In Chapter 1, “Software Testing Background,” and Chapter 2, “The Software Development Process,” you learned about the basics of software testing and the software development process. The information presented in these chapters offered a very high-level and arguably idealistic view of how software projects might be run. Unfortunately, in the real world you will never see a project perfectly follow any of the development models. You will never be given a thoroughly detailed specification that perfectly meets the customer’s needs and you will never have enough time to do all the testing you need to do. It just doesn’t happen. But, to be an effective software tester, you need to understand what the ideal process is so that you have something to aim for.

The goal of this chapter is to temper that idealism with a reality check from a software tester’s perspective. It will help you see that, in practice, trade-offs and concessions must be made throughout the development cycle. Many of those trade-offs are directly related to the software test effort. The bugs you find and the problems you prevent all significantly affect the project. After reading this chapter, you’ll have a much clearer picture of the roles, the impact, and the responsibilities that software testing has and you’ll hopefully appreciate the behind-the-scenes decisions that must be made to create a software product.

The highlights of this chapter include

- Why software can never be perfect
- Why software testing isn’t just a technical problem
- The terms commonly used by software testers
Testing Axioms

This first section of this chapter is a list of axioms, or truisms. Think of them as the “rules of the road” or the “facts of life” for software testing and software development. Each of them is a little tidbit of knowledge that helps put some aspect of the overall process into perspective.

It’s Impossible to Test a Program Completely

As a new tester, you might believe that you can approach a piece of software, fully test it, find all the bugs, and assure that the software is perfect. Unfortunately, this isn’t possible, even with the simplest programs, due to four key reasons:

- The number of possible inputs is very large.
- The number of possible outputs is very large.
- The number of paths through the software is very large.
- The software specification is subjective. You might say that a bug is in the eye of the beholder.

Multiply all these “very large” possibilities together and you get a set of test conditions that’s too large to attempt. If you don’t believe it, consider the example shown in Figure 3.1, the Microsoft Windows Calculator.

Assume that you are assigned to test the Windows Calculator. You decide to start with addition. You try \(1+0=\). You get an answer of 1. That’s correct. Then you try \(1+1=\). You get 2. How far do you go? The calculator accepts a 32-digit number, so you must try all the possibilities up to

\[1+99999999999999999999999999999999=\]
Once you complete that series, you can move on to \(2 + 0 =\), \(2 + 1 =\), \(2 + 2 =\), and so on. Eventually you’ll get to

\[99999999999999999999999999999999 + 99999999999999999999999999999999 =\]

Next you should try all the decimal values: \(1.0 + 0.1\), \(1.0 + 0.2\), and so on.

Once you verify that regular numbers sum properly, you need to attempt illegal inputs to assure that they’re properly handled. Remember, you’re not limited to clicking the numbers onscreen—you can press keys on your computer keyboard, too. Good values to try might be \(1 + a\), \(2 + 1\), \(1a1 + 2b2\),…. There are literally billions upon billions of these.

Edited inputs must also be tested. The Windows Calculator allows the Backspace and Delete keys, so you should try them. \(1 <\text{backspace}> 2 + 2\) should equal 4. Everything you’ve tested so far must be retested by pressing the Backspace key for each entry, for each two entries, and so on.

If you or your heirs manage to complete all these cases, you can then move on to adding three numbers, then four numbers,….

There are so many possible entries that you could never complete them, even if you used a supercomputer to feed in the numbers. And that’s only for addition. You still have subtraction, multiplication, division, square root, percentage, and inverse to cover.

The point of this example is to demonstrate that it’s impossible to completely test a program, even software as simple as a calculator. If you decide to eliminate any of the test conditions because you feel they’re redundant or unnecessary, or just to save time, you’ve decided not to test the program completely.

**Software Testing Is a Risk-Based Exercise**

If you decide not to test every possible test scenario, you’ve chosen to take on risk. In the calculator example, what if you choose not to test that \(1024 + 1024 = 2048\)? It’s possible the programmer accidentally left in a bug for that situation. If you don’t test it, a customer will eventually enter it, and he or she will discover the bug. It’ll be a costly bug, too, since it wasn’t found until the software was in the customer’s hands.

This may all sound pretty scary. You can’t test everything, and if you don’t, you will likely miss bugs. The product has to be released, so you will need to stop testing, but if you stop too soon, there will still be areas untested. What do you do?

One key concept that software testers need to learn is how to reduce the huge domain of possible tests into a manageable set, and how to make wise risk-based decisions on what’s important to test and what’s not.
Figure 3.2 shows the relationship between the amount of testing performed and the number of bugs found. If you attempt to test everything, the costs go up dramatically and the number of missed bugs declines to the point that it’s no longer cost effective to continue. If you cut the testing short or make poor decisions of what to test, the costs are low but you’ll miss a lot of bugs. The goal is to hit that optimal amount of testing so that you don’t test too much or too little.

**FIGURE 3.2** Every software project has an optimal test effort.

You will learn how to design and select test scenarios that minimize risk and optimize your testing in Chapters 4 through 7.

**Testing Can’t Show That Bugs Don’t Exist**

Think about this for a moment. You’re an exterminator charged with examining a house for bugs. You inspect the house and find evidence of bugs—maybe live bugs, dead bugs, or nests. You can safely say that the house has bugs.

You visit another house. This time you find no evidence of bugs. You look in all the obvious places and see no signs of an infestation. Maybe you find a few dead bugs or old nests but you see nothing that tells you that live bugs exist. Can you absolutely, positively state that the house is bug free? Nope. All you can conclude is that in your search you didn’t find any live bugs. Unless you completely dismantled the
house down to the foundation, you can’t be sure that you didn’t simply just miss them.

Software testing works exactly as the exterminator does. It can show that bugs exist, but it can’t show that bugs don’t exist. You can perform your tests, find and report bugs, but at no point can you guarantee that there are no longer any bugs to find. You can only continue your testing and possibly find more.

**The More Bugs You Find, the More Bugs There Are**

There are even more similarities between real bugs and software bugs. Both types tend to come in groups. If you see one, odds are there will be more nearby.

Frequently, a tester will go for long spells without finding a bug. He’ll then find one bug, then quickly another and another. There are several reasons for this:

- **Programmers have bad days.** Like all of us, programmers can have off days. Code written one day may be perfect; code written another may be sloppy. One bug can be a tell-tale sign that there are more nearby.

- **Programmers often make the same mistake.** Everyone has habits. A programmer who is prone to a certain error will often repeat it.

- **Some bugs are really just the tip of the iceberg.** Very often the software’s design or architecture has a fundamental problem. A tester will find several bugs that at first may seem unrelated but eventually are discovered to have one primary serious cause.

It’s important to note that the inverse of this “bugs follow bugs” idea is true, as well. If you fail to find bugs no matter how hard you try, it may very well be that the feature you’re testing was cleanly written and that there are indeed few if any bugs to be found.

**The Pesticide Paradox**

In 1990, Boris Beizer, in his book *Software Testing Techniques*, Second Edition, coined the term *pesticide paradox* to describe the phenomenon that the more you test software, the more immune it becomes to your tests. The same thing happens to insects with pesticides (see Figure 3.3). If you keep applying the same pesticide, the insects eventually build up resistance and the pesticide no longer works.

Remember the spiral model of software development described in Chapter 2? The test process repeats each time around the loop. With each iteration, the software testers receive the software for testing and run their tests. Eventually, after several passes, all the bugs that those tests would find are exposed. Continuing to run them won’t reveal anything new.
FIGURE 3.3 Software undergoing the same repetitive tests eventually builds up resistance to them.

To overcome the pesticide paradox, software testers must continually write new and different tests to exercise different parts of the program and find more bugs.

Not All the Bugs You Find Will Be Fixed
One of the sad realities of software testing is that even after all your hard work, not every bug you find will be fixed. Now, don’t be disappointed—this doesn’t mean that you’ve failed in achieving your goal as a software tester, nor does it mean that you or your team will release a poor quality product. It does mean, however, that you’ll need to rely on a couple of those traits of a software tester listed in Chapter 1—exercising good judgment and knowing when perfection isn’t reasonably attainable. You and your team will need to make trade-offs, risk-based decisions for each and every bug, deciding which ones will be fixed and which ones won’t.

There are several reasons why you might choose not to fix a bug:

- **There’s not enough time.** In every project there are always too many software features, too few people to code and test them, and not enough room left in the schedule to finish. If you’re working on a tax preparation program, April 15 isn’t going to move—you must have your software ready in time.

- **It’s really not a bug.** Maybe you’ve heard the phrase, “It’s not a bug, it’s a feature!” It’s not uncommon for misunderstandings, test errors, or spec changes to result in would-be bugs being dismissed as features.

- **It’s too risky to fix.** Unfortunately, this is all too often true. Software can be fragile, intertwined, and sometimes like spaghetti. You might make a bug fix
that causes other bugs to appear. Under the pressure to release a product under a tight schedule, it might be too risky to change the software. It may be better to leave in the known bug to avoid the risk of creating new, unknown ones.

- **It's just not worth it.** This may sound harsh, but it's reality. Bugs that would occur infrequently or bugs that appear in little-used features may be dismissed. Bugs that have work-arounds, ways that a user can prevent or avoid the bug, are often not fixed. It all comes down to a business decision based on risk.

The decision-making process usually involves the software testers, the project managers, and the programmers. Each carries a unique perspective on the bugs and has his own information and opinions as to why they should or shouldn't be fixed. In Chapter 19, “Reporting What You Find,” you’ll learn more about reporting bugs and getting your voice heard.

**WHAT HAPPENS WHEN YOU MAKE THE WRONG DECISION?**

Remember the Intel Pentium bug described in Chapter 1? The Intel test engineers found this bug before the chip was released, but the product team decided that it was such a small, rare bug that it wasn’t worth fixing. They were under a tight schedule and decided to meet their current deadline and fix the bug in later releases of the chip. Unfortunately, the bug was discovered and the rest, they say, is history.

In any piece of software, for every “Pentium type” bug that makes the headlines, there could be perhaps hundreds of bugs that are left unfixed because they were perceived to not have a major negative effect. Only time can tell if those decisions were right or wrong.

**When a Bug’s a Bug Is Difficult to Say**

If there’s a problem in the software but no one ever discovers it—not programmers, not testers, and not even a single customer—is it a bug?

Get a group of software testers in a room and ask them this question. You’ll be in for a lively discussion. Everyone has their own opinion and can be pretty vocal about it. The problem is that there’s no definitive answer. The answer is based on what you and your development team decide works best for you.

For the purposes of this book, refer back to the rules to define a bug from Chapter 1:

1. The software doesn’t do something that the product specification says it should do.

2. The software does something that the product specification says it shouldn’t do.

3. The software does something that the product specification doesn’t mention.
4. The software doesn’t do something that the product specification doesn’t mention but should.

5. The software is difficult to understand, hard to use, slow, or—in the software tester’s eyes—will be viewed by the end user as just plain not right.

Following these rules helps clarify the dilemma by making a bug a bug only if it’s observed. To claim that the software does or doesn’t do “something” implies that the software was run and that “something” or the lack of “something” was witnessed. Since you can’t report on what you didn’t see, you can’t claim that a bug exists if you didn’t see it.

Here’s another way to think of it. It’s not uncommon for two people to have completely different opinions on the quality of a software product. One may say that the program is incredibly buggy and the other may say that it’s perfect. How can both be right? The answer is that one has used the product in a way that reveals lots of bugs. The other hasn’t.

NOTE

Bugs that are undiscovered or haven’t yet been observed are often referred to as latent bugs.

If this is as clear as mud, don’t worry. Discuss it with your peers in software testing and find out what they think. Listen to others’ opinions, test their ideas, and form your own definition. Remember the old question, “If a tree falls in the forest and there’s no one there to hear it, does it make a sound?”

Product Specifications Are Never Final

Software developers have a problem. The industry is moving so fast that last year’s cutting-edge products are obsolete this year. At the same time, software is getting larger and gaining more features and complexity, resulting in longer and longer development schedules. These two opposing forces result in conflict, and the result is a constantly changing product specification.

There’s no other way to respond to the rapid changes. Assume that your product had a locked-down, final, absolutely-can’t-change-it product spec. You’re halfway through the planned two-year development cycle, and your main competitor releases a product very similar to yours but with several desirable features that your product doesn’t have. Do you continue with your spec as is and release an inferior product in another year? Or, does your team regroup, rethink the product’s features, rewrite the product spec, and work on a revised product? In most cases, wise business dictates the latter.
As a software tester, you must assume that the spec will change. Features will be added that you didn’t plan to test. Features will be changed or even deleted that you had already tested and reported bugs on. It will happen. You’ll learn techniques for being flexible in your test planning and test execution in the remainder of this book.

**Software Testers Aren’t the Most Popular Members of a Project Team**

Remember the goal of a software tester?

> The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.

Your job is to inspect and critique your peer’s work, find problems with it, and publicize what you’ve found. Ouch! You won’t win a popularity contest doing this job.

Here are a couple of tips to keep the peace with your fellow teammates:

- **Find bugs early.** That’s your job, of course, but work hard at doing this. It’s much less of an impact and much more appreciated if you find a serious bug three months before, rather than one day before, a product’s scheduled release.

- **Temper your enthusiasm.** Okay, you really love your job. You get really excited when you find a terrible bug. But, if you bounce into a programmer’s cubicle with a huge grin on your face and tell her that you just found the nastiest bug of your career and it’s in her code, she won’t be happy.

- **Don’t just report bad news.** If you find a piece of code surprisingly bug free, tell the world. Pop into a programmer’s cubicle occasionally just to chat. If all you ever do is report bad news, people will see you coming and will run and hide.

**Software Testing Is a Disciplined Technical Profession**

It used to be that software testing was an afterthought. Software products were small and not very complicated. The number of people with computers using software was limited. And, the few programmers on a project team could take turns debugging each others’ code. Bugs weren’t that much of a problem. The ones that did occur were easily fixed without much cost or disruption. If software testers were used, they were frequently untrained and brought into the project late to do some “ad-hoc banging on the code to see what they might find.” Times have changed.

Look at the software help-wanted ads and you’ll see numerous listings for software testers. The software industry has progressed to the point where professional software testers are mandatory. It’s now too costly to build bad software.
To be fair, not every company is on board yet. Many computer game and small-time software companies still use a fairly loose development model—usually big-bang or code-and-fix. But much more software is now developed with a disciplined approach that has software testers as core, vital members of their staff.

This is great news if you’re interested in software testing. It can now be a career choice—a job that requires training and discipline, and allows for advancement.

Software Testing Terms and Definitions

This chapter wraps up the first section of this book with a list of software testing terms and their definitions. These terms describe fundamental concepts regarding the software development process and software testing. Because they’re often confused or used inappropriately, they’re defined here as pairs to help you understand their true meanings and the differences between them. Be aware that there is little agreement in the software industry over the definition of many, seemingly common, terms. As a tester, you should frequently clarify the meaning of the terms your team is using. It’s often best to agree to a definition rather than fight for a “correct” one.

Precision and Accuracy

As a software tester, it’s important to know the difference between precision and accuracy. Suppose that you’re testing a calculator. Should you test that the answers it returns are precise or accurate? Both? If the project schedule forced you to make a risk-based decision to focus on only one of these, which one would you choose?

What if the software you’re testing is a simulation game such as baseball or a flight simulator? Should you primarily test its precision or its accuracy?

Figure 3.4 helps to graphically describe these two terms. The goal of this dart game is to hit the bull’s-eye in the center of the board. The darts on the board in the upper left are neither precise nor accurate. They aren’t closely grouped and not even close to the center of the target.

The board on the upper right shows darts that are precise but not accurate. They are closely grouped, so the thrower has precision, but he’s not very accurate because the darts didn’t even hit the board.

The board on the lower left is an example of accuracy but poor precision. The darts are very close to the center, so the thrower is getting close to what he’s aiming at, but because they aren’t closely positioned, the precision is off.

The board in the lower right is a perfect match of precision and accuracy. The darts are closely grouped and on target.
FIGURE 3.4  Darts on a dartboard demonstrate the difference between precision and accuracy.

Whether the software you test needs to be precise or accurate depends much on what the product is and ultimately what the development team is aiming at (excuse the pun). A software calculator likely demands that both are achieved—a right answer is a right answer. But, it may be decided that calculations will only be accurate and precise to the fifth decimal place. As long as the testers are aware of that specification, they can tailor their testing to confirm it.

Verification and Validation  

Verification and validation are often used interchangeably but have different definitions. These differences are important to software testing.

Verification is the process confirming that something—software—meets its specification. Validation is the process confirming that it meets the user’s requirements. These may sound very similar, but an explanation of the Hubble space telescope problems will help show the difference.

In April 1990, the Hubble space telescope was launched into orbit around the Earth. As a reflective telescope, Hubble uses a large mirror as its primary means to magnify the objects it’s aiming at. The construction of the mirror was a huge undertaking requiring extreme precision and accuracy. Testing of the mirror was difficult since
the telescope was designed for use in space and couldn’t be positioned or even viewed through while it was still on Earth. For this reason, the only means to test it was to carefully measure all its attributes and compare the measurements with what was specified. This testing was performed and Hubble was declared fit for launch.

Unfortunately, soon after it was put into operation, the images it returned were found to be out of focus. An investigation discovered that the mirror was improperly manufactured. The mirror was ground according to the specification, but the specification was wrong. The mirror was extremely precise, but it wasn’t accurate. Testing had confirmed that the mirror met the spec—verification—but it didn’t confirm that it met the original requirement—validation.

In 1993, a space shuttle mission repaired the Hubble telescope by installing a “corrective lens” to refocus the image generated by the improperly manufactured mirror.

Although this is a not a software example, verification and validation apply equally well to software testing. Never assume that the specification is correct. If you verify the spec and validate the final product, you help avoid problems such as the one that hit the Hubble telescope.

**Quality and Reliability**

Merriam-Webster’s Collegiate Dictionary defines quality as “a degree of excellence” or “superiority in kind.” If a software product is of high quality, it will meet the customer’s needs. The customer will feel that the product is excellent and superior to his other choices.

Software testers often fall into the trap of believing that quality and reliability are the same thing. They feel that if they can test a program until it’s stable, dependable, and reliable, they are assuring a high-quality product. Unfortunately, that isn’t necessarily true. Reliability is just one aspect of quality.

A software user’s idea of quality may include the breadth of features, the ability of the product to run on his old PC, the software company’s phone support availability, and, often, the price of the product. Reliability, or how often the product crashes or trashes his data, may be important, but not always.

To ensure that a program is of high quality and is reliable, a software tester must both verify and validate throughout the product development process.

**Testing and Quality Assurance (QA)**

The last pair of definitions is testing and quality assurance (sometimes shortened to QA). These two terms are the ones most often used to describe either the group or the process that’s verifying and validating the software. In Chapter 21, “Software
Quality Assurance," you'll learn more about software quality assurance, but for now, consider these definitions:

- The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.
- A software quality assurance person's main responsibility is to create and enforce standards and methods to improve the development process and to prevent bugs from ever occurring.

Of course, there is overlap. Some testers will do a few QA tasks and some QA-ers will perform a bit of testing. The two jobs and their tasks are intertwined. What's important is that you know what your primary job responsibilities are and communicate that information to the rest of the development team. Confusion among the team members about who's testing and who's not has caused lots of process pain in many projects.

Summary

Sausages, laws, and software—watching them being made can be pretty messy. Hopefully the previous three chapters haven't scared you off.

Many software testers have come into a project not knowing what was happening around them, how decisions were being made, or what procedure they should be following. It's impossible to be effective that way. With the information you've learned so far about software testing and the software development process, you'll have a head start when you begin testing for the first time. You'll know what your role should be, or at least know what questions to ask to find your place in the big picture.

For now, all the process stuff is out of the way, and the next chapter of this book begins a new section that will introduce you to the basic techniques of software testing.

Quiz

These quiz questions are provided for your further understanding. See Appendix A, “Answers to Quiz Questions,” for the answers—but don’t peek!

1. Given that it's impossible to test a program completely, what information do you think should be considered when deciding whether it's time to stop testing?

2. Start the Windows Calculator. Type $5,000-5=$ (the comma is important). Look at the result. Is this a bug? Why or why not?
3. If you were testing a simulation game such as a flight simulator or a city simulator, what do you think would be more important to test—its accuracy or its precision?

4. Is it possible to have a high-quality and low-reliability product? What might an example be?

5. Why is it impossible to test a program completely?

6. If you were testing a feature of your software on Monday and finding a new bug every hour, at what rate would you expect to find bugs on Tuesday?