Introduction to Software Testing

Ammann & Offutt

Part I

Overview

This version of chapter 1 is a sample for consideration by instructors. You may use this to evaluate the book, but please do not distribute this to students or company employees in any way whatsoever, including making copies, email, or posting it on other websites.

Chapter 1

Introduction

The true subject matter of the programmer is not programs, but the design of computations.
— Dijkstra, SigPlan, July 1985
The true subject matter of the tester is not test cases, but the analysis of computations.

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 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

> *The purpose of a fish trap is to catch fish, and when the fish are caught, the trap is forgotten. The purpose of a rabbit snare is to catch rabbits. When the rabbits are caught, the snare is forgotten. The purpose of words is to convey ideas. When the ideas are grasped, the words are forgotten. Where can I find a man who has forgotten words? He is the one I would like to talk to.*
>
> – Chuang Tzu

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 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

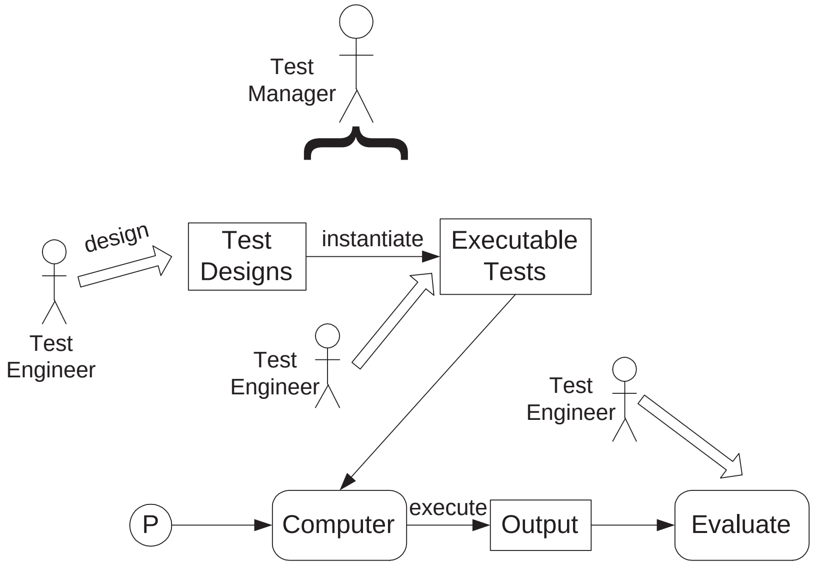

Figure 1.1: Activities of test engineers.

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.

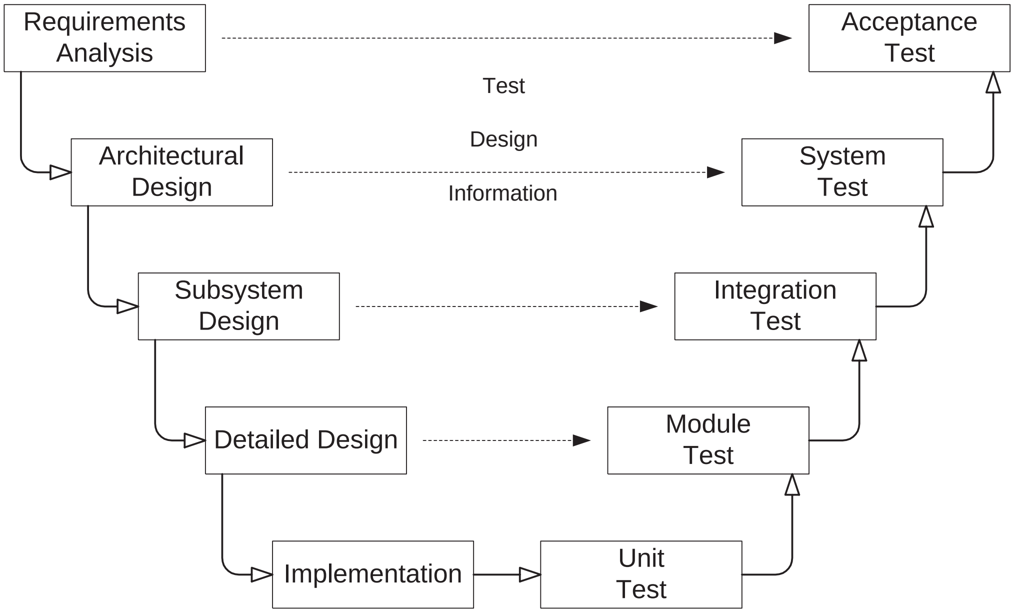

Figure 1.2: Software development activities and testing levels–the "V Model".

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 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 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 multi-million 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 the 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. *Intra-method testing* is when tests are constructed for individual methods. *Inter-method testing* is when pairs of methods within the same class are tested in concert. *Intra-class testing* is when tests are constructed for a single entire class, usually as sequences of calls to methods within the class. Finally, *inter-class testing* is when more than one class is tested at the same time. The first three are variations of unit and module testing, whereas inter-class 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 [3].

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 percent 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] Liskov's *Program Development in Java*, especially chapters 9 and 10, is a great source for students who wish to pursue this direction further.

1.2 Software Testing Limitations and Terminology

Perfection is the enemy of the good.

— Gustave Flaubert.

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 non-developers. 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 development. This can sometimes mean that validation is given more weight than verification.

Two terms that we have already used are fault and failure. Understanding this distinction is the first step in moving from level 0 thinking to level 1 thinking. We adopt the definition of software fault, error, and failure from the dependability community.

Definition 1.3 Software Fault: *A static defect in the software.*

Definition 1.4 Software Error: *An incorrect internal state that is the manifestation of some fault.*

Definition 1.5 Software Failure: *External, incorrect behavior with respect to the requirements or other description of the expected behavior.*

Consider a medical doctor making a diagnosis for a patient. The patient enters the doctor's office with a list of *failures* (that is, *symptoms*). The doctor then must discover the *fault*, or root cause of the symptom. To aid in the diagnosis, a doctor may order tests that look for anomalous internal conditions, such as high blood pressure, an irregular heartbeat, high levels of blood glucose, or high cholesterol. In our terminology, these anomalous internal conditions correspond to *errors*.

While this analogy may help the student clarify his or her thinking about faults, errors, and failures, software testing and a doctor's diagnosis differ in one crucial way. Specifically, faults in software are *design mistakes*. They do not appear spontaneously, but rather exist as a result of some (unfortunate) decision by a human. Medical problems (as well as faults in computer system hardware), on the other hand, are often a result of physical degradation. This distinction is important because it explains the limits on the extent to which any process can hope to control software faults. Specifically, since no foolproof way exists to catch arbitrary mistakes made by humans, we cannot eliminate all faults from software. In colloquial terms, we can make software development foolproof, but we cannot, and should not attempt to, make it damn-foolproof.

For a more technical example of the definitions of fault, error, and failure, consider the following Java method:

```
public static int numZero (int[] x) {
   // Effects: if x == null throw NullPointerException
   //    else return the number of occurrences of 0 in x
   int count = 0;
   for (int i = 1; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         count++;
      }
   }
   return count;
}
```

The fault in this program is that it starts looking for zeroes at index 1 instead of index 0, as is necessary for arrays in Java. For example, `numZero([2, 7, 0])` correctly evaluates to 1, while `numZero([0, 7, 2])` incorrectly evaluates to 0. In both of these cases the fault is executed. Although both of these cases result in an error, only the second case results in failure. To understand the error states, we need to identify the state for the program. The state for `numZero` consists of values for the variables `x`, `count`, `i`, and the program counter (denoted PC). For the first example given above, the state at the if statement on the very first iteration of the loop is (x = [2, 7, 0], count = 0, i = 1, PC = if). Notice that this state is in error precisely because the value of i should be zero on the first iteration. However, since the value of count is coincidentally correct, the error state does not propagate to the output, and hence the software does not fail. In other words, a state is in error simply if it is not the expected state, even if all of the values in the state, considered in isolation, are acceptable. More generally, if the required sequence of states is $s_0, s_1, s_2, ...$, and the actual sequence of states is $s_0, s_2, s_3, ...$, then state s_2 is in error in the second sequence.

In the second case the corresponding (error) state is (x = [0, 7, 2], count = 0, i = 1, PC = if). In this case, the error propagates to the variable count and is present in the return value of the method. Hence a failure results.

The definitions of fault and failure allow us to distinguish testing from debugging.

Definition 1.6 Testing: *Evaluating software by observing its execution.*

Definition 1.7 Test Failure: *Execution that results in a failure.*

Definition 1.8 Debugging: *The process of finding a fault given a failure.*

Of course the central issue is that for a given fault, not all inputs will "trigger" the fault into creating incorrect output (a failure). Also, it is often very difficult to relate a failure to the associated fault. Analyzing these ideas leads to the fault/failure model, which states that three conditions must be present for a failure to be observed.

1. First, the location or locations in the program that contain the fault must be reached (*Reachability*).

2. Second, after executing the location, the state of the program must be incorrect (*Infection*).

3. Third, the infected state must propagate to cause some output of the program to be incorrect (*Propagation*).

This "RIP" model is very important for coverage criteria such as mutation (Chapter 5) and for automatic test data generation. It is important to note that the RIP model applies even in the case of faults of omission. In particular, when execution traverses the missing code, the program counter, which is part of the internal state, necessarily has the wrong value.

The next definitions are less standardized and the literature varies widely. The definitions are our own but are consistent with common usage. A test engineer must recognize that tests include more than just input values, but are actually multi-part software artifacts. The piece of a test case that is referred to the most often is what we call the test case value.

Definition 1.9 Test Case Values: *The input values necessary to complete some execution of the software under test.*

Note that the definition of test case values is quite broad. In a traditional batch environment, the definition is extremely clear. In a web application, a complete execution might be as small as the generation of part of a simple web page, or it might be as complex as the completion of a set of commercial transactions. In a real-time system such as an avionics application, a complete execution might be a single frame, or it might be an entire flight.

Test case values are the inputs to the program that test engineers typically focus on during testing. They really define what sort of testing we will achieve. However, test case values are not enough. In addition to test case values, other inputs are often needed to run a test. These inputs may depend on the source of the tests, and may be commands, user inputs, or a software method to call with values for its parameters. In order to evaluate the results of a test, we must know what output a correct version of the program would produce for that test.

Definition 1.10 Expected Results: *The result that will be produced when executing the test if and only if the program satisfies its intended behavior.*

Two common practical problems associated with software testing are how to provide the right values to the software and observing details of the software's behavior. These two ideas are used to refine the definition of a test case.

Definition 1.11 Software Observability: *How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components.*

Definition 1.12 Software Controllability: *How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors.*

These ideas are easily illustrated in the context of embedded software. Embedded software often does not produce output for human consumption, but affects the behavior of some piece of hardware. Thus, observability will be quite low. Likewise, software for which all inputs are values entered from a keyboard is easy to control. But an embedded program that gets its inputs from hardware sensors is more difficult to control and some inputs may be difficult, dangerous or impossible to supply (for example, how does the automatic pilot behave when a train jumps off-track). Many observability and controllability problems can be addressed with simulation, by extra software built to "bypass" the hardware or software components that interfere with testing. Other applications that sometimes have low observability and controllability include component-based software, distributed software and web applications.

Depending on the software, the level of testing, and the source of the tests, the tester may need to supply other inputs to the software to affect controllability or observability. For example, if we are testing software for a mobile telephone, the test case values may be long distance phone numbers. We may also need to turn the phone on to put it in the appropriate state and then we may need to press "talk" and "end" buttons to view the results of the test case values and terminate the test. These ideas are formalized as follows.

Definition 1.13 Prefix Values: *Any inputs necessary to put the software into the appropriate state to receive the test case values.*

Definition 1.14 Postfix Values: *Any inputs that need to be sent to the software after the test case values are sent.*

Postfix values can be subdivided into two types.

Definition 1.15 Verification Values: *Values necessary to see the results of the test case values.*

Definition 1.16 Exit Commands: *Values needed to terminate the program or otherwise return it to a stable state.*

A test case is the combination of all these components (test case values, expected results, prefix values, and postfix values). When it is clear from context, however, we will follow tradition and use the term "test case" in place of "test case values."

Definition 1.17 Test Case: *A test case is composed of the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test.*

We provide an explicit definition for a test set to emphasize that coverage is a property of a set of test cases, rather than a property of a single test case.

Definition 1.18 Test Set: *A test set is simply a set of test cases.*

Finally, wise test engineers automate as many test activities as possible. A crucial way to automate testing is to prepare the test inputs as executable tests for the software. This may be done as Unix shell scripts, input files, or through the use of a tool that can control the software or software component being tested. Ideally, the execution should be complete in the sense of running the software with the test case values, getting the results, comparing the results with the expected results, and preparing a clear report for the test engineer.

Definition 1.19 Executable Test Script: *A test case that is prepared in a form to be executed automatically on the test software and produce a report.*

The only time a test engineer would not want to automate is if the cost of automation outweighs the benefits. For example, this may happen if we are sure the test will only be used once or if the automation requires knowledge or skills that the test engineer does not have.

Exercises, Section 1.2.

1. For what do testers use automation? What are the limitations of automation?

2. How are faults and failures related to testing and debugging?

3. Below are four faulty programs. Each includes a test case that results in failure. Answer the following questions about each program.

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {
      if (x[i] == y)
      {
         return i;
      }
   }
   return -1;
}
   // test:  x=[2, 3, 5]; y = 2
   //       Expected = 0
```

```
public static int lastZero (int[] x) {
//Effects: if x==null throw NullPointerException
//   else return the index of the LAST 0 in x.
//   Return -1 if 0 does not occur in x

   for (int i = 0; i < x.length; i++)
   {
      if (x[i] == 0)
      {
         return i;
      }
   }
   return -1;
}
   // test:  x=[0, 1, 0]
   //       Expected = 2
```

```
public int countPositive (int[] x) {
//Effects: If x==null throw NullPointerException
//   else return the number of
//        positive elements in x.
   int count = 0;
   for (int i=0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}
   // test:  x=[-4, 2, 0, 2]
   //       Expected = 2
```

```
public static int oddOrPos(int[] x) {
//Effects: if x==null throw NullPointerException
// else return the number of elements in x that
//      are either odd or positive (or both)
   int count = 0;
   for (int i = 1; i < x.length; i++)
   {
      if (x[i]%2 == 0 || x[i] > 0)
      {
         count++;
      }
   }
   return count;
}
   // test:  x=[-3, -2, 0, 1, 4]
   //       Expected = 3
```

(a) Identify the fault.

(b) If possible, identify a test case that does **not** execute the fault.

(c) If possible, identify a test case that executes the fault, but does **not** result in an error state.

(d) If possible identify a test case that results in an error, but **not** a failure. Hint: Don't forget about the program counter.

(e) For the given test case, identify the first error state. Be sure to describe the complete state.

(f) Fix the fault and verify that the given test now produces the expected output.

1.3 Coverage Criteria for Testing

criterion–plural *criteria. A standard on which a judgment or decision may be based.*

— Definition from Webster

Some ill-defined terms occasionally used in testing are "complete testing," "exhaustive testing," and "full coverage." These terms are poorly defined because of a fundamental theoretical limitation of software. Specifically, the number of potential inputs for most programs is so large as to be effectively infinite. Consider a Java compiler–the number of potential inputs to the compiler is not just all Java programs, or even all almost correct Java programs, but all strings. The only limitation is the size of the file that can be read by the parser. Therefore, the number of inputs is effectively infinite and cannot be explicitly enumerated.

This is where formal coverage criteria come in. Since we cannot test with all inputs, coverage criteria are used to decide which test inputs to use. The software testing community believes that effective use of coverage criteria makes it more likely that test engineers will find faults in a program and provides informal assurance that the software is of high quality and reliability. While this is, perhaps, more an article of faith than a scientifically supported proposition, it is, in our view, the best option currently available. From a practical perspective, coverage criteria provide useful rules for when to stop testing.

This book defines coverage criteria in terms of test requirements. The basic idea is that we want our set of test cases to have various properties, each of which is provided (or not) by an individual test case.[2]

Definition 1.20 Test Requirement: *A test requirement is a specific element of a software artifact that a test case must satisfy or cover.*

Test requirements usually come in sets, and we use the abbreviation *TR* to denote a set of test requirements.

Test requirements can be described with respect to a variety of software artifacts, including the source code, design components, specification modeling elements, or even descriptions of the input space. Later in this book, test requirements will be generated from all of these.

Let's begin with a non-software example. Suppose we are given the enviable task of testing bags of jelly beans. We need to come up with ways to sample from the bags. Suppose these jelly beans have the following six flavors and come in four colors: Lemon (colored Yellow), Pistachio (Green), Cantaloupe (Orange), Pear (White), Tangerine (also Orange), and Apricot (also Yellow). A simple approach to testing might be to test one jelly bean of each flavor. Then we have six test requirements, one for each flavor. We satisfy the test requirement "Lemon" by selecting and, of course, tasting a Lemon jelly bean from a bag of jelly beans. The reader might wish to ponder how to decide, prior to the tasting step, if a given Yellow jelly bean is Lemon or Apricot. This dilemma illustrates a classic controllability issue.

As a more software-oriented example, if the goal is to cover all decisions in the program (branch coverage), then each decision leads to two test requirements, one for the decision to evaluate to false, and one for the decision to evaluate to true. If every method must be called at least once (call coverage), each method leads to one test requirement.

[2] While this is a good general rule, exceptions exist. For example, test requirements for some logic coverage criteria demand pairs of related test cases instead of individual test cases.

A coverage criterion is simply a recipe for generating test requirements in a systematic way:

Definition 1.21 Coverage Criterion: *A coverage criterion is a rule or collection of rules that impose test requirements on a test set.*

That is, the criterion describes the test requirements in a complete and unambiguous manner. The "flavor criterion" yields a simple strategy for selecting jelly beans. In this case, the set of test requirements, TR, can be formally written out as:

$$\begin{aligned} TR \; = \; \{ & flavor = Lemon,\; flavor = Pistachio,\; flavor = Cantaloupe, \\ & flavor = Pear,\; flavor = Tangerine,\; flavor = Apricot\} \end{aligned}$$

Test engineers need to know how good a collection of tests is, so we measure test sets against a criterion in terms of coverage.

Definition 1.22 Coverage: *Given a set of test requirements TR for a coverage criterion C, a test set T satisfies C if and only if for every test requirement tr in TR, at least one test t in T exists such that t satisfies tr.*

To continue the example, a test set T with 12 beans: three Lemon, one Pistachio, two Cantaloupe, one Pear, one Tangerine, and four Apricot satisfies the "flavor criterion." Notice that it is perfectly acceptable to satisfy a given test requirement with more than one test.

Coverage is important for two reasons. First, it is sometimes expensive to satisfy a coverage criterion, so we want to compromise by trying to achieve a certain coverage level.

Definition 1.23 Coverage Level: *Given a set of test requirements TR and a test set T, the coverage level is simply the ratio of the number of test requirements satisfied by T to the size of TR.*

Second, and more importantly, some requirements cannot be satisfied. Suppose Tangerine jelly beans are rare, some bags may not contain any, or it may simply be too difficult to find a Tangerine bean. In this case, the flavor criterion cannot be 100% satisfied, and the maximum coverage level possible is 5/6 or 83%. It often makes sense to drop unsatisfiable test requirements from the set TR–or to replace them with less stringent test requirements.

Test requirements that cannot be satisfied are called *infeasible*. Formally, no test case values exist that meet the test requirements. Examples for specific software criteria will be shown throughout the book, but some may already be familiar. Dead code results in infeasible test requirements because the statements cannot be reached. The detection of infeasible test requirements is formally undecidable for most coverage criteria, and even though some researchers have tried to find partial solutions, they have had only limited success. Thus, 100% coverage is impossible in practice.

Coverage criteria are traditionally used in one of two ways. One method is to directly generate test case values to satisfy a given criterion. This method is often assumed by the research community and is the most obvious way to use criteria. It is also very hard in some cases, particularly if we do not have enough automated tools to support test case value generation. The other method is to generate test case values externally (by hand or using a pseudo-random tool, for example) and then measure the tests against the criterion in terms of their coverage. This method is usually favored by industry practitioners, because generating tests to directly satisfy the criterion is too hard. Unfortunately, this use is sometimes misleading. If our tests do not reach 100% coverage, what does that mean? We really have no data on how much, say, 99% coverage is worse than 100% coverage, or 90%, or even 75%. Because of this use of the criteria to evaluate existing test sets, coverage criteria are sometimes called *metrics.*

This distinction actually has a strong theoretical basis. A *generator* is a procedure that automatically generates values to satisfy a criterion, and a *recognizer* is a procedure that decides whether a given set of test case values satisfies a criterion. Theoretically, both problems are provably undecidable in the general case for most criteria. In practice, however, it is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion. The primary problem with recognition is infeasible test requirements; if no infeasible test requirements are present then the problem becomes decidable.

In practical terms of commercial automated test tools, a generator corresponds to a tool that automatically creates test case values. A recognizer is a coverage analysis tool. Coverage analysis tools are quite plentiful, both as commercial products and freeware.

It is important to appreciate that the set TR depends on the specific artifact under test. In the jelly bean example, the test requirement $color = Purple$ doesn't make sense because we assumed that the factory does not make Purple jelly beans. In the software context, consider statement coverage. The test requirement "Execute statement 42" makes sense only if the program under test does indeed have a statement 42. A good way to think of this issue is that the test engineer starts with a given software artifact and then chooses a particular coverage criterion. Combining the artifact with the criterion yields the specific set TR that is relevant to the test engineer's task.

Coverage criteria are often related to one another, and compared in terms of subsumption. Recall that the "flavor criterion" requires that every flavor be tried once. We could also define a "color criterion," which requires that we try one jelly bean of each color $\{yellow, green, orange, white\}$. If we satisfy the flavor criterion, then we have also implicitly satisfied the color criterion. This is the essence of subsumption; that satisfying one criterion will guarantee that another one is satisfied.

Definition 1.24 Criteria Subsumption: *A coverage criterion C_1 subsumes C_2 if and only if every test set that satisfies criterion C_1 also satisfies C_2.*

Note that this has to be true for **every** test set, not just some sets. Subsumption has a strong similarity with set subset relationships, but it is not exactly the same. Generally, a criterion C_1 can subsume another C_2 in one of two ways. The simpler way is if the test requirements for C_1 always form a superset of the requirements for C_2. For example, another jelly bean criterion may be to try all flavors whose name begins with the letter 'C'. This would result in the test requirements $\{Cantaloupe\}$, which is a subset of the requirements for the flavor criterion: $\{Lemon, Pistachio, Cantaloupe, Pear, Tangerine, Apricot\}$. Thus, the flavor criterion subsumes the "starts-with-C" criterion.

The relationship between the flavor and the color criteria illustrate the other way that subsumption can be shown. Since every flavor has a specific color, and every color is represented by at least one flavor, if we satisfy the flavor criterion we will also satisfy the color criterion. Formally, a many-to-one mapping exists between the requirements for the flavor criterion and the requirements for the color criterion. Thus, the flavor criterion subsumes the color criterion. (If a one-to-one mapping exists between requirements from two criteria, then they would subsume each other.)

For a more realistic software-oriented example, consider branch and statement coverage. (These should already be familiar, at least intuitively, and will be defined formally in Chapter 2.) If a test set has covered every branch in a program (satisfied branch coverage), then the test set is guaranteed to have covered every statement as well. Thus, the branch coverage criterion subsumes the statement coverage criterion. We will return to subsumption with more rigor and more examples in subsequent chapters.

1.3.1 Infeasibility and Subsumption

A subtle relationship exists between infeasibility and subsumption. Specifically, sometimes a criterion C_1 will subsume another criterion C_2 if and only if all test requirements are feasible. If some test requirements in C_1 are infeasible, however, C_1 may not subsume C_2.

Infeasible test requirements are common and occur quite naturally. Suppose we partition the jelly beans into *Fruits* and *Nuts*.[3] Now, consider the *Interaction Criterion*, where each flavor of bean is sampled in conjunction with some other flavor in the same block. Such a criterion has a useful counterpart in the software domain in cases where feature interactions are a source of concern. So, for example, we might try Lemon with Pear or Tangerine, but we would not try Lemon with itself or with Pistachio. We might think that the Interaction Criterion subsumes the Flavor criterion, since every flavor is tried in conjunction with some other flavor. Unfortunately, in our example, Pistachio is the only member of the *Nuts* block, and hence the test requirement to try it with some other flavor in the *Nuts* block is infeasible.

One possible strategy to reestablish subsumption is to replace each infeasible test requirement for the Interaction Criterion with the corresponding one from the Flavor criterion. In this example, we would simply taste Pistachio nuts by themselves. In general, it is desirable to define coverage criteria so that they are robust with respect to subsumption in the face of infeasible test requirements. This is not commonly done in the testing literature, but we make an effort to do so in this book.

[3] The reader might wonder whether we need an *Other* category to ensure that we have a partition. In our example, we are ok, but in general, one would need such a category to handle jelly beans such as Potato, Spinach, or Ear Wax.

That said, this problem is mainly theoretical and should not overly concern practical testers. Theoretically, sometimes a coverage criterion C_1 will subsume another C_2 if we assume that C_1 has **no** infeasible test requirements, but if C_1 does create an infeasible test requirement for a program, a test suite that satisfies C_1 while skipping the infeasible test requirements might also "skip" some test requirements from C_2 that are satisfiable. In practice, only a few test requirements for C_1 are infeasible for any given program, and if some are, it is often true that corresponding test requirements in C_2 will also be infeasible. If not, the few test cases that are lost will **probably** not make a difference in the test results.

1.3.2 Characteristics of a Good Coverage Criterion

Given the above discussion, an interesting question is "what makes a coverage criterion good?" Certainly, no definitive answers exist to this question, a fact that may partly explain why so many coverage criteria have been designed. However, three important issues can affect the use of coverage criteria.

1. The difficulty of computing test requirements
2. The difficulty of generating tests
3. How well the tests reveal faults

Subsumption is at best a very rough way to compare criteria. Our intuition may tell us that if one criterion subsumes another, then it should reveal more faults. However, no theoretical guarantee exists and the experimental studies have usually not been convincing and are far from complete. Nevertheless, the research community has reasonably wide agreement on relationships among some criteria. The difficulty of computing test requirements will depend on the artifact being used as well as the criterion. The fact that the difficulty of generating tests can be directly related to how well the tests reveal faults should not be surprising. A software tester must strive for balance and choose criteria that have the right cost / benefit tradeoffs for the software under test.

Exercises, Section 1.3.

1. Suppose that coverage criterion C_1 subsumes coverage criterion C_2. Further suppose that test set T_1 satisfies C_1 and on program P test set T_2 satisfies C_2, also on P.

   (a) Does T_1 necessarily satisfy C_2? Explain.
   (b) Does T_2 necessarily satisfy C_1? Explain.

(c) If P contains a fault, and T_2 reveals the fault, T_1 does **not** necessarily also reveal the fault. Explain.[4]

2. How else could we compare test criteria besides subsumption?

1.4 Older Software Testing Terminology

The only man who behaves sensibly is my tailor; he takes my measure anew every time he sees me, whilst all the rest go on with their old measurements, expecting them to fit me.
— George Bernard Shaw

The testing research community has been very active in the past two decades, and some of our fundamental views of what and how to test have changed. This section presents some of the terminology that has been in use for many years, but for various reasons has become dated. Despite the fact that they are not as relevant now as they were at one time, these terms are still used and it is important that testing students and professionals be familiar with them.

From an abstract perspective, black-box and white-box testing are very similar. In this book in particular, we present testing as proceeding from abstract models of the software such as graphs, which can as easily be derived from a black-box view or a white-box view. Thus, one of the most obvious effects of the unique philosophical structure of this book is that these two terms become obsolete.

Definition 1.25 Black-box testing: *Deriving tests from external descriptions of the software, including specifications, requirements, and design.*

Definition 1.26 White-box testing: *Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements.*

In the early 1980s, a discussion took place over whether testing should proceed from the top down or from the bottom up. This was an echo of a previous discussion over how to develop software. This distinction has pretty much disappeared as we first learned that top-down testing is impractical, then OO design pretty much made the distinction obsolete. The following pair of definitions assumes that software can be viewed as a tree of software procedures, where the edges represent calls and the root of the tree is the main procedure.

Definition 1.27 Top-Down Testing: *Test the main procedure, then go down through procedures it calls, and so on.*

[4] Correctly answering this question goes a long way towards understanding the weakness of the subsumption relation.

Definition 1.28 Bottom-Up Testing: *Test the leaves in the tree (procedures that make no calls), and move up to the root. Each procedure is tested only if all of its children have been tested.*

OO software leads to a more general problem. The relationships among classes can be formulated as general graphs with cycles, requiring test engineers to make the difficult choice of what order to test the classes in. This problem is discussed in Chapter 6.

Some parts of the literature separate static and dynamic testing as follows:

Definition 1.29 Static Testing: *Testing without executing the program. This includes software inspections and some forms of analysis.*

Definition 1.30 Dynamic Testing: *Testing by executing the program with real inputs.*

Most of the literature currently uses "testing" to refer to dynamic testing and "static testing" is called "verification activities." We follow that use in this book and it should be pointed out that this book is only concerned with dynamic or *execution-based* testing.

One last term bears mentioning because of the lack of definition. *Test Strategy* has been used to mean a variety of things, including coverage criterion, test process, and technologies used. We will avoid using it.

1.5 Bibliographic Notes

All books on software testing and all researchers owe major thanks to the landmark books in 1979 by Myers [34], in 1990 by Beizer [3], and in 2000 by Binder [4]. Some excellent overviews of unit testing criteria have also been published, including one by White [52] and more recently by Zhu, Hall and May [54]. The statement that software testing requires up to 50 percent of software development costs is from Myers and Sommerville [34, 44]. The recent text from Pezze and Young [41] reports relevant processes, principles, and techniques from the testing literature, and includes many useful classroom materials. The Pezze and Young text presents coverage criteria in the traditional lifecycle-based manner, and does not organize criteria into the four abstract models discussed in this chapter.

Numerous other software testing books were not intended as textbooks, or do not offer general coverage for classroom use. Beizer's Software System Testing and Quality Assurance [2] and Hetzel's The Complete Guide to Software Testing [22] cover various aspects of management and process for software testing. Several books cover specific aspects of testing [25, 30, 43]. The STEP project at Georgia Institute of Technology resulted in a comprehensive survey of the practice of software testing by Department of Defense contractors in the 1980s [11].

The definition of *unit* is from Stevens, Myers and Constantine [46], and the definition of *module* is from Sommerville [44]. The definition of *integration testing* is from Beizer [3]. The clarification for OO testing levels with the terms *intra-method*, *inter-method*, and *intra-class* testing is from Harrold and Rothermel [21] and *inter-class* testing is from Gallagher, Offutt and Cincotta [16].

The information for the Pentium bug and Mars lander was taken from several sources, including by Edelman, Moler, Nuseibeh, Knutson, and Peterson [13, 28, 31, 35, 40]. The accident report [29] is the best source for understanding the details of the Ariane 5 Flight 501 Failure.

The testing levels in Section 1.1.2 were first defined by Beizer [3].

The elementary result that finding all failures in a program is undecidable is due to Howden [23].

Most of the terminology in testing is from standards documents, including the IEEE Standard Glossary of Software Engineering Terminology [26], the US Department of Defense [36, 37], the US Federal Aviation Administration FAA-DO178B, and the British Computer Society's Standard for Software Component Testing [45]. The definitions for observability and controllability come from Freedman [15]. Similar definitions were also given in Binder's book, *Testing Object-oriented Systems* [4].

The fault/failure model was developed independently by Offutt and Morell in their dissertations [12, 32, 33, 38]. Morell used the terms execution, infection, and propagation [33, 32], and Offutt used reachability, sufficiency, and necessity [12, 38]. This book merges the two sets of terms by using what we consider to be the most descriptive terms.

The multiple parts of the test case that we use are based on research in test case specifications [1, 47].

One of the first discussions of infeasibility from other than a purely theoretical view was by Frankl and Weyuker [14]. The problem was shown to be undecidable by Goldberg et al. [17] and DeMillo and Offutt [12]. Some partial solutions have been presented [16, 17, 27, 39].

Budd and Angluin [5] analyzed the theoretical distinctions between generators and recognizers from a testing viewpoint. They showed that both problems are formally undecidable, and discussed tradeoffs in approximating the two.

Subsumption has been widely used as a way to analytically compare testing techniques. We follow Weiss [50] and Frankl and Weyuker [14] for our definition of subsumption. Frankl and Weyuker actually used the term *includes.* The term subsumption was defined by Clarke et al.: A criterion C_1 *subsumes* a criterion C_2 if and only if every set of execution paths P that satisfies C_1 also satisfies C_2 [7]. The term subsumption is currently the more widely used and the two definitions are equivalent; this book follows Weiss's suggestion to use the term *subsumes* to refer to Frankl and Weyuker's definition.

The descriptions of excise and revenue tasks were taken from Cooper [8].

Although this book does not focus heavily on the theoretical underpinnings of software testing, students interested in research should study such topics more in depth. A number of the papers are quite old, often do not appear in current literature, and their ideas are beginning to disappear. The authors encourage the study of the older papers. Among those are truly seminal papers in the 1970s by Goodenough and Gerhart [18] and Howden [23], and Demillo, Lipton, Sayward and Perlis [9, 10]. These papers were followed up and refined by Weyuker and Ostrand [51], Hamlet [20], Budd and Angluin [5], Gourlay [19], Prather [42], Howden [24], and Cherniavsky and Smith [6]. Later theoretical papers were contributed by Morell [33], Zhu [53] and Wah [48, 49]. Every PhD student's adviser will certainly have his or her own favorite theoretical papers, but this list should provide a good starting point.

Bibliography

[1] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218, Key West Florida, December 1989. ACM SIGSOFT 89.

[2] Boris Beizer. *Software System Testing and Quality Assurance*. Van Nostrand, New York, 1984.

[3] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[4] Robert Binder. *Testing Object-oriented Systems*. Addison-Wesley Publishing Company Inc., New York, New York, 2000.

[5] Tim Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[6] J. C. Cherniavsky and C. H. Smith. A theory of program testing with applications. *Workshop on Software Testing*, pages 110–121, July 1986.

[7] Lori Clarke, Andy Podgurski, Debra Richardson, and Steven Zeil. A comparison of data flow path selection criteria. In *Eighth International Conference on Software Engineering*, pages 244–251, London UK, August 1985. IEEE Computer Society Press.

[8] Alan Cooper. *About Face: The Essentials of User Interface Design*. Hungry Minds, New York, 1995.

[9] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5), May 1979.

[10] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[11] Richard A. DeMillo, W. Michael McCracken, Rhonda J. Martin, and John F. Passafiume. *Software Testing and Evaluation*. Benjamin/Cummings, Menlo Park CA, 1987.

[12] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[13] Alan Edelman. The mathematics of the Pentium division bug. *SIAM Review*, 39:54–67, March 1997. http://www.siam.org/journals/sirev/39-1/29395.html.

[14] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[15] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, June 1991.

[16] Leonard Gallagher, Jeff Offutt, and Tony Cincotta. Integration testing of object-oriented components using finite state machines. *Software Testing, Verification, and Reliability*, 17(1):215–266, January 2007.

[17] A. Goldberg, T. C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *1994 International Symposium on Software Testing, and Analysis*, pages 80–94, Seattle WA, August 1994.

[18] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2), June 1975.

[19] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, November 1983.

[20] Richard Hamlet. Reliability theory of program testing. *Acta Informatica*, pages 31–43, 1981.

[21] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Symposium on Foundations of Software Engineering*, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.

[22] Bill Hetzel. *The Complete Guide to Software Testing*. Wiley-QED, second edition, 1988.

[23] Willam E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.

[24] Willam E. Howden. The theory and practice of function testing. *IEEE Software*, 2(5), September 1985.

[25] William E. Howden. *Functional Program Testing and Analysis*. McGraw-Hill Book Company, New York NY, 1987.

[26] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 610.12-1990, 1996.

[27] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *1994 International Symposium on Software Testing, and Analysis*, pages 95–107, Seattle WA, August 1994.

[28] Charles Knutson and Sam Carmichael. Safety first: Avoiding software mishaps, November 2000. http://www.embedded.com/2000/0011/0011feat1.htm.

[29] J. L. Lions. Ariane 5 flight 501 failure: Report by the inquiry board, July 1996. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[30] Brian Marick. *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing.* Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[31] Cleve Moler. A tale of two numbers. *SIAM News*, 28(1), January 1995.

[32] Larry J. Morell. *A Theory of Error-Based Testing.* PhD thesis, University of Maryland, College Park MD, 1984. Technical Report TR-1395.

[33] Larry J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[34] Glenford Myers. *The Art of Software Testing.* John Wiley and Sons, New York NY, 1979.

[35] Bashar Nuseibeh. Who dunnit? *IEEE Software*, 14:15–16, May/June 1997.

[36] Department of Defense. *DOD-STD-2167A: Defense System Software Development.* Department of Defense, February 1988.

[37] Department of Defense. *MIL-STD-498: Software Development and Documentation.* Department of Defense, December 1994.

[38] Jeff Offutt. *Automatic Test Data Generation.* PhD thesis, Georgia Institute of Technology, Atlanta GA, 1988. Technical report GIT-ICS 88/28.

[39] Jeff Offutt and Jie Pan. Detecting equivalent mutants and the feasible path problem. *Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.

[40] Ivars Peterson. Pentium bug revisited, May 1997. http://www.maa.org/mathland/mathland_5_12.html.

[41] Mauro Pezze and Michal Young. *Software Testing and Analysis: Process, Principles, and Techniques.* Wiley, Hoboken, NJ, 2008.

[42] R. E. Prather. Theory of program testing – an overview. *The Bell System Technical Journal*, 62(10), December 1983.

[43] Marc Roper. *Software Testing.* International Software Quality Assurance Series. McGraw-Hill, 1994.

[44] Ian Sommerville. *Software Engineering.* Addison-Wesley Publishing Company Inc., 6th edition, 2001.

[45] British Computer Society Specialist Interest Group in Software Testing. *Standard for Software Component Testing, Working Draft 3.3.* British Computer Society, 1997. http://www.rmcs.cranfield.ac.uk/∼cised/sreid/BCS_SIG/.

[46] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[47] Phil Stocks and Dave Carrington. Test Templates: A Specification-Based Testing Framework. In *Fifteenth International Conference on Software Engineering*, pages 405–414, Baltimore, MD, May 1993.

[48] K. S. How Tai Wah. Fault coupling in finite bijective functions. *Software Testing, Verification, and Reliability*, 5(1):3–47, March 1995.

[49] K. S. How Tai Wah. A theoretical study of fault coupling. *Software Testing, Verification, and Reliability*, 10(1):3–46, March 2000.

[50] Steward N. Weiss. What to compare when comparing test data adequacy criteria. *ACM SIGSOFT Notes*, 14(6):42–49, October 1989.

[51] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

[52] Lee J. White. Software testing and verification. In Marshall C. Yovits, editor, *Advances in Computers*, volume 26, pages 335–390. Academic Press, Inc, 1987.

[53] Hong Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, April 1996.

[54] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.