technical aspects of this book? These questions are answered in Chapter 1.
- What is software quality? In particular, what are the different views of quality? Is quality a single, atomic concept, or does it consist of many different attributes or characteristics? What is the relationship between quality, correctness, and defect? Can we narrow down the definition of quality to better focus our attention on various QA activities commonly carried out during software life cycles? These questions are answered in Chapter 2.
- What is QA? The question is answered from a particular perspective in Chapter 3, representing a defect-based interpretation of quality and QA.
- What are the different QA activities and related techniques? A defect-based classification is presented, also in Chapter 3, for the major QA alternatives and techniques, such as testing, inspection, formal verification, fault tolerance, and so on.
- How to fit the different QA activities into the software development processes? What about other frameworks to classify QA activities? These questions are answered in Chapter 4.

- The QA activities are broadened in Chapter 5 into quality engineering that includes quality planning prior to specific QA activities and measurement, analysis, and feedback activities to close the loop for quality assessment and quantifiable improvement.

### Part II: Software Testing

Part II deals with all the important topics related to software testing, with an emphasis on commonly used testing techniques that have proven to be effective and efficient in many practical application environments. The chapters in this part are organized into two subparts: Descriptions of specific testing techniques (Chapters 8 through 11) are surrounded by chapters on the general issues of testing (Chapters 6, 7, and 12). Individual chapters are described below:

- General questions, issues, terminology about testing, including the generic testing process and a taxonomy for testing, are discussed in Chapter 6.

- The major testing activities, people's roles and responsibilities in these activities, test management, and test automation issues are covered in Chapter 7.

- Checklist and partition-based testing: Chapter 8 starts with the simplest testing of them all, *ad hoc* testing, then progresses to more organized testing using simple models such as *lists* and *partitions*. Specific testing techniques covered in Chapter 8 include:

  - testing with different types of general checklists;
  - decision and predicate testing;
  - usage-based statistical testing using flat operational profiles.

- Boundary testing: As a special case and extension of partition testing, we cover boundary testing in Chapter 9. Application of boundary testing ideas in other testing situations is also covered.

- State-based testing: Both the finite-state machines (FSMs), which serve as the basis for state-based testing, and the augmented FSMs, which form Markov chains for more in-depth usage-based statistical testing, are covered in Chapter 10.

- Interaction testing: Instead of focusing on individual partitions or states, the testing techniques described in Chapter 11 deal with the interactions along a complete execution path or a dependency slice. Specifically, this chapter covers the following traditional testing techniques:

  - control-flow testing (CFT);
  - data-flow testing (DFT).

- Chapter 12 discusses application of specific testing techniques for specific testing tasks in different sub-phases or in specialized tasks. The integration of different testing techniques to fulfill some common purposes is also discussed.

### Part III: Quality Assurance Beyond Testing

Part III covers important QA techniques other than testing, including the ones described below, and a comparison of all the QA alternatives at the end.

- Various defect prevention techniques are described in Chapter 13.
- Software inspection, or critical examination of software artifacts by human inspectors, is described in Chapter 14.
- Formal verification of program correctness with respect to its formal specifications is covered in Chapter 15.
- Fault tolerance techniques that prevent failures through some redundancy or duplication are discussed in Chapter 16. Related techniques based on similar ideas, such as failure containment to minimize failure impact, are also discussed in Chapter 16.
- Some program analysis techniques, specifically static analyses, are also covered in Chapter 14 in connection to inspection. Related topics on dynamic program analyses are briefly covered in Chapter 12 in connection to specialized testing techniques.
- Comparison of different QA alternatives and techniques, including those covered in Part III as well as testing covered in Part II, is presented in Chapter 17.

### Part IV: Quantifiable Quality Improvement

Part IV covers the important activities carried out in parallel or as follow-up to the main QA activities described in Part II and Part III. The purpose of these activities is to monitor the QA activities to provide quantitative quality assessment and feedback to the quality engineering process. Such assessment and feedback can be used to help with decision making, project management, and various improvement initiatives. The main contents of the specific chapters in this part are described below:

- First, the parallel and follow-up activities, as well as the collection and usage of the raw and processed data in related analyses to provide specific feedback for various purposes, are described in Chapter 18.
- Chapter 19 describes different models and measurements for quality assessment and improvement, and classifies them according to the information provided and the specific types of data required.
- Defect classification and analysis models are described in Chapter 20, as an important sub-class of quality assessment models that focuses on the collection and analysis of detailed defect information.
- Further analysis of the discovered defects and other measurement data from QA and overall development activities can be carried out to identify high-risk or high-defect areas for focused remedial actions aimed at effective quality improvement. Various risk identification techniques and related models for doing this are presented in Chapter 21.
- As an alternative to the defect-based view of quality that is closer to the developers' perspective, reliability is a quality measure that is closer to the users' perspective

and more meaningful to target customers. Chapter 22 presents software reliability models and analysis techniques to provide reliability assessments and guidance for reliability improvement.

## 1.3 DEPENDENCY AND SUGGESTED USAGE

The integration of the interconnected chapters is an important feature of this book. We next examine the topic and chapter dependencies, and discuss different ways that these topics can be combined for different readers with different purposes in mind.

### Chapter dependency

Figure 1.2 depicts the dependencies among different chapters, as well as among different parts, with each part grouped by dotted lines. We use solid lines to depict essential dependencies and dashed lines to depict dependencies that are desirable but not essential. An example of the latter type of dependencies is the non-essential dependency between quality assessment and analysis in Part IV and QA topics in Parts II and III: The knowledge of the topics presented in Parts II and III would make most of topics covered in Part IV more meaningful. However, one can have a general understanding of Part IV without a thorough knowledge of Parts II and III. Similarly, although all the chapters in Part III except the last one can be treated as parallel ones, Chapters 13 through 16 generally follow the sequence of activities or phases in the development process. Therefore, it would be more logical to follow this sequence. Some specific dependencies are explained below:

- In addition to Chapter 17's dependency on previous chapters of Part III, it should also be preceded by chapters in Part II, at least Chapter 6, because the comparison of QA alternatives in Chapter 17 rely on the general knowledge of individual alternatives and techniques.

- The chapters on testing techniques in Part II follow the natural progression from simple models to complex ones. However, there is no essential dependency between those based on simple partitions (Chapters 8 and 9) and those based on more complex models (Chapters 10 and 11).

- The last two chapters in Part IV can be treated as parallel chapters except that part of Chapter 22, the topic on tree-based reliability models (TBRMs), uses the modeling technique called tree-based modeling covered in Chapter 21.

### Suggested usage

This book is suitable as the main textbook for a one-semester course in various software engineering programs. Other people who are interested in learning all the major topics in software quality engineering should also read the whole book. However, for people who just want to get a general idea of the topics covered in this book, the following chapters are appropriate:

- *The minimal set:* Chapters 1–6, 17, and 18. This minimal set includes all five chapters in Part I and one chapter each from Parts II, III, and IV, respectively.

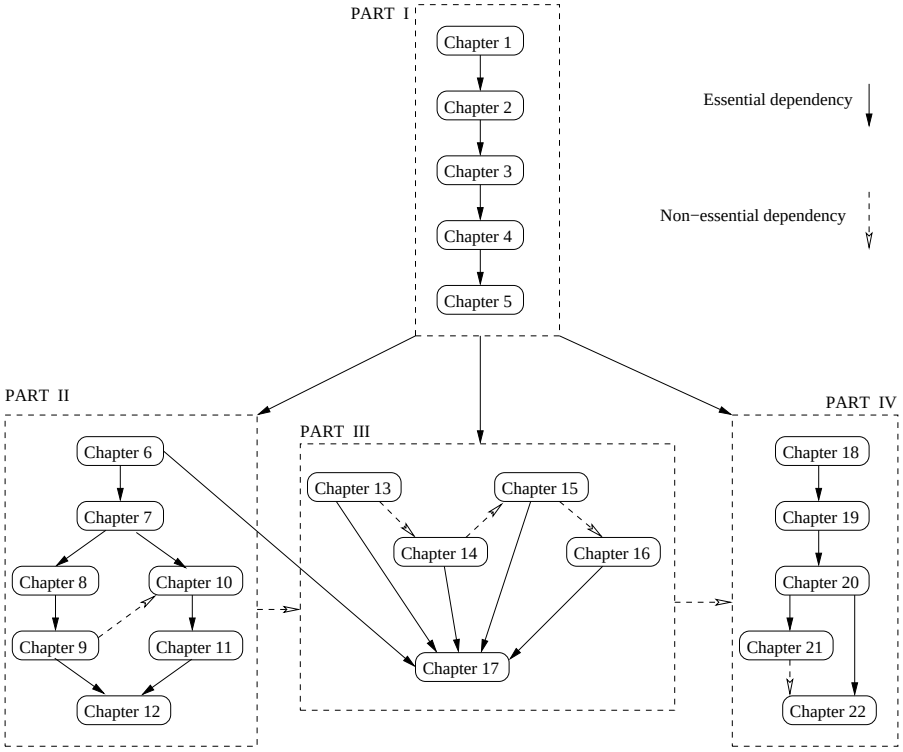


**Figure 1.2** Chapter and PART dependency diagram

Between these two extremes (the minimal set and all chapters), there are also other possible usages of this book. All the following would assume the basic coverage of minimal set of chapters above and some other chapters in addition to it. Some suggested usages are given below:

- Half semester course: Cover all in selective details, with emphasis on either Part II, III, or IV.

- Short course on specialized topics: minimal set above plus one of the part from Parts II, III, and IV. Such short courses would be similar in length to about ten hours or 3–4 weeks of class lectures.

- Other combinations of chapters are also possible, but would require the reader to keep track of the cross-references in topics and related dependencies using Figure 1.2 as the guide.

In addition to its use as a textbook, or as a technical book that introduces other people to the important topics of software quality engineering, the comprehensive coverage of all the important topics and pointers to further reading should also make this book a good reference for readers in their professional career.

## 1.4 READER PREPARATION AND BACKGROUND KNOWLEDGE

To have a good understanding of the technical details, the readers need to have a general knowledge of mathematics, statistics, computer science, and software engineering, equivalent to that at the level of college juniors, seniors, or new graduate students in computer science, software engineering, or a related field. The following is intended as a general checklist for the readers: If you find that you lack certain background knowledge listed below, you need to study or review them on your own before proceeding to related technical discussions. This checklist will help readers link specific pieces of background knowledge to specific parts of the book.

### Mathematical and statistical knowledge

Reviewing standard textbooks on mathematics and statistics covering the following topics would be useful if you are unfamiliar with some of them:

- Basic concepts of relations, algebra, and set theory: Used throughout the book, and especially in the following:
  - Sets, subsets, partitions, basic types of relations, and equivalence classes in Chapter 8 for partition-based testing.
  - Use of algebraic equations to define boundaries in Chapter 9 for boundary testing.
  - Precedence and dependency relations in Chapter 11 for control-flow and data-flow testing.
  - Cause–effect relations in Chapter 16 for hazard analysis and safety assurance, and in Chapter 20 for defect analysis.
- Logic, particularly Boolean logic, and related formalisms: Used throughout the book, and especially in the following:
  - Boolean logic for predicate and decision testing in Chapter 8.
  - Mathematical logic and formalisms in Chapter 15 for formal verification of program correctness.
- Some basic concepts of graph theory: Used throughout the book, and especially in the following:
  - Decision trees in Chapter 8 for operational profiles used in statistical testing.
  - Graph elements for finite-state machines (FSMs) and related testing in Chapter 10.
  - Flow-chart like situations for control-flow testing in Chapter 11.
  - Data dependency graphs (a tree-structured graph) for data-flow testing in Chapter 11.
  - Trees in fault-tree analysis and event-tree analysis in Chapter 16 for hazard analysis and safety assurance.
  - Tree-based models for risk identification in Chapter 21 and for reliability analysis in Chapter 22.

- Basic concepts of probability and statistics: Particularly important to the following topics:
  - Usage-based testing in Chapters 8 and 10.
  - Defect classification and distribution analysis in Chapter 20.
- Basic concepts of statistical analysis and modeling: Important to the topics in Part IV, in particular,
  - General analysis and modeling techniques in Chapter 19.
  - Various specific types of analyses for risk identification in Chapter 21.
  - Stochastic process and analysis for software reliability modeling in Chapter 22.

### Computer science knowledge

Reviewing standard textbooks on computer science covering the following topics would be useful if you are unfamiliar with some of them:

- Familiarity with programming and general software development using a high-level language. However, to make the understanding of basic concepts independent of specific implementation languages, example programs in the book are given in pseudocode form. Therefore, at a minimum, the readers need to be familiar with pseudocode commonly used to present basic algorithms in computer science literature and sometimes to illustrate design ideas during software development.
- Fundamentals of computing, particularly:
  - Finite-state machines (FSMs), which are the basis for state-based testing in Chapter 10.
  - Execution flow and data dependencies, which are the basis for control flow and data flow testing in Chapter 11.
  - Some formalisms about computing and programming languages used in Chapters 10, 11, and 15.
  - Some analysis techniques commonly identified with computer science and artificial intelligence, such as pattern matching, learning algorithms, and neural networks used in Chapter 21.
- Design and organization of computer and software systems such as used in parallel and redundant systems in Chapter 16.

### Software engineering knowledge

Reviewing standard textbooks on software engineering covering the following topics would be useful if you are unfamiliar with some of them:

- General knowledge of software development and maintenance activities, including requirement analysis, product specification, design, coding, testing, release, support, etc.

- General awareness of different software development processes, including waterfall, spiral, incremental, iterative, extreme programming (XP), etc., and the software process capability maturity model (CMM).
- General awareness with software management and system engineering issues, including economic consequences of project decisions, tradeoffs between different objectives and concerns, feedback and improvement mechanisms, optimization, etc.
- Familiarity with at least one of the commonly used development methodologies (and related tools), such as object-oriented development (OOD), structured development (SD), Cleanroom technology, agile methods, formal methods, etc.
- Practical experience working with some industrial software projects would be extremely helpful.

## Problems

**1.1** Consider some of your daily activities and classify them according the role played by computers and underlying software: no role, minor role, major role, and critical role. If "no role" is your answer for all the areas/activities, STOP — this is not a book for you. Otherwise, perform an overall assessment on how important software quality is to your daily activities.

**1.2** Use the dependency diagram in Figure 1.2 and related explanations in Section 1.3 to construct your individual study plan to fulfill your personal goals.

**1.3** Use the checklist in Section 1.4 and your personal goals to see if you need to review any background knowledge. If so, construct your individual study plan to get yourself ready for the rest of the book.