PART 1

Overview

1

Introduction

The ideas and techniques of software testing have become essential knowledge for all software developers. A software developer can expect to use the concepts presented in this book many times during his or her career. This chapter introduces the subject of software testing by describing the activities of a test engineer, defining a number of key terms, and then explaining the central notion of test coverage.

Software is a key ingredient in many of the devices and systems that pervade our society. Software defines the behavior of network routers, financial networks, telephone switching networks, the Web, and other infrastructure of modern life. Software is an essential component of embedded applications that control exotic applications such as airplanes, spaceships, and air traffic control systems, as well as mundane appliances such as watches, ovens, cars, DVD players, garage door openers, cell phones, and remote controllers. Modern households have over 50 processors, and some new cars have over 100; all of them running software that optimistic consumers assume will never fail! Although many factors affect the engineering of reliable software, including, of course, careful design and sound process management, testing is the primary method that the industry uses to evaluate software under development. Fortunately, a few basic software testing concepts can be used to design tests for a large variety of software applications. A goal of this book is to present these concepts in such a way that the student or practicing engineer can easily apply them to any software testing situation.

This textbook differs from other software testing books in several respects. The most important difference is in how it views testing techniques. In his landmark book *Software Testing Techniques*, Beizer wrote that testing is simple – all a tester needs to do is "find a graph and cover it." Thanks to Beizer's insight, it became evident to us that the myriad testing techniques present in the literature have much more in common than is obvious at first glance. Testing techniques typically are presented in the context of a particular software artifact (for example, a requirements document or code) or a particular phase of the lifecycle (for example, requirements analysis or implementation). Unfortunately, such a presentation obscures the underlying similarities between techniques. This book clarifies these similarities.

It turns out that graphs do not characterize all testing techniques well; other abstract models are necessary. Much to our surprise, we have found that a small number of abstract models suffice: graphs, logical expressions, input domain characterizations, and syntactic descriptions. The main contribution of this book is to simplify testing by classifying coverage criteria into these four categories, and this is why Part II of this book has exactly four chapters.

This book provides a balance of theory and practical application, thereby presenting testing as a collection of objective, quantitative activities that can be measured and repeated. The theory is based on the published literature, and presented without excessive formalism. Most importantly, the theoretical concepts are presented when needed to support the practical activities that test engineers follow. That is, this book is intended for software developers.

1.1 ACTIVITIES OF A TEST ENGINEER

In this book, a *test engineer* is an information technology (IT) professional who is in charge of one or more technical test activities, including designing test inputs, producing test case values, running test scripts, analyzing results, and reporting results to developers and managers. Although we cast the description in terms of test engineers, every engineer involved in software development should realize that he or she sometimes wears the hat of a test engineer. The reason is that each software artifact produced over the course of a product's development has, or should have, an associated set of test cases, and the person best positioned to define these test cases is often the designer of the artifact. A *test manager* is in charge of one or more test engineers. Test managers set test policies and processes, interact with other managers on the project, and otherwise help the engineers do their work.

Figure 1.1 shows some of the major activities of test engineers. A test engineer must design tests by creating test requirements. These requirements are then

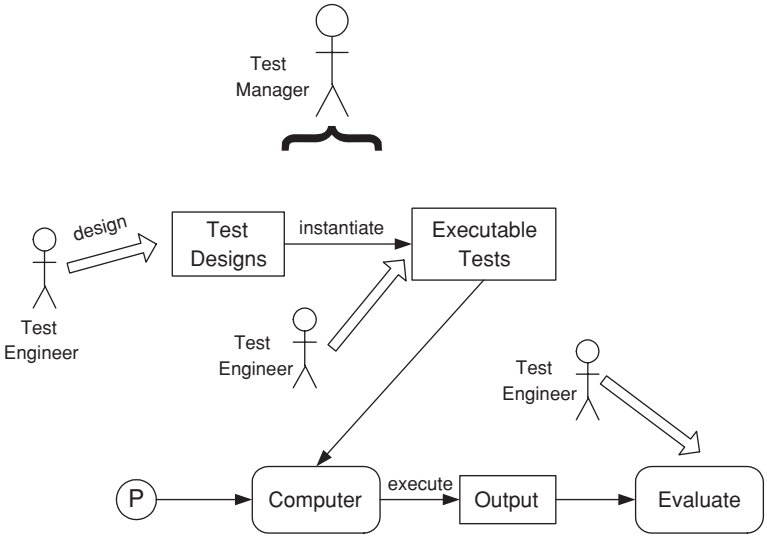

Figure 1.1. Activities of test engineers.

transformed into actual values and scripts that are ready for execution. These executable tests are run against the software, denoted P in the figure, and the results are evaluated to determine if the tests reveal a fault in the software. These activities may be carried out by one person or by several, and the process is monitored by a test manager.

One of a test engineer's most powerful tools is a formal coverage criterion. Formal coverage criteria give test engineers ways to decide what test inputs to use during testing, making it more likely that the tester will find problems in the program and providing greater assurance that the software is of high quality and reliability. Coverage criteria also provide stopping rules for the test engineers. The technical core of this book presents the coverage criteria that are available, describes how they are supported by tools (commercial and otherwise), explains how they can best be applied, and suggests how they can be integrated into the overall development process.

Software testing activities have long been categorized into levels, and two kinds of levels have traditionally been used. The most often used level categorization is based on traditional software process steps. Although most types of tests can only be run after some part of the software is implemented, tests can be designed and constructed during all software development steps. The most time-consuming parts of testing are actually the test design and construction, so test activities can and should be carried out throughout development. The second-level categorization is based on the attitude and thinking of the testers.

1.1.1 Testing Levels Based on Software Activity

Tests can be derived from requirements and specifications, design artifacts, or the source code. A different level of testing accompanies each distinct software development activity:

- Acceptance Testing – assess software with respect to requirements.
- System Testing – assess software with respect to architectural design.
- Integration Testing – assess software with respect to subsystem design.
- Module Testing – assess software with respect to detailed design.
- Unit Testing – assess software with respect to implementation.

Figure 1.2 illustrates a typical scenario for testing levels and how they relate to software development activities by isolating each step. Information for each test level is typically derived from the associated development activity. Indeed, standard advice is to design the tests concurrently with each development activity, even though the software will not be in an executable form until the implementation phase. The reason for this advice is that the mere process of explicitly articulating tests can identify defects in design decisions that otherwise appear reasonable. Early identification of defects is by far the best means of reducing their ultimate cost. Note that this diagram is **not** intended to imply a waterfall process. The synthesis and analysis activities generically apply to any development process.

The *requirements analysis* phase of software development captures the customer's needs. *Acceptance testing* is designed to determine whether the completed software in fact meets these needs. In other words, acceptance testing probes

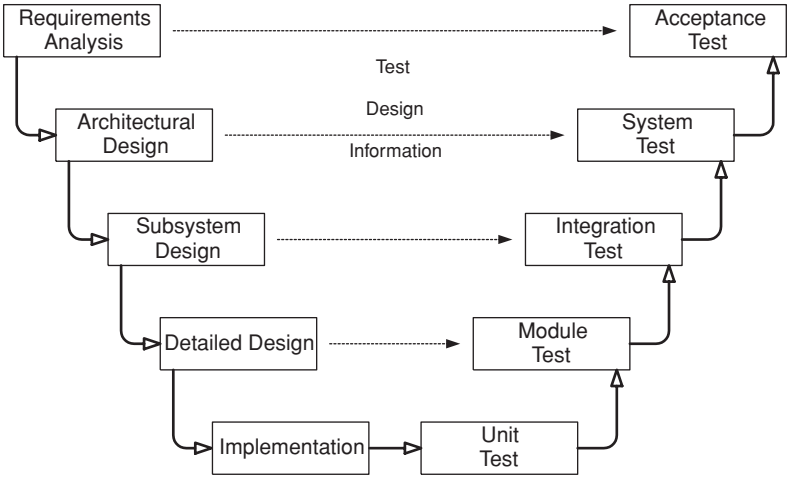

Figure 1.2. Software development activities and testing levels – the "V Model".

whether the software does what the users want. Acceptance testing must involve users or other individuals who have strong domain knowledge.

The *architectural design* phase of software development chooses components and connectors that together realize a system whose specification is intended to meet the previously identified requirements. *System testing* is designed to determine whether the assembled system meets its specifications. It assumes that the pieces work individually, and asks if the system works as a whole. This level of testing usually looks for design and specification problems. It is a very expensive place to find lower-level faults and is usually not done by the programmers, but by a separate testing team.

The *subsystem design* phase of software development specifies the structure and behavior of subsystems, each of which is intended to satisfy some function in the overall architecture. Often, the subsystems are adaptations of previously developed software. *Integration testing* is designed to assess whether the interfaces between modules (defined below) in a given subsystem have consistent assumptions and communicate correctly. Integration testing must assume that modules work correctly. Some testing literature uses the terms integration testing and system testing interchangeably; in this book, integration testing does **not** refer to testing the integrated system or subsystem. Integration testing is usually the responsibility of members of the development team.

The *detailed design* phase of software development determines the structure and behavior of individual modules. A program *unit*, or procedure, is one or more contiguous program statements, with a name that other parts of the software use to call it. Units are called functions in C and C++, procedures or functions in Ada, methods in Java, and subroutines in Fortran. A *module* is a collection of related units that are assembled in a file, package, or class. This corresponds to a file in C, a package in Ada, and a class in C++ and Java. *Module testing* is designed to assess individual modules in isolation, including how the component units interact with each other and their associated data structures. Most software development organizations make module testing the responsibility of the programmer.

Implementation is the phase of software development that actually produces code. *Unit testing* is designed to assess the units produced by the implementation phase and is the "lowest" level of testing. In some cases, such as when building general-purpose library modules, unit testing is done without knowledge of the encapsulating software application. As with module testing, most software development organizations make unit testing the responsibility of the programmer. It is straightforward to package unit tests together with the corresponding code through the use of tools such as *JUnit* for Java classes.

Not shown in Figure 1.2 is regression testing, a standard part of the maintenance phase of software development. *Regression testing* is testing that is done after changes are made to the software, and its purpose is to help ensure that the updated software still possesses the functionality it had before the updates.

Mistakes in requirements and high-level design wind up being implemented as faults in the program; thus testing can reveal them. Unfortunately, the software faults that come from requirements and design mistakes are visible only through testing months or years after the original mistake. The effects of the mistake tend to be dispersed throughout multiple software components; hence such faults are usually difficult to pin down and expensive to correct. On the positive side, even if tests cannot be executed, the very process of defining tests can identify a significant fraction of the mistakes in requirements and design. Hence, it is important for test planning to proceed concurrently with requirements analysis and design and not be put off until late in a project. Fortunately, through techniques such as use-case analysis, test planning is becoming better integrated with requirements analysis in standard software practice.

Although most of the literature emphasizes these levels in terms of **when** they are applied, a more important distinction is on the **types of faults** that we are looking for. The faults are based on the software **artifact** that we are testing, and the software **artifact** that we derive the tests from. For example, unit and module tests are derived to test units and modules, and we usually try to find faults that can be found when executing the units and modules individually.

One of the best examples of the differences between unit testing and system testing can be illustrated in the context of the infamous Pentium bug. In 1994, Intel introduced its Pentium microprocessor, and a few months later, Thomas Nicely, a mathematician at Lynchburg College in Virginia, found that the chip gave incorrect answers to certain floating-point division calculations.

The chip was slightly inaccurate for a few pairs of numbers; Intel claimed (probably correctly) that only one in nine billion division operations would exhibit reduced precision. The fault was the omission of five entries in a table of 1,066 values (part of the chip's circuitry) used by a division algorithm. The five entries should have contained the constant +2, but the entries were not initialized and contained zero instead. The MIT mathematician Edelman claimed that "the bug in the Pentium was an easy mistake to make, and a difficult one to catch," an analysis that misses one of the essential points. This was a very difficult mistake to find during system testing, and indeed, Intel claimed to have run millions of tests using this table. But the table entries were left empty because a loop termination condition was incorrect; that is, the loop stopped storing numbers before it was finished. This turns out to be a very simple fault to find during unit testing; indeed analysis showed that almost any unit level coverage criterion would have found this multimillion dollar mistake.

The Pentium bug not only illustrates the difference in testing levels, but it is also one of the best arguments for paying more attention to unit testing. There are no shortcuts – all aspects of software need to be tested.

On the other hand, some faults can only be found at the system level. One dramatic example was the launch failure of the first Ariane 5 rocket, which exploded 37 seconds after liftoff on June 4, 1996. The low-level cause was an unhandled floating-point conversion exception in an internal guidance system function. It turned out that the guidance system could never encounter the unhandled exception when used on the Ariane 4 rocket. In other words, the guidance system function is correct for Ariane 4. The developers of the Ariane 5 quite reasonably wanted to reuse the successful inertial guidance system from the Ariane 4, but no one reanalyzed the software in light of the substantially different flight trajectory of Ariane 5. Furthermore, the system tests that would have found the problem were technically difficult to execute, and so were not performed. The result was spectacular – and expensive!

Another public failure was the Mars lander of September 1999, which crashed due to a misunderstanding in the units of measure used by two modules created by separate software groups. One module computed thruster data in English units and forwarded the data to a module that expected data in metric units. This is a very typical integration fault (but in this case enormously expensive, both in terms of money and prestige).

One final note is that object-oriented (OO) software changes the testing levels. OO software blurs the distinction between units and modules, so the OO software testing literature has developed a slight variation of these levels. *Intramethod testing* is when tests are constructed for individual methods. *Intermethod testing* is when pairs of methods within the same class are tested in concert. *Intraclass testing* is when tests are constructed for a single entire class, usually as sequences of calls to methods within the class. Finally, *interclass testing* is when more than one class is tested at the same time. The first three are variations of unit and module testing, whereas interclass testing is a type of integration testing.

1.1.2 Beizer's Testing Levels Based on Test Process Maturity

Another categorization of levels is based on the test process maturity level of an organization. Each level is characterized by the goal of the test engineers. The following material is adapted from Beizer [29].

Level 0 There's no difference between testing and debugging.
Level 1 The purpose of testing is to show that the software works.
Level 2 The purpose of testing is to show that the software doesn't work.
Level 3 The purpose of testing is not to prove anything specific, but to reduce the risk of using the software.
Level 4 Testing is a mental discipline that helps all IT professionals develop higher quality software.

Level 0 is the view that testing is the same as debugging. This is the view that is naturally adopted by many undergraduate computer science majors. In most CS programming classes, the students get their programs to compile, then debug the programs with a few inputs chosen either arbitrarily or provided by the professor.

This model does not distinguish between a program's incorrect behavior and a mistake within the program, and does very little to help develop software that is reliable or safe.

In **Level 1** testing, the purpose is to show correctness. While a significant step up from the naive level 0, this has the unfortunate problem that in any but the most trivial of programs, correctness is virtually impossible to either achieve or demonstrate. Suppose we run a collection of tests and find no failures. What do we know? Should we assume that we have good software or just bad tests? Since the goal of correctness is impossible, test engineers usually have no strict goal, real stopping rule, or formal test technique. If a development manager asks how much testing remains to be done, the test manager has no way to answer the question. In fact, test managers are in a powerless position because they have no way to quantitatively express or evaluate their work.

In **Level 2** testing, the purpose is to show failures. Although looking for failures is certainly a valid goal, it is also a negative goal. Testers may enjoy finding the problem, but the developers never want to find problems – they want the software to work (level 1 thinking is natural for the developers). Thus, level 2 testing puts testers and developers into an adversarial relationship, which can be bad for team morale. Beyond that, when our primary goal is to look for failures, we are still left wondering what to do if no failures are found. Is our work done? Is our software very good, or is the testing weak? Having confidence in when testing is complete is an important goal for all testers.

The thinking that leads to **Level 3** testing starts with the realization that testing can show the presence, but not the absence, of failures. This lets us accept the fact that whenever we use software, we incur some risk. The risk may be small and the consequences unimportant, or the risk may be great and the consequences catastrophic, but risk is always there. This allows us to realize that the entire development team wants the same thing – to reduce the risk of using the software. In level 3 testing, both testers and developers work together to reduce risk.

Once the testers and developers are on the same "team," an organization can progress to real **Level 4** testing. Level 4 thinking defines testing as *a mental discipline that increases quality*. Various ways exist to increase quality, of which creating tests that cause the software to fail is only one. Adopting this mindset, test engineers can become the technical leaders of the project (as is common in many other engineering disciplines). They have the primary responsibility of measuring and improving software quality, and their expertise should help the developers. An analogy that Beizer used is that of a spell checker. We often think that the purpose of a spell checker is to find misspelled words, but in fact, the best purpose of a spell checker is to improve our ability to spell. Every time the spell checker finds an incorrectly spelled word, we have the opportunity to learn how to spell the word correctly. The spell checker is the "expert" on spelling quality. In the same way, level 4 testing means that the purpose of testing is to improve the ability of the developers to produce high quality software. The testers should train your developers.

As a reader of this book, you probably start at level 0, 1, or 2. Most software developers go through these levels at some stage in their careers. If you work in software development, you might pause to reflect on which testing level describes your company or team. The rest of this chapter should help you move to level 2 thinking, and to understand the importance of level 3. Subsequent chapters will give

you the knowledge, skills, and tools to be able to work at level 3. The ultimate goal of this book is to provide a philosophical basis that will allow readers to become "change agents" in their organizations for level 4 thinking, and test engineers to become **software quality experts**.

1.1.3 Automation of Test Activities

Software testing is expensive and labor intensive. Software testing requires up to 50% of software development costs, and even more for safety-critical applications. One of the goals of software testing is to automate as much as possible, thereby significantly reducing its cost, minimizing human error, and making regression testing easier.

Software engineers sometimes distinguish *revenue tasks*, which contribute directly to the solution of a problem, from *excise* tasks, which do not. For example, compiling a Java class is a classic excise task because, although necessary for the class to become executable, compilation contributes nothing to the particular behavior of that class. In contrast, determining which methods are appropriate to define a given data abstraction as a Java class is a revenue task. Excise tasks are candidates for automation; revenue tasks are not. Software testing probably has more excise tasks than any other aspect of software development. Maintaining test scripts, rerunning tests, and comparing expected results with actual results are all common excise tasks that routinely consume large chunks of test engineer's time. Automating excise tasks serves the test engineer in many ways. First, eliminating excise tasks eliminates drudgery, thereby making the test engineers job more satisfying. Second, automation frees up time to focus on the fun and challenging parts of testing, namely the revenue tasks. Third, automation can help eliminate errors of omission, such as failing to update all the relevant files with the new set of expected results. Fourth, automation eliminates some of the variance in test quality caused by differences in individual's abilities.

Many testing tasks that defied automation in the past are now candidates for such treatment due to advances in technology. For example, generating test cases that satisfy given test requirements is typically a hard problem that requires intervention from the test engineer. However, there are tools, both research and commercial, that automate this task to varying degrees.

EXERCISES

Section 1.1.

1. What are some of the factors that would help a development organization move from Beizer's testing level 2 (*testing is to show errors*) to testing level 4 (*a mental discipline that increases quality*)?
2. The following exercise is intended to encourage you to think of testing in a more rigorous way than you may be used to. The exercise also hints at the strong relationship between specification clarity, faults, and test cases.[1]
   (a) Write a Java method with the signature
   `public static Vector union (Vector a, Vector b)`
   The method should return a Vector of objects that are in either of the two argument Vectors.

(b) Upon reflection, you may discover a variety of defects and ambiguities in the given assignment. In other words, ample opportunities for faults exist. Identify as many possible faults as you can. (*Note:* `Vector` *is a Java* `Collection` *class. If you are using another language, interpret* `Vector` *as a list.*)
(c) Create a set of test cases that you think would have a reasonable chance of revealing the faults you identified above. Document a rationale for each test in your test set. If possible, characterize all of your rationales in some concise summary. Run your tests against your implementation.
(d) Rewrite the method signature to be precise enough to clarify the defects and ambiguities identified earlier. You might wish to illustrate your specification with examples drawn from your test cases.

1.2 SOFTWARE TESTING LIMITATIONS AND TERMINOLOGY

As said in the previous section, one of the most important limitations of software testing is that testing can show only the presence of failures, not their absence. This is a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a program is undecidable. Testers often call a successful (or effective) test one that finds an error. While this is an example of level 2 thinking, it is also a characterization that is often useful and that we will use later in this book.

The rest of this section presents a number of terms that are important in software testing and that will be used later in this book. Most of these are taken from standards documents, and although the phrasing is ours, we try to be consistent with the standards. Useful standards for reading in more detail are the IEEE Standard Glossary of Software Engineering Terminology, DOD-STD-2167A and MIL-STD-498 from the US Department of Defense, and the British Computer Society's Standard for Software Component Testing.

One of the most important distinctions to make is between validation and verification.

> *Definition 1.1 Validation:* The process of evaluating software at the end of software development to ensure compliance with intended usage.
>
> *Definition 1.2 Verification:* The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase.

Verification is usually a more technical activity that uses knowledge about the individual software artifacts, requirements, and specifications. Validation usually depends on domain knowledge; that is, knowledge of the application for which the software is written. For example, validation of software for an airplane requires knowledge from aerospace engineers and pilots.

The acronym "IV&V" stands for "independent verification and validation," where "independent" means that the evaluation is done by nondevelopers. Sometimes the IV&V team is within the same project, sometimes the same company, and sometimes it is entirely an external entity. In part because of the independent nature of IV&V, the process often is not started until the software is complete and is often done by people whose expertise is in the application domain rather than software