In 1947, computers were big, room-sized machines operating on mechanical relays and glowing vacuum tubes. The state of the art at the time was the Mark II, a behemoth being built at Harvard University. Technicians were running the new computer through its paces when it suddenly stopped working. They scrambled to figure out why and discovered, stuck between a set of relay contacts deep in the bowels of the computer, a moth. It had apparently flown into the system, attracted by the light and heat, and was zapped by the high voltage when it landed on the relay.

The computer bug was born. Well, okay, it died, but you get the point.

Welcome to the first chapter of *Software Testing*. In this chapter, you'll learn about the history of software bugs and software testing.

Highlights of this chapter include

- How software bugs impact our lives
- What bugs are and why they occur
- Who software testers are and what they do

Infamous Software Error Case Studies

It's easy to take software for granted and not really appreciate how much it has infiltrated our daily lives. Back in 1947, the Mark II computer required legions of programmers to constantly maintain it. The average person never conceived of someday having his own computer in his home. Now there's free software CD-ROMs attached to cereal boxes and more software in our kids' video games than on the space shuttle. What once were techie gadgets, such as pagers and cell phones, have become commonplace. Most of us now can't go a day without logging on to the Internet and checking our email. We rely on overnight packages, long-distance phone service, and cutting-edge medical treatments.

Software is everywhere. However, it's written by people—so it's not perfect, as the following examples show.

Disney's *Lion King*, 1994–1995

In the fall of 1994, the Disney company released its first multimedia CD-ROM game for children, *The Lion King Animated Storybook*. Although many other companies had been marketing children's programs for years, this was Disney's first venture into the market and it was highly promoted and advertised. Sales were huge. It was "the game to buy" for children that holiday season. What happened, however, was a huge debacle. On December 26, the day after Christmas, Disney's customer support phones began to ring, and ring, and ring. Soon the phone support technicians were swamped with calls from angry parents with crying children who couldn't get the software to work. Numerous stories appeared in newspapers and on TV news.

It turns out that Disney failed to properly test the software on the many different PC models available on the market. The software worked on a few systems—likely the ones that the Disney programmers used to create the game—but not on the most common systems that the general public had.

Intel Pentium Floating-Point Division Bug, 1994

Enter the following equation into your PC's calculator:

```
(4195835 / 3145727) * 3145727 - 4195835
```

If the answer is zero, your computer is just fine. If you get anything else, you have an old Intel Pentium CPU with a floating-point division bug—a software bug burned into a computer chip and reproduced over and over in the manufacturing process.

On October 30, 1994, Dr. Thomas R. Nicely of Lynchburg (Virginia) College traced an unexpected result from one of his experiments to an incorrect answer by a division problem solved on his Pentium PC. He posted his find on the Internet and soon afterward a firestorm erupted as numerous other people duplicated his problem and found additional situations that resulted in wrong answers. Fortunately, these cases were rare and resulted in wrong answers only for extremely math-intensive, scientific, and engineering calculations. Most people would never encounter them doing their taxes or running their businesses.

What makes this story notable isn't the bug, but the way Intel handled the situation:

- Their software test engineers had found the problem while performing their own tests before the chip was released. Intel's management decided that the problem wasn't severe enough or likely enough to warrant fixing it, or even publicizing it.
- Once the bug was found, Intel attempted to diminish its perceived severity through press releases and public statements.
- When pressured, Intel offered to replace the faulty chips, but only if a user could prove that he was affected by the bug.

There was a public outcry. Internet newsgroups were jammed with irate customers demanding that Intel fix the problem. News stories painted the company as uncaring and incredulous. In the end, Intel apologized for the way it handled the bug and took a charge of over $400 million to cover the costs of replacing bad chips. Intel now reports known problems on its Web site and carefully monitors customer feedback on Internet newsgroups.

> **NOTE**
>
> On August 28th, 2000, shortly before this book went to press, Intel announced a recall of all the 1.13MHz Pentium III processors it had shipped after the chip had been in production for a month. A problem was discovered with the execution of certain instructions that could cause running applications to freeze. Computer manufacturers were creating plans for recalling the PCs already in customers' hands and calculating the costs of replacing the defective chips. As the baseball legend Yogi Berra once said, "This is like *déjà vu* all over again."

NASA Mars Polar Lander, 1999

On December 3, 1999, NASA's Mars Polar Lander disappeared during its landing attempt on the Martian surface. A Failure Review Board investigated the failure and determined that the most likely reason for the malfunction was the unexpected setting of a single data bit. Most alarming was why the problem wasn't caught by internal tests.

In theory, the plan for landing was this: As the lander fell to the surface, it was to deploy a parachute to slow its descent. A few seconds after the chute deployed, the probe's three legs were to snap open and latch into position for landing. When the probe was about 1,800 meters from the surface, it was to release the parachute and ignite its landing thrusters to gently lower it the remaining distance to the ground.

To save money, NASA simplified the mechanism for determining when to shut off the thrusters. In lieu of costly radar used on other spacecraft, they put an inexpensive contact switch on the leg's foot that set a bit in the computer commanding it to shut off the fuel. Simply, the engines would burn until the legs "touched down."

Unfortunately, the Failure Review Board discovered in their tests that in most cases when the legs snapped open for landing, a mechanical vibration also tripped the touch-down switch, setting the fatal bit. It's very probable that, thinking it had landed, the computer turned off the thrusters and the Mars Polar Lander smashed to pieces after falling 1,800 meters to the surface.

The result was catastrophic, but the reason behind it was simple. The lander was tested by multiple teams. One team tested the leg fold-down procedure and another the landing process from that point on. The first team never looked to see if the touch-down bit was set—it wasn't their area; the second team always reset the computer, clearing the bit, before it started its testing. Both pieces worked perfectly individually, but not when put together.

Patriot Missile Defense System, 1991

The U.S. Patriot missile defense system is a scaled-back version of the Strategic Defense Initiative ("Star Wars") program proposed by President Ronald Reagan. It was first put to use in the Gulf War as a defense for Iraqi Scud missiles. Although there were many news stories touting the success of the system, it did fail to defend against several missiles, including one that killed 28 U.S. soldiers in Dhahran, Saudi Arabia. Analysis found that a software bug was the problem. A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

The Y2K (Year 2000) Bug, circa 1974

Sometime in the early 1970s a computer programmer—let's suppose his name was Dave—was working on a payroll system for his company. The computer he was using had very little memory for storage, forcing him to conserve every last byte he could. Dave was proud that he could pack his programs more tightly than any of his peers. One method he used was to shorten dates from their 4-digit format, such as 1973, to a 2-digit format, such as 73. Because his payroll program relied heavily on date processing, Dave could save lots of expensive memory space. He briefly considered the problems that might occur when the current year hit 2000 and his program began doing computations on years such as 00 and 01. He knew there would be problems but decided that his program would surely be replaced or updated in 25 years and his immediate tasks were more important than planning for something that far out in time. After all, he had a deadline to meet. In 1995, Dave's program was still being used, Dave was retired, and no one knew how to get into the program to check if it was Y2K compliant, let alone how to fix it.

It's estimated that several hundred billion dollars were spent, worldwide, to replace or update computer programs such as Dave's, to fix potential Year 2000 failures.

What Is a Bug?

You've just read examples of what happens when software fails. It can be inconvenient, as when a computer game doesn't work properly, or it can be catastrophic, resulting in the loss of life. In these instances, it was obvious that the software didn't operate as intended. As a software tester you'll discover that most failures are hardly ever this obvious. Most are simple, subtle failures, with many being so small that it's not always clear which ones are true failures, and which ones aren't.

Terms for Software Failures

Depending on where you're employed as a software tester, you will use different terms to describe what happens when software fails. Here are a few:

Defect	Variance
Fault	Failure
Problem	Inconsistency
Error	Feature
Incident	Bug
Anomaly	

(There's also a list of unmentionable terms, but they're most often used privately among programmers.)

You might be amazed that so many names could be used to describe a software failure. Why so many? It's all really based on the company's culture and the process the company uses to develop its software. If you look up these words in the dictionary, you'll find that they all have slightly different meanings. They also have inferred meanings by how they're used in day-to-day conversation.

For example, *fault*, *failure*, and *defect* tend to imply a condition that's really severe, maybe even dangerous. It doesn't sound right to call an incorrectly colored icon a *fault*. These words also tend to imply blame: "It's his fault that the software failed."

Anomaly, *incident*, and *variance* don't sound quite so negative and infer more unintended operation than an all-out failure. "The president stated that it was a software anomaly that caused the missile to go off course."

Problem, *error*, and *bug* are probably the most generic terms used.

Just Call It What It Is and Get On with It

It's interesting that some companies and product teams will spend hours and hours of precious development time arguing and debating which term to use. A well-known computer company spent weeks in discussion with its engineers before deciding to rename Product Anomaly Reports (PARs) to Product Incident Reports (PIRs). Countless dollars were spent in the process of deciding which term was better. Once the decision was made, all the paperwork, software, forms, and so on had to be updated to reflect the new term. It's unknown if it made any difference to the programmer's or tester's productivity.

So, why bring this topic up? It's important as a software tester to understand the personality behind the product development team you're working with. How they refer to their software problems is a tell-tale sign of how they approach their overall development process. Are they cautious, careful, direct, or just plain blunt?

In this book, all software problems will be called *bugs*. It doesn't matter if it's big, small, intended, unintended, or someone's feelings will be hurt because they create one. There's no reason to dice words. A bug's a bug's a bug.

Software Bug: A Formal Definition

Calling any and all software problems *bugs* may sound simple enough, but doing so hasn't really addressed the issue. Now the word *problem* needs to be defined. To keep from running in circular definitions, there needs to be a definitive description of what a bug is.

First, you need a supporting term: *product specification*. A product specification, sometimes referred to as simply a *spec* or *product spec*, is an agreement among the software development team. It defines the product they are creating, detailing what it will be, how it will act, what it will do, and what it won't do. This agreement can range in form from a simple verbal understanding to a formalized written document. In Chapter 2, "The Software Development Process," you will learn more about software specifications and the development process, but for now, this definition is sufficient.

For the purposes of this book and much of the software industry, a *software bug* occurs when one or more of the following five rules is true:

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.

To better understand each rule, try the following example of applying them to a calculator.

The specification for a calculator probably states that it will perform correct addition, subtraction, multiplication, and division. If you, as the tester, receive the calculator, press the + key, and nothing happens, that's a bug because of Rule #1. If you get the wrong answer, that's also a bug because of Rule #1.

The product spec might state that the calculator should never crash, lock up, or freeze. If you pound on the keys and get the calculator to stop responding to your input, that's a bug because of Rule #2.

Suppose that you receive the calculator for testing and find that besides addition, subtraction, multiplication, and division, it also performs square roots. Nowhere was this ever specified. An ambitious programmer just threw it in because he felt it would be a great feature. This isn't a feature—it's really a bug because of Rule #3.

The fourth rule may read a bit strange with its double negatives, but its purpose is to catch things that were forgotten in the specification. You start testing the calculator and discover when the battery gets weak that you no longer receive correct answers to your calculations. No one ever considered how the calculator should react in this mode. A bad assumption was made that the batteries would always be fully charged. You expected it to keep working until the batteries were completely dead, or at least notify you in some way that they were weak. Correct calculations didn't happen with weak batteries and it wasn't specified what should happen. Rule #4 makes this a bug.

Rule #5 is the catch-all. As a tester you are the first person to really use the software. If you weren't there, it would be the customer using the product for the first time. If you find something that you don't feel is right, for whatever reason, it's a bug. In the case of the calculator, maybe you found that the buttons were too small. Maybe the placement of the = key made it hard to use. Maybe the display was difficult to read under bright lights. All of these are bugs because of Rule #5.

> **NOTE**
>
> Every person who uses a piece of software will have different expectations and opinions as to how it should work. It would be impossible to write software that every user thought was perfect. As a software tester, you should keep this in mind when you apply Rule #5 to your testing. Be thorough, use your best judgment, and be reasonable.

These are greatly simplified examples, so think about how the rules apply to software that you use every day. What is expected, what is unexpected? What do you think was specified and what was forgotten? And, what do you just plain dislike about the software?

This definition of a bug covers a lot of ground but it assures that all problems are identified.

Why Do Bugs Occur?

Now that you know what bugs are, you might be wondering why they occur. What you'll be surprised to find out is that most of them aren't caused by programming errors. Numerous studies have been performed on very small to extremely large projects and the results are always the same. The number one cause of software bugs is the specification (see Figure 1.1).
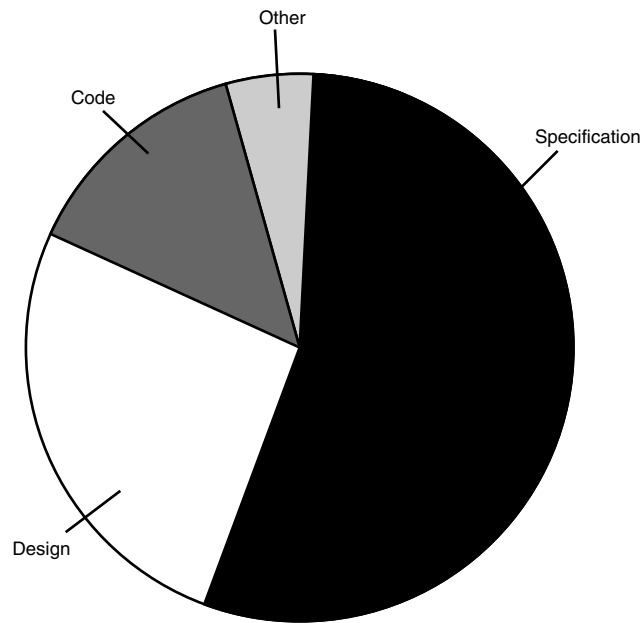
FIGURE 1.1
Bugs are caused for numerous reasons, but the main cause can be traced to the specification.

There are several reasons specifications are the largest bug producer. In many instances a spec simply isn't written. Other reasons may be that the spec isn't thorough enough, it's constantly changing, or it's not communicated well to the entire development team. Planning software is vitally important. If it's not done correctly, bugs will be created.

The next largest source of bugs is the design. This is where the programmers lay out their plan for the software. Compare it to an architect creating the blueprints for a building. Bugs occur here for the same reason they occur in the specification. It's rushed, changed, or not well communicated.

NOTE

There's an old saying, "If you can't say it, you can't do it." This applies perfectly to software development and testing.

Coding errors may be more familiar to you if you're a programmer. Typically, these can be traced to the software's complexity, poor documentation (especially in code that's being

updated or revised), schedule pressure, or just plain dumb mistakes. It's important to note that many bugs that appear on the surface to be programming errors can really be traced to specification and design errors. It's quite common to hear a programmer say, "Oh, so that's what it's supposed to do. If somebody had just told me that I wouldn't have written the code that way."

The other category is the catch-all for what's left. Some bugs can be blamed on false positives, conditions that were thought to be bugs but really weren't. There may be duplicate bugs, multiple ones that resulted from the same root cause. Some bugs can also be traced to testing errors. In the end, these bugs usually make up such a small percentage of all bugs found that they aren't worth worrying about.

The Cost of Bugs

As you will learn in Chapter 2, software doesn't just magically appear—there's usually a planned, methodical development process used to create it. From its inception, through the planning, programming, and testing, to its use by the public, there's the potential for bugs to be found. Figure 1.2 shows how the cost of fixing these bugs grows over time.
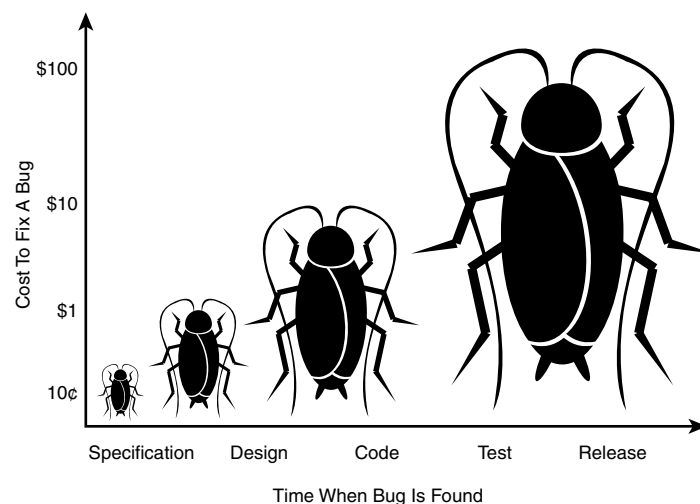
FIGURE 1.2
The cost to fix bugs increases dramatically over time.

The costs are logarithmic—that is, they increase tenfold as time increases. A bug found and fixed during the early stages when the specification is being written might cost next to nothing, or 10 cents in our example. The same bug, if not found until the software is coded and tested, might cost $1 to $10. If a customer finds it, the cost could easily top $100.

As an example of how this works, consider the Disney *Lion King* case discussed earlier. The root cause of the problem was that the software wouldn't work on a very popular PC platform. If, in the early specification stage, someone had researched what PCs were popular and specified that the software needed to be designed and tested to work on those configurations, the cost of that effort would have been almost nothing. If that didn't occur, a backup would have been for the software testers to collect samples of the popular PCs and verify the software on them. They would have found the bug, but it would have been more expensive to fix because the software would have to be debugged, fixed, and retested. The development team could have also sent out a preliminary version of the software to a small group of customers in what's called a *beta test*. Those customers, chosen to represent the larger market, would have likely discovered the problem. As it turned out, however, the bug was completely missed until many thousands of CD-ROMs were created and purchased. Disney ended up paying for telephone customer support, product returns, replacement CD-ROMs, as well as another debug, fix, and test cycle. It's very easy to burn up your entire product's profit if serious bugs make it to the customer.

What Exactly Does a Software Tester Do?

You've now seen examples of really nasty bugs, you know what the definition of a bug is, and you know how costly they can be. By now it should be pretty evident what a tester's goal is:

> *The goal of a software tester is to find bugs.*

You may run across product teams who want their testers to simply confirm that the software works, not to find bugs. Reread the case study about the Mars Polar Lander, and you'll see why this is the wrong approach. If you're only testing things that should work and setting up your tests so they'll pass, you will miss the things that don't work. You will miss the bugs.

If you're missing bugs, you're costing your project and your company money. As a software tester you shouldn't be content at just finding bugs—you should think about how to find them sooner in the development process, thus making them cheaper to fix.

> *The goal of a software tester is to find bugs, and find them as early as possible.*

But, finding bugs, even finding them early, isn't enough. Remember the definition of a bug. You, the software tester, are the customer's eyes, the first one to see the software. You speak for the customer and must demand perfection.

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

This final definition is very important. Commit it to memory and refer back to it as you learn the testing techniques discussed throughout the rest of this book.

What Makes a Good Software Tester?

In the movie *Star Trek II: The Wrath of Khan*, Spock says, "As a matter of cosmic history, it has always been easier to destroy than to create." At first glance, it may appear that a software tester's job would be easier than a programmer's. Breaking code and finding bugs must surely be easier than writing the code in the first place. Surprisingly, it's not. The methodical and disciplined approach to software testing that you'll learn in this book requires the same hard work and dedication that programming does. It involves very similar skills, and although a software tester doesn't necessarily need to be a full-fledged programmer, having that knowledge is a great benefit.

Today, most mature companies treat software testing as a technical engineering profession. They recognize that having trained software testers on their project teams and allowing them to apply their trade early in the development process allows them to build better quality software.

Here's a list of traits that most software testers should have:

- **They are explorers.** Software testers aren't afraid to venture into unknown situations. They love to get a new piece of software, install it on their PC, and see what happens.
- **They are troubleshooters.** Software testers are good at figuring out why something doesn't work. They love puzzles.
- **They are relentless.** Software testers keep trying. They may see a bug that quickly vanishes or is difficult to re-create. Rather than dismiss it as a fluke, they will try every way possible to find it.
- **They are creative.** Testing the obvious isn't sufficient for software testers. Their job is to think up creative and even off-the-wall approaches to find bugs.
- **They are (mellowed) perfectionists.** They strive for perfection, but they know when it becomes unattainable and they're OK with getting as close as they can.
- **They exercise good judgment.** Software testers need to make decisions about what they will test, how long it will take, and if the problem they're looking at is really a bug.
- **They are tactful and diplomatic.** Software testers are always the bearers of bad news. They have to tell the programmers that their baby is ugly. Good software testers know how to do so tactfully and professionally and know how to work with programmers who aren't always tactful and diplomatic.
- **They are persuasive.** Bugs that testers find won't always be viewed as severe enough to be fixed. Testers need to be good at making their points clear, demonstrating why the bug does indeed need to be fixed, and following through on making it happen.

> ### Software Testing Is Fun!
> A fundamental trait of software testers is that they simply like to break things. They live to find those elusive system crashes. They take great satisfaction in laying to waste the most complex programs. They're often seen jumping up and down in glee, giving each other high-fives, and doing a little dance when they bring a system to its knees. It's the simple joys of life that matter the most.

In addition to these traits, having some education in software programming is a big plus. As you'll see in Chapter 6, "Examining the Code," knowing how software is written can give you a different view of where bugs are found, thus making you a more efficient and effective tester. It can also help you develop the testing tools discussed in Chapter 14, "Automated Testing and Test Tools."

Lastly, if you're an expert in some non-computer field, your knowledge can be invaluable to a software team creating a new product. Software is being written to do just about everything today. Your knowledge of teaching, cooking, airplanes, carpentry, medicine, or whatever would be a tremendous help finding bugs in software for those areas.

Summary

Software testing is a critical job. With the size and complexity of today's software, it's imperative that software testing be performed professionally and effectively. Too much is at risk. We don't need more defective computer chips or lost Mars landers.

In the following chapters of Part I, you'll learn more about the big picture of software development and how software testing fits in. This knowledge is critical to helping you apply the specific test techniques covered in the remainder of this book.

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. In the Year 2000 bug example, did Dave do anything wrong?

2. **True or False:** It's important what term your company or team calls a problem in its software.

3. What's wrong with just testing that a program works as expected?

4. How much more does it cost to fix a bug found after the product is released than it does from the very start of the project?

5. What's the goal of a software tester?

6. **True or False:** A good tester relentlessly strives for perfection.

7. Give several reasons why the product specification is usually the largest source of bugs in a software product.

You're now down to the final chapter of software testing. Well, okay, maybe the final chapter of the book *Software Testing*, but definitely not of the job. Your work in that area has only just begun.

You probably began reading this book with little knowledge of what software testing is all about. You've likely experienced the minor little annoyances and the occasional crashes of the software you use on your computer at home or at work. You've seen and heard news stories of major software bugs and you know about the infamous Y2K bug, which, after all the last-minute testing and preparation, didn't bite as hard as expected.

Hopefully you've now been enlightened and understand why these bugs can still happen despite the best efforts of the people behind the software. You've learned about the test planning process, where to look for bugs, and how to report them. You now understand the difficult decision-making process that's involved in determining which bugs to fix and which ones to defer, and you've seen the graphs that show a product that's ready to release and one that still has a long way to go.

Above all else, you should now understand that software testing is a complex and difficult job. To be successful at it requires discipline, training, and experience. Simply sitting down, pounding the keys, and shouting over the wall to the programmer when you see something odd won't cut it. Software is too important. Businesses have failed, careers have been ruined, and people have died because of software bugs. Your job as a software tester is to find those bugs, efficiently and professionally, before they make it out the door.

This final chapter will give you pointers to more information about software testing, explain a few of the possible career options, and leave you with an important message about software quality. Highlights of this chapter include

- The career path options available for software testers
- Where to look for a testing job
- How to get more hands-on experience at finding bugs
- Where to learn more about software testing
- The Computer User's Bill of Rights

Your Job as a Software Tester

One serious misconception about software testing is that it's only an entry-level position in the software industry. This erroneous belief persists because of the ignorance of what software testing is and what it involves—mainly due to the number of companies still developing software without any real process. They don't yet know that they need software testers of all skill levels to create great software. But, as more emphasis is put on creating software of higher and higher quality, the value of software testing as a career is becoming understood.

Because of this increased awareness, the opportunities are there for the taking. Software testers with just a couple years of experience are highly sought after. Testers who can also program and perform white-box testing or develop automated tests are even more in demand. And, if you've been through a few product development cycles and can lead a small team of other testers, you're in a highly marketable position. It's truly a job-hunter's market for software testers.

Here's a breakout of various software testing positions and their descriptions. Keep in mind, as you learned in Chapter 20, "Software Quality Assurance," the names vary and may not mean exactly what the job really is, but ultimately most software testing jobs fall into these categories.

- **Software test technician.** This is often a true entry-level test position. You would be responsible for setting up test hardware and software configurations, running simple test scripts or test automation, and possibly working with beta sites to isolate and reproduce bugs. Some work can become mundane and repetitive, but being a test technician is a good way to become introduced to software testing.

- **Software tester or software test engineer.** Most companies have several levels of software testers based on experience and expertise. An entry-level tester may perform the duties of technician, working their way up to running more advanced and complex tests. As you progress, you'll write your own test cases and test procedures and might attend design and specification reviews. You'll perform testing and isolate, reproduce, and report the bugs you find. If you have programming abilities, you'll write test automation or testing tools and work closely with the programmers as you perform white-box testing.

- **Software test lead.** A test lead is responsible for the testing of a major portion of a software project or sometimes an entire small project. They often generate the test plan for their areas and oversee the testing performed by other testers. They're frequently involved in collecting metrics for their products and presenting them to management. They usually also perform the duties of a software tester.

- **Software test manager.** A test manager oversees the testing for an entire project or multiple projects. The test leads report to them. They work with the project managers and development managers to set schedules, priorities, and goals. They're responsible for providing the appropriate testing resources—people, equipment, space, and so on—for their projects. They set the tone and strategy for the testing their teams perform.

Finding a Software Testing Position

So where do you look for a software testing job? The answer is the same places you would look for a programming job—with any business or company that develops software.

- **Use the Internet.** A quick search done using several job search engines just before this book went to print found more than 1,000 open software testing positions at companies all around the country. Many of these positions were for entry-level testers. There were

jobs testing music software, interactive TV, networking, medical equipment, Web sites, and more. You name it.

- **Look through newspapers and magazines.** As with the Internet, most large city newspapers list numerous software testing jobs every weekend in their high-tech or computer help wanted ads. Computing and programming magazines are also a good source for job ads.

- **Simply call and ask.** Do you like a certain technology or a specific software application, or are you interested in a particular area of computing? Look up the company or companies writing that software, give them a call or send them your résumé. Often there are open software testing jobs that never make it to the help wanted ads. Resourceful testers grab them before anyone knows they're available.

- **Look into internships and co-ops.** If you're in college, take a summer or a semester as an intern or co-op at a software company as a software tester. Most provide great positions where you'll actually be contributing to the success of a real product as you gain experience in the field. If you work hard, you might be offered a full-time position after you graduate.

- **Take temporary jobs.** Many software companies hire software testers under temporary contracts to help run test scripts as a product nears completion. The jobs may last just one month or several months, but you'll gain valuable experience and can take on more difficult jobs each time, learning as you go. Some software testers love to work this way because it gives them the opportunity to work at different companies and try their hand testing very different software.

Gaining Hands-On Experience

Software testing is just like most other computer topics—you can read about it all day, but until you actually try the things you've read about, it's difficult to comprehend. For this reason, one of the best ways to learn software testing is to try it yourself, on your own computer with your own software.

Choose a program you're familiar with and enjoy using or one that you've never used before. Read the manual and the help files as though they were the product specification. Put together a test plan, design the test cases, and look for bugs. Use a spreadsheet or word processor to log them and report what you find to the software company that wrote the application (almost all software companies have a means for reporting problems, usually through their Web site). You'll be surprised what you find, and maybe the company will, too.

With a little experience at this type of testing, you could sign up to be a *beta tester* for new software products. As you learned in Chapter 15, "Bug Bashes and Beta Testing," beta testers receive copies of the software before the general public does. You'll have an opportunity to see

and use software that's not quite finished, find bugs that were missed by the company's internal software testers, and, based on what you find, possibly have an impact on the product's design. Every software company has its own system for administering beta tests. Search their Web site for "beta tester" or call them and ask to talk with someone about becoming a beta tester.

> **NOTE**
>
> Be careful if you use beta software on your home or work PC. By its nature, beta software isn't ready for public release and is full of bugs. Some of those bugs can cause serious problems with your computer and existing software, including frequent crashes and loss of data. Make backups of anything important before you run beta software.

Becoming a usability test subject (see Chapter 11, "Usability Testing") is another way to gain some hands-on experience at software testing. Most large software companies making personal computer software have usability labs or contract out to independent usability labs in their area. If you're interested in testing user interfaces, make some phone calls and inquire about becoming a subject for software usability testing. Often you'll be asked to fill out a form to measure your experience and familiarity with certain types of software. As projects go to usability testing, your profile will be reviewed to see if you're the type of person they're looking for; depending on the product being tested, they may need absolute beginners all the way to experts. If you're a match, you'll be called in to try out features in a new product or even a prototype of a new product. The people administering the test will watch what you do, record your actions, and observe how you react to the software. They may invite you back to try out changes they make based on your findings and you'll often be compensated for your time— usually with free software.

Formal Training Opportunities

With the realization that software testing is an important field of study, many colleges and universities have begun to offer classes on the subject. If you're currently in an engineering or computer degree program, it would be well worth your time to enroll in one of these classes. Even if your plans are to become a programmer or engineer, gaining a better knowledge of software testing can help you perform your job even better.

Many community and technical colleges are now offering day and evening classes on software testing and the use of popular software testing tools. Some even award associate degrees and certificates in software testing.

Another option for training is to attend professional software testing conferences. Held throughout the year and in various parts of the U.S. and even the world, these conferences provide an opportunity for you to hear speakers from across the testing industry. Class material spans the entire range from very basic to very advanced and technical. The best part of these conferences is having the opportunity to meet and talk with fellow software testers, sharing ideas, war stories, and solutions. The following list represents some of the more popular conferences, but definitely not all of them. Presence or absence doesn't reflect an endorsement or an opinion.

- **International Conference and Exposition on Testing Computer Software (TCS),** sponsored by the U.S. Professional Development Institute (`www.uspdi.org`). TCS features international speakers, presentations of current software testing best practices by experienced professionals, opportunities to share lessons and experiences with your peers, and a vendor show of products and services.

- **International Quality Week,** sponsored by Software Research, Inc. (`www.soft.com`). The mission of the Quality Week and Quality Week Europe Conferences is to "bring advanced technology R&D work relating to software quality, testing and process, contemporary software quality and testing needs, and knowledge and experience from the software quality industry together in a balanced technical forum."

- **International Software Testing Conference (ISTC),** sponsored by the Quality Assurance Institute (`www.qaiusa.com`). ISTC is a weeklong conference featuring expert speakers from across the software testing and quality assurance industry. Topics range from basic software testing to test automation to testing specific new technologies.

- **Software Testing Analysis & Review (STAR),** sponsored by Software Quality Engineering (`www.sqe.com/stareast` and `www.sqe.com/starwest`). The STAR conferences are focused exclusively on software testing and software engineering. They provide classes, tutorials, and discussions by software testing experts and hold an exposition where test tool, technology, and service companies demonstrate their products.

- **International Conference on Software Quality (ICSQ),** sponsored by the Software Division of the American Society for Quality (`www.asq-software.org`). ICSQ, like the other conferences, provides an opportunity to share testing ideas and methods with other software testing and quality assurance professionals.

- **International Conference on Software Testing (ICSTEST),** sponsored by Software Quality Systems (`www.icstest.com`). ICSTEST is an international testing conference held in Germany. It's a forum for presentations, tutorials, discussions, and exchange of experiences on software testing.

- **The Second World Congress for Software Quality (2WCSQ)** (`www.juse.or.jp/e-renmei/2WCSQMAIN.htm`). 2WCSQ is a worldwide conference on software quality with members from more than 27 countries. The year 2000 conference was held in Japan.

Internet Links

The Internet has a wealth of material on software testing. You could always do a search for "software testing" or "software test," but here's a list of popular Web sites dedicated to software testing and software bugs that will get you started:

- BugNet (`www.bugnet.com`) publicizes bugs found in commercial software and points you to the appropriate fixes.

- Software Testing Hotlist (`www.io.com/~wazmo/qa`) lists dozens of pointers to software testing–related Web sites and articles.

- Software Testing Online Resources (`www.mtsu.edu/~storm`) is the self-proclaimed "nexus of Software Testing Online Resources…designed to be a 'first-stop' on the Web for software testing researchers and practitioners."

- QA Forums (`www.qaforums.com`) provide ongoing discussions of software testing, automated testing, test management, test tools, and many other topics.

- The newsgroup `comp.software.testing` and its FAQ (frequently asked questions) document at `www.faqs.org/faqs/software-eng/testing-faq` provide lots of ongoing discussions by testers and test managers regarding tools, techniques, and projects.

- The newsgroup `comp.risks` describes and analyzes recent software failures.

- The Risks Digest (`catless.ncl.ac.uk/Risks/`) is a forum on risks to the public in computers and related systems.

Professional Organizations

Several professional nonprofit organizations are dedicated to software, software testing, and software quality assurance that may be of interest to you. Their Web sites provide details on their specific area of coverage:

- The American Society for Quality (ASQ) at `www.asq.org` and its software division at `www.asq-software.org` sponsor the National Quality Forum annually in October (national quality month). They publish journals and articles on quality and administer the Certified Quality Engineer (CQE) and the Certified Software Quality Engineer (CSQE) designation.

- The Association of Computing Machinery (ACM) at `www.acm.org` and its Special Interest Group on Software Engineering (SIGSOFT) at `www.acm.org/sigsoft` has more than 80,000 members in educational and scientific computing. The software engineering

interest group publishes a bimonthly newsletter with a popular forum titled "Risks to the Public," which details recent serious software bugs.

- The Society for Software Quality (SSQ) at `www.ssq.org` lists its vision as "To be recognized as the Society for those interested in promoting 'quality' as a universal goal for software."

Further Reading

Numerous books are available on the topics of software testing and software quality assurance. Each has its own audience and approach. Some are very technical, whereas others are very process and managerial oriented. Your best bet to find something that interests you is to visit a large bookstore, university library, or favorite online bookstore and look for titles by Boris Beizer, Rex Black, Bill Hetzel, Cem Kaner, Edward Kit, Glen Myers, and William Perry.

If you're interested in learning more about general product quality assurance techniques, look for titles by Philip Crosby, W. Edwards Deming, and Joseph Juran.

The following Sams Publishing titles are a sampling of some books that you might want to look at to improve your understanding of computers and programming. These books were specifically chosen to get you up to speed quickly and easily and are very appropriate reading for the novice software tester:

- *Sams Teach Yourself Beginning Programming in 24 Hours* is a great introduction to the basics of programming. You won't become a white-box tester just by reading this book, but you will gain a better insight into how software is written—which will help you design better test cases.

- *Sams Teach Yourself HTML in 24 Hours*, *Sams Teach Yourself Visual Basic in 24 Hours*, *Sams Teach Yourself Java in 24 Hours*, and *Sams Teach Yourself C++ in 24 Hours* are a good next step once you've mastered the basics of programming.

- If your goal is to be a serious white-box tester, *Sams Teach Yourself Visual Basic in 21 Days*, *Sams Teach Yourself Java in 21 Days*, *Sams Teach Yourself C in 21 Days*, and *Sams Teach Yourself C++ in 21 Days* will teach you the details of programming in specific languages.

- *Sams Teach Yourself Upgrading and Fixing PCs in 24 Hours* will teach you the basics of adding new hardware and peripherals to your PC—a very important topic for a software tester, especially if you're interested in configuration testing.

- *Internationalization with Visual Basic*

NOTE

To keep up with the latest in programming titles, regularly check out Macmillan Computer Publishing's Web site at `www.mcp.com`.

Summary

It seems appropriate to close out this book with a mantra that sums up what you should hope to achieve through your work as a software tester. Frequently throughout this book, qualifiers such as "depending on your company or your project team" and "based on your industry" were used when describing development processes, testing techniques, and quality levels. The use of such qualifiers makes it impossible to universally define a common goal for software quality. The qualifiers are necessary, though, because unfortunately, at least so far, the definition of software quality "depends."

In 1998, Dr. Clare-Marie Karat, a psychologist and user interface designer at IBM's Thomas J. Watson Research Center in Hawthorne, NY, proposed a computer user's bill of rights. This bill of rights sets a minimum quality bar, a minimum set of expectations that computer users should have the rights to with the software they use. The computer industry has a long way to go to achieve this level of quality, but with your work as a software tester, you can help to make it a reality.

The Computer User's Bill of Rights (reprinted with Dr. Karat's permission)[1]:

1. **Perspective.** The user is always right. If there's a problem with the use of the system, the system is the problem, not the user.
2. **Installation.** The user has the right to easily install and uninstall software and hardware systems without negative consequences.
3. **Compliance.** The user has the right to a system that performs exactly as promised.
4. **Instruction.** The user has the right to easy-to-use instructions (user guides, online or contextual help, error messages) for understanding and utilizing a system to achieve desired goals and recover efficiently and gracefully from problem situations.
5. **Control.** The user has the right to be in control of the system and to be able to get the system to respond to a request for attention.
6. **Feedback.** The user has the right to a system that provides clear, understandable, and accurate information regarding the task it's performing and the progress toward completion.

---

[1]*Published with permission from IBM Corporation.*

7. **Dependencies.** The user has the right to be clearly informed about all systems requirements for successfully using software or hardware.

8. **Scope.** The user has the right to know the limits of the system's capabilities.

9. **Assistance.** The user has the right to communicate with the technology provider and receive a thoughtful and helpful response when raising concerns.

10. **Usability.** The user should be the master of software and hardware technology, not vice versa. Products should be natural and intuitive to use.

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. When looking for a software testing position on the Internet, what keywords should you use in your search?

2. Name two ways that you can become involved in testing computer software before it is released to the public.

3. What's the goal of a software tester?