1

Why Do We Test Software?

The true subject matter of the tester is not testing, but the design of test cases.

The purpose of this book is to teach software engineers how to test. This knowledge is useful whether you are a programmer who needs to unit test your own software, a full-time tester who works mostly from requirements at the user level, a manager in charge of testing or development, or any position in between. As the software industry moves into the second decade of the 21st century, software quality is increasingly becoming essential to all businesses and knowledge of software testing is becoming necessary for all software engineers.

Today, software defines behaviors that our civilization depends on in systems such as network routers, financial calculation engines, switching networks, the Web, power grids, transportation systems, and essential communications, command, and control services. Over the past two decades, the software industry has become much bigger, is more competitive, and has more users. Software is an essential component of exotic embedded 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, mobile phones, and remote controllers. Modern households have hundreds of processors, and new cars have over a thousand; 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 way industry evaluates software during development. The recent growth in agile processes puts increased pressure on testing; unit testing is emphasized heavily and test-driven development makes tests key to functional requirements. It is clear that industry is deep into a revolution in what testing means to the success of software products.

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 students and practicing engineers 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 of testing techniques present in the literature have much more in common than is obvious at first glance. Testing techniques are typically 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 underlying similarities among techniques.

This book clarifies these similarities with two innovative, yet simplifying, approaches. First, we show how testing is more efficient and effective by using a classical engineering approach. Instead of designing and developing tests on concrete software artifacts like the source code or requirements, we show how to develop abstraction models, design tests at the abstract level, and then implement actual tests at the concrete level by satisfying the abstract designs. This is the exact process that traditional engineers use, except whereas they usually use calculus and algebra to describe the abstract models, software engineers usually use discrete mathematics. Second, we recognize that all test criteria can be defined with a very short list of abstract models: input domain characterizations, graphs, logical expressions, and syntactic descriptions. These are directly reflected in the four chapters of Part II of this book.

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 all software developers.

1.1 WHEN SOFTWARE GOES BAD

As said, we consider the development of software to be engineering. And like any engineering discipline, the software industry has its shares of failures, some spectacular, some mundane, some costly, and sadly, some that have resulted in loss of life. Before learning about software disasters, it is important to understand the difference between faults, errors, and failures. We adopt the definitions of software fault, error, and failure from the dependability community.

Definition 1.1 Software Fault: A static defect in the software.

Definition 1.2 Software Error: An incorrect internal state that is the manifestation of some fault.

Definition 1.3 Software Failure: External, incorrect behavior with respect to the requirements or another description of the expected behavior.

Consider a medical doctor diagnosing 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 symptoms. 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 exist as a result of a 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 limits 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 can never 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 precise example of the definitions of fault, error, and failure, we need to clarify the concept of the state. A *program state* is defined during execution of a program as the current value of all live variables and the current location, as given by the program counter. The *program counter* (PC) is the next statement in the program to be executed and can be described with a line number in the file ($PC = 5$) or the statement as a string ($PC =$ *"if (x > y)"*). Most of the time, what we mean by a statement is obvious, but complex structures such as *for* loops have to be treated specially. The program line "*for (i=1; i<N; i++)*" actually has three statements that can result in separate states. The loop initialization ("*i=1*") is separate from the loop test ("*i<N*"), and the loop increment ("*i++*") occurs at the end of the loop body. As an illustrative example, consider the following Java method:

```
/**
 * Counts zeroes in an array
 *
 * @param x array to count zeroes in
 * @return number of occurrences of 0 in x
 * @throws NullPointerException if x is null
 */
public static int numZero (int[] x)
{
  int count = 0;
  for (int i = 1; i < x.length; i++)
  {
    if (x[i] == 0) count++;
  }
  return count;
}
```

Sidebar

Programming Language Independence

This book strives to be independent of language, and most of the concepts in the book are. At the same time, we want to illustrate these concepts with specific examples. We choose Java, and emphasize that most of these examples would be very similar in many other common languages.

The fault in this method 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 tests the faulty statement is executed. Although both of these tests result in an error, only the second results in failure. To understand the error states, we need to identify the state for the method. The state for `numZero()` consists of values for the variables `x`, `count`, `i`, and the program counter (`PC`). For the first example above, the state at the loop test on the very first iteration of the loop is (`x` = [2, 7, 0], `count` = 0, `i` = 1, `PC` = "`i < x.length`"). Notice that this state is erroneous 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 $s_0, s_1, s_2, \ldots$, and the actual sequence of states is $s_0, s_2, s_3, \ldots$, then state s_2 is in error in the second sequence. The fault model described here is quite deep, and this discussion gives the broad view without going into unneeded details. The exercises at the end of the section explore some of the subtleties of the fault model.

In the second test for our example, the error state is (`x` = [0, 7, 2], `count` = 0, `i` = 1, `PC` = "`i < x.length`"). 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 term *bug* is often used informally to refer to all three of fault, error, and failure. This book will usually use the specific term, and avoid using "bug." A favorite story of software engineering teachers is that Grace Hopper found a moth stuck in a relay on an early computing machine, which started the use of bug as a problem with software. It is worth noting, however, that the term bug has an old and rich history, predating software by at least a century. The first use of bug to generally mean a problem we were able to find is from a quote by Thomas Edison:

It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise–this thing gives out and [it is] then that 'Bugs'–as such little faults and difficulties are called–show themselves and months of intense watching, study and labor are requisite.

— Thomas Edison

A very 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 of the most famous cases of software killing people is the Therac-25 radiation therapy machine. Software faults were found to have caused at least three deaths due to excessive radiation. Another dramatic failure was the launch failure of the first Ariane 5 rocket, which exploded 37 seconds after liftoff in 1996. The low-level cause was an unhandled floating point conversion exception in an inertial guidance system function. It turned out that the guidance system could never encounter the unhandled exception when used on the Ariane 4 rocket. That is, the guidance system function was 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!

The famous Pentium bug was an early alarm of the need for better testing, especially unit testing. Intel introduced its Pentium microprocessor in 1994, 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 an essential point. 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. Thus, this would have been 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 great northeast blackout of 2003 was started when a power line in Ohio brushed against overgrown trees and shut down. This is called a *fault* in the power industry. Unfortunately, the software alarm system failed in the local power company, so system operators could not understand what happened. Other lines also sagged into trees and switched off, eventually overloading other power lines, which then cut off. This cascade effect eventually caused a blackout throughout southeastern Canada and eight states in the northeastern part of the US. This is considered the biggest blackout in North American history, affecting 10 million people in Canada and 40 million in the USA, contributing to at least 11 deaths and costing up to $6 billion USD.

Some software failures are felt widely enough to cause severe embarrassment to the company. In 2011, a centralized students data management system in Korea miscalculated the academic grades of over 29,000 middle and high school students.

This led to massive confusion about college admissions and a government investigation into the software engineering practices of the software vendor, Samsung Electronics.

A 1999 study commissioned by the U.S. National Research Council and the U.S. President's commission on critical infrastructure protection concluded that the current base of science and technology is inadequate for building systems to control critical software infrastructure. A 2002 report commissioned by the National Institute of Standards and Technology (NIST) estimated that defective software costs the U.S. economy $59.5 billion per year. The report further estimated that 64% of the costs were a result of user mistakes and 36% a result of design and development mistakes, and suggested that improvements in testing could reduce this cost by about a third, or $22.5 billion. Blumenstyk reported that web application failures lead to huge losses in businesses; $150,000 per hour in media companies, $2.4 million per hour in credit card sales, and $6.5 million per hour in the financial services market.

Software faults do not just lead to functional failures. According to a Symantec security threat report in 2007, 61 percent of all vulnerabilities disclosed were due to faulty software. The most common are web application vulnerabilities that can be attacked by some common attack techniques using invalid inputs.

These public and expensive software failures are getting more common and more widely known. This is simply a symptom of the change in expectations of software. As we move further into the 21st century, we are using more safety critical, real-time software. Embedded software has become ubiquitous; many of us carry millions of lines of embedded software in our pockets. Corporations rely more and more on large-scale enterprise applications, which by definition have large user bases and high reliability requirements. Security, which used to depend on cryptography, then database security, then avoiding network vulnerabilities, is now largely about avoiding software faults. The Web has had a major impact. It features a deployment platform that offers software services that are very competitive and available to millions of users. They are also distributed, adding complexity, and must be highly reliable to be competitive. More so than at any previous time, industry desperately needs to apply the accumulated knowledge of over 30 years of testing research.

1.2 GOALS OF TESTING SOFTWARE

Surprisingly, many software engineers are not clear about their testing goals. Is it to show correctness, find problems, or something else? To explore this concept, we first must separate validation and verification. Most of the definitions in this book 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.

> *Definition 1.4 Verification:* The process of determining whether the products of a phase of the software development process fulfill the requirements established during the previous phase.

Definition 1.5 Validation: The process of evaluating software at the end of software development to ensure compliance with intended usage.

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.

As a familiar example, consider a light switch in a conference room. Verification asks if the lighting meets the specifications. The specifications might say something like, "The lights in front of the projector screen can be controlled independently of the other lights in the room." If the specifications are written down somewhere and the lights **cannot** be controlled independently, then the lighting fails verification, precisely because the implementation does not satisfy the specifications. Validation asks whether users are satisfied, an inherently fuzzy question that has nothing to do with verification. If the "independent control" specification is neither written down nor satisfied, then, despite the disappointed users, verification nonetheless succeeds, because the implementation satisfies the specification. But validation fails, because the specification for the lighting does not reflect the true needs of the users. This is an important general point: validation exposes flaws in specifications.

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. This book emphasizes verification more than validation, although most of the specific test criteria we discuss can be used for both activities.

Beizer discussed the goals of testing in terms of the "test process maturity levels" of an organization, where the levels are characterized by the testers' goals. He defined five levels, where the lowest level is not worthy of being given a number.

Level 0 There is no difference between testing and debugging.

Level 1 The purpose of testing is to show correctness.

Level 2 The purpose of testing is to show that the software does not 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 weak 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 inherently negative. Testers may enjoy finding the problem, but the developers never want to find problems–they want the software to work (yes, level 1 thinking can be 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. It is our view that this level currently dominates the software industry.

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. We see more and more companies move to this testing maturity level every year.

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. Beizer used the analogy 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 be the experts who 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 remaining chapters in Part I 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. An 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**. Although level 4 thinking is currently rare in the software industry, it is common in more mature engineering fields.

These considerations help us decide at a strategic level why we test. At a more tactical level, it is important to know why **each test** is present. If you do not know why you are conducting each test, the test will not be very helpful. What fact is each test trying to verify? It is essential to document test objectives and test requirements, including the planned coverage levels. When the test manager attends a planning meeting with the other managers and the project manager, the test manager must be able to articulate clearly how much testing is enough and when testing will complete. In the 1990s, we could use the "date criterion," that is, testing is "complete" when the ship date arrives or when the budget is spent.

Figure 1.1 dramatically illustrates the advantages of testing early rather than late. This chart is based on a detailed analysis of faults that were detected and fixed during several large government contracts. The bars marked '**A**' indicate what percentage of faults *appeared* in that phase. Thus, 10% of faults appeared during the requirements phase, 40% during design, and 50% during implementation. The bars marked '**D**' indicated the percentage of faults that were *detected* during each phase. About 5% were detected during the requirements phase, and over 35% during system testing. Lastly is the cost analysis. The solid bars marked '**C**' indicate the relative *cost* of finding and fixing faults during each phase. Since each project was different, this is averaged to be based on a "unit cost." Thus, faults detected and fixed during requirements, design, and unit testing were a single unit cost. Faults detected and fixed during integration testing cost five times as much, 10 times as much during system testing, and 50 times as much after the software is deployed.

If we take the simple assumption of $1000 USD unit cost per fault, and 100 faults, that means we spend $39,000 to find and correct faults during requirements, design, and unit testing. During integration testing, the cost goes up to $100,000. But system testing and deployment are the serious problems. We find more faults during system testing at ten times the cost, for a total of $360,000. And even though we only find a few faults after deployment, the cost being 50 X unit means we spend $250,000! Avoiding the work early (requirements analysis and unit testing) saves money in the short term. But it leaves faults in software that are like little bombs, ticking away, and the longer they tick, the bigger the explosion when they finally go off.

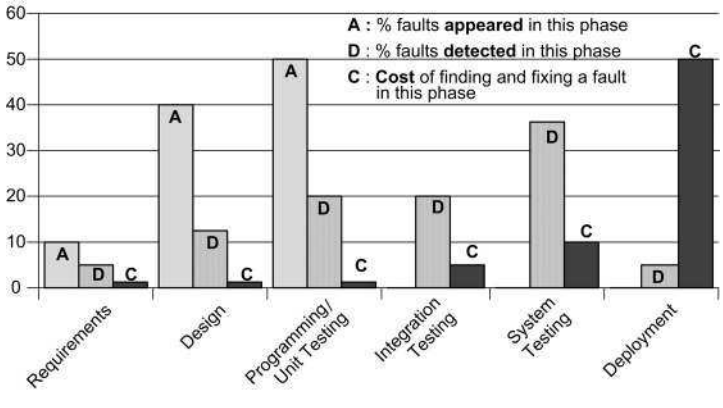

Figure 1.1. Cost of late testing.

To put Beizer's level 4 test maturity level in simple terms, the goal of testing is to eliminate faults as **early** as possible. We can never be perfect, but every time we eliminate a fault during unit testing (or sooner!), we save money. The rest of this book will teach you how to do that.

EXERCISES
Chapter 1.

1. What are some 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. What is the difference between software **fault** and software **failure**?
3. What do we mean by "*level 3 thinking is that the purpose of testing is to reduce risk?*" What risk? Can we reduce the risk to zero?
4. 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. Describe 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.

[1] Liskov's *Program Development in Java*, especially chapters 9 and 10, is a great source for students who wish to learn more about this.