This document contains Chapters 1 – 14 of the book.

CONTENTS

CHAPTER 1. SOFTWARE TESTING BACKGROUND

IN THIS CHAPTER

- **Infamous Software Error Case Studies**
- **What Is a Bug?**
- **Why Do Bugs Occur?**
- **The Cost of Bugs**
- **What Exactly Does a Software Tester Do?**
- **What Makes a Good Software Tester?**

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 test the software on a broad representation of 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 more than $400 million to cover the costs of replacing bad chips. Intel now reports known problems on its website and carefully monitors customer feedback on Internet newsgroups.

NOTE

On August 28th, 2000, shortly before the first edition of this book went to press, Intel announced a recall of all the 1.13GHz 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.

DANGEROUS VIEWING AHEAD, 2004

On April 1, 1994, a message was posted to several Internet user groups and then quickly circulated as an email that a virus was discovered embedded in several JPEG format pictures available on the Internet. The warning stated that simply opening and viewing the infected pictures would install the virus on your PC. Variations of the warning stated that the virus could damage your monitor and that Sony Trinitron monitors were "particularly susceptible."

Many heeded the warning, purging their systems of JPEG files. Some system administrators even went so far as to block JPEG images from being received via email on the systems.

Eventually people realized that the original message was sent on "April Fools Day" and that the whole event was nothing but a joke taken too far. Experts chimed in that there was no possible way viewing a JPEG image could infect your PC with a virus. After all, a picture is just data, it's not executable program code.

Ten years later, in the fall of 2004, a proof-of-concept virus was created, proving that a JPEG picture could be loaded with a virus that would infect the system used to view it. Software patches were quickly made and updates distributed to prevent such a virus from spreading. However, it may only be a matter of time until a means of transmission, such as an innocuous picture, succeeds in wreaking havoc over the Internet.

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. It can cost only pennies to fix but millions of dollars to distribute a solution. In the examples, above, 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 are often used to infer unintended operation rather than 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?
Although your team may choose a different name, 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, an email, or a scribble on a napkin, to a highly detailed, 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.

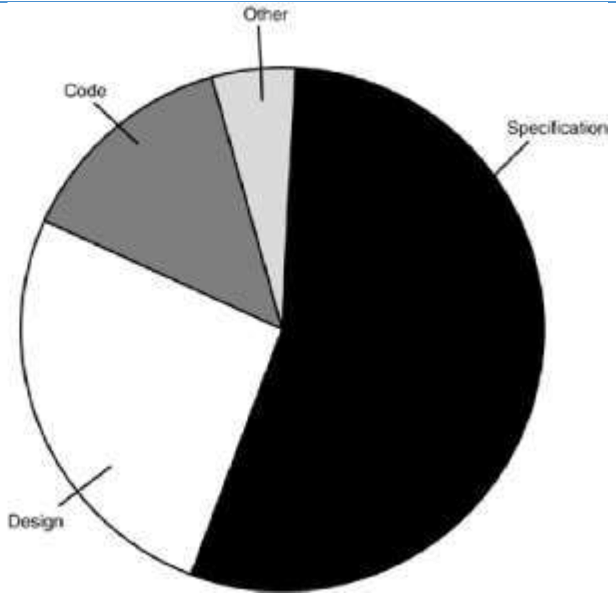
To better understand each rule, try the following example of applying them to a calculator.
The specification for a calculator probably states that it will perform correct addition, subtraction, multiplication, and division. If you, as the tester, receive the calculator, press the + key, and nothing 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 software is doing something that the product specification didn't mention. This unintended operation, although maybe nice to have, will add to the test effort and will likely introduce even more bugs.
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, most importantly, be reasonable. Your opinion counts, but, as you'll learn in later chapters, not all the bugs you find can or will be fixed.

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 using all five of its rules will help you identify the different types of problems in the software you're testing.

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, IN THIS SAMPLE PROJECT ANALYSIS, 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

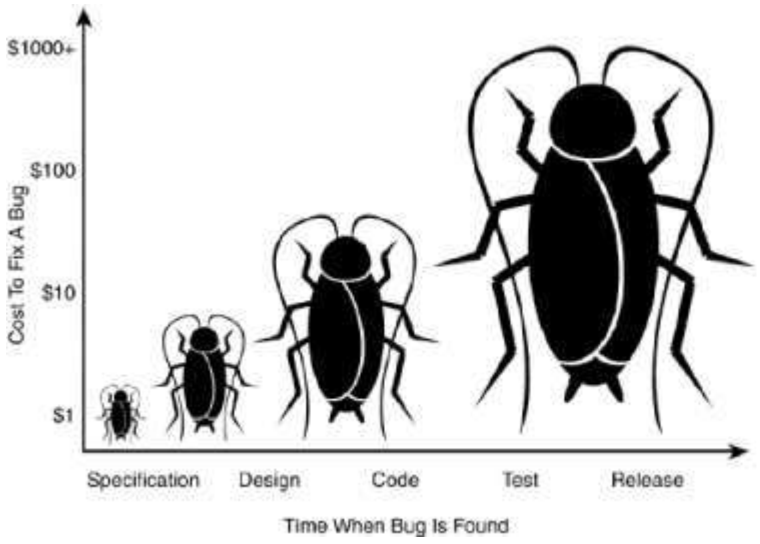
There's an old saying, "If you can't say it, you can't do it." This applies perfectly to software development and testing.

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 (or what once were thought of as bugs) turn out to not be bugs at all and make up a very small percentage of all the bugs reported.

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 an example of how the cost of fixing these bugs can grow 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 $1 in our example. The same bug, if not found until the software is coded and tested, might cost $10 to $100. If a customer finds it, the cost could easily be thousands or even millions of dollars.

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 minimal. 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 seek 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.

NOTE
It's important to note that "fixing" a bug does not necessarily imply correcting the software. It could mean adding a comment in the user manual or providing special training to the customers. It could require changing the statistics that the marketing group advertises or even postponing the release of the buggy feature. You'll learn throughout this book that although you're seeking perfection and making sure that the bugs get fixed, that there are practical realities to software testing. Don't get caught in the dangerous spiral of unattainable perfection.

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. Unfortunately, there are still a few companies that don't appreciate the challenge of software testing and the value of well-done testing effort. In a free market society, these companies usually aren't around for long because the customers speak with their wallets and choose not to buy their buggy products. A good test organization (or the lack of one) can make or break a company.
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 okay 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 15, "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, crashed systems, or stolen credit card numbers.

In the following chapters of Part I, "**The Big Picture**," 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.

CHAPTER 2. THE SOFTWARE DEVELOPMENT PROCESS

IN THIS CHAPTER

- **Product Components**
- **Software Project Staff**
- **Software Development Lifecycle Models**

To be an effective software tester, it's important to have at least a high-level understanding of the overall process used to develop software. If you write small programs as a student or hobbyist, you'll find that the methods you use are much different from what big companies use to develop software. The creation of a new software product may involve dozens, hundreds, even thousands of team members all playing different roles and working together under tight schedules. The specifics of what these people do, how they interact, and how they make decisions are all part of the software development process.

The goal of this chapter isn't to teach you everything about the software development process that would take an entire book! The goal is to give you an overview of the all the pieces that go into a software product and a look at a few of the common approaches in use today. With this knowledge you'll have a better understanding of how best to apply the software testing skills you learn in the later chapters of this book.

The highlights of this chapter include

- What major components go into a software product
- What different people and skills contribute to a software product

- How software progresses from an idea to a final product

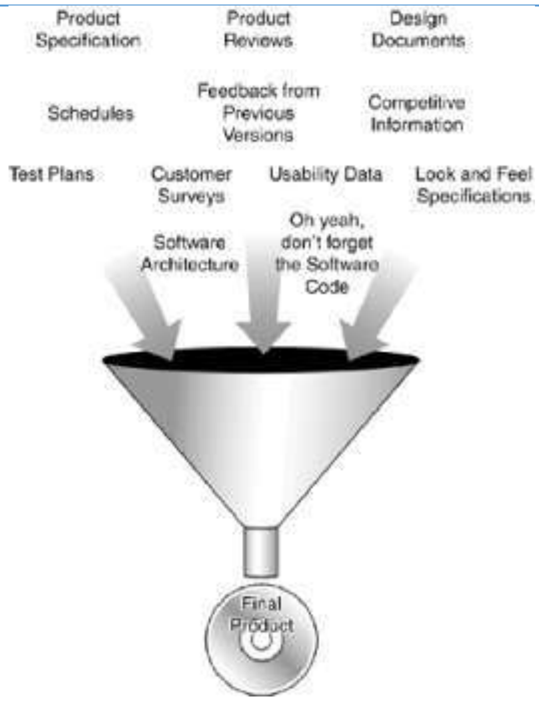
PRODUCT COMPONENTS

What exactly is a software product? Many of us think of it as simply a program that we download from the Internet or install from a DVD that runs on our computer. That's a pretty good description, but in reality, many hidden pieces go into making that software. There are also many pieces that "come in the box" that are often taken for granted or might even be ignored. Although it may be easy to forget about all those parts, as a software tester, you need to be aware of them, because they're all testable pieces and can all have bugs.

WHAT EFFORT GOES INTO A SOFTWARE PRODUCT?

First, look at what effort goes into a software product. Figure 2.1 identifies a few of the abstract pieces that you may not have considered.

FIGURE 2.1. A LOT OF HIDDEN EFFORT GOES INTO A SOFTWARE PRODUCT.



So what are all these things, besides the actual code, that get funneled into the software? At first glance they probably seem much less tangible than the program listing a programmer creates. And they definitely aren't something that can be viewed directly from the product's CD-ROM. But, to paraphrase a line from an old spaghetti sauce commercial, "they're in there." At least, they should be.
The term used in the software industry to describe a software product component that's created and passed on to someone else is deliverable. The easiest way to explain what all these deliverables are is to organize them into major categories.

CUSTOMER REQUIREMENTS

Software is written to fulfill some need that a person or a group of people has. Let's call them the customer. To properly fill that need, the product development team must find out what the customer wants. Some teams simply guess, but most collect detailed information in the form of surveys, feedback from previous versions of the software, competitive product information, magazine reviews, focus groups, and numerous other methods, some formal, some not. All this information is then studied, condensed, and interpreted to decide exactly what features the software product should have.

PUT YOUR FEATURES IN PERSPECTIVE WITH FOCUS GROUPS

A popular means to get direct feedback from potential customers of a software product is to use focus groups. Focus groups are often organized by independent survey companies who set up offices in shopping malls. The surveyors typically walk around the mall with a clipboard and ask passers-by if they want to take part in a study. They'll ask a few questions to qualify you such as "Do you have a PC at home? Do you use software X? How much time do you spend online?" And so on. If you fit their demographic, they'll invite you to return for a few hours to participate with several other people in a focus group. There, you'll be asked more detailed questions about computer software. You may be shown various software boxes and be asked to choose your favorite. Or, you may discuss as a group features you'd like to see in a new product. Best of all, you get paid for your time.
Most focus groups are conducted in such a way that the software company requesting the information is kept anonymous. But, it's usually easy to figure out who they are.

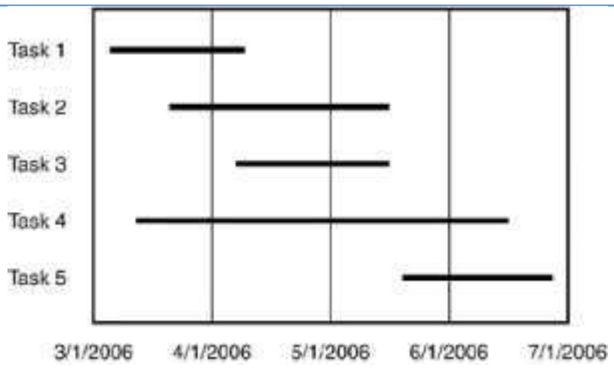
SPECIFICATIONS

The result of the customer requirements studies is really just raw data. It doesn't describe the proposed product, it just confirms whether it should (or shouldn't) be created and what features the customers want. The specifications take all this information plus any unstated but mandatory requirements and truly define what the product will be, what it will do, and how it will look.
The format of specifications varies greatly. Some companies specially those developing products for the government, aerospace, financial and medical industries use a very rigorous process with many checks and balances. The result is an extremely detailed and thorough specification that's locked down, meaning that it can't change except under very extreme conditions. Everyone on the development team knows exactly what they are creating.
There are development teams, usually ones creating software for less-critical applications, who produce specifications on cocktail napkins, if they create them at all. This has the distinct advantage of being very flexible, but there's lots of risk that not everyone is "on the same page." And, what the product finally becomes isn't known until it's released.

SCHEDULES

A key part of a software product is its schedule. As a project grows in size and complexity, with many pieces and many people contributing to the product, it becomes necessary to have some mechanism to track its progress. This could range from simple task lists to Gantt charts (see Figure 2.2) to detailed tracking of every minute task with project management software.

FIGURE 2.2. A GANTT CHART IS A BAR CHART THAT SHOWS A PROJECT'S TASKS AGAINST A HORIZONTAL TIMELINE.



The goals of scheduling are to know which work has been completed, how much work is still left to do, and when it will all be finished.

SOFTWARE DESIGN DOCUMENTS

One common misconception is that when a programmer creates a program, he simply sits down and starts writing code. That may happen in some small, informal software shops, but for anything other than the smallest programs, there must be a design process to plan out how the software will be written. Think about this book, which required an outline before the first words were typed, or a building, which has blueprints drawn before the first concrete is poured. The same planning should happen with software.

The documents that programmers create vary greatly depending on the company, the project, and the team, but their purpose is to plan and organize the code that is to be written.

Here is a list of a few common software design documents:

- Architecture. A document that describes the overall design of the software, including descriptions of all the major pieces and how they interact with each other.
- Data Flow Diagram. A formalized diagram that shows how data moves through a program. It's sometimes referred to as a bubble chart because it's drawn by using circles and lines.
- State Transition Diagram. Another formalized diagram that breaks the software into basic states, or conditions, and shows the means for moving from one state to the next.
- Flowchart. The traditional means for pictorially describing a program's logic. Flowcharting isn't very popular today, but when it's used, writing the program code from a detailed flowchart is a very simple process.
- Commented Code. There's an old saying that you may write code once, but it will be read by someone at least 10 times. Properly embedding useful comments in the software code itself is extremely important, so that programmers assigned to maintain the code can more easily figure out what it does and how.

TEST DOCUMENTS

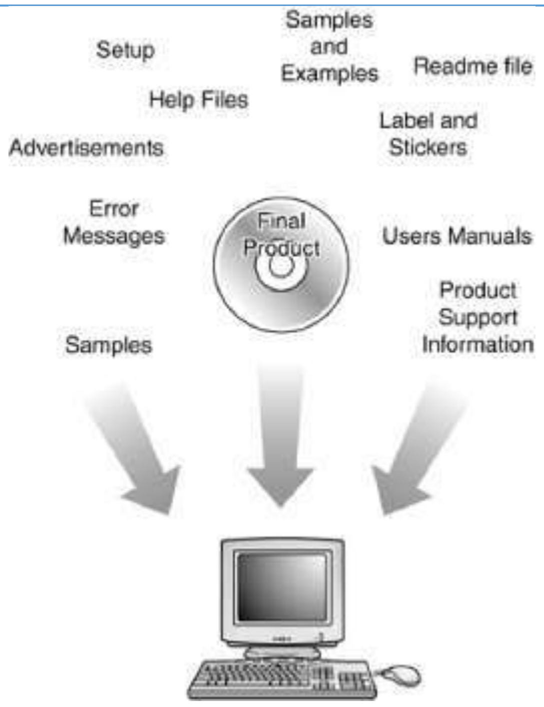
Test documentation is discussed in detail in Chapters 1720 but is mentioned here because it's integral to what makes up a software product. For the same reasons that programmers must plan and document their work, software testers must as well. It's not unheard of for a software test team to create more deliverables than the programmers.
Here's a list of the more important test deliverables:

- The test plan **describes the overall method** to be used to verify that the software meets the product specification and the customer's needs. It includes the quality objectives, resource needs, schedules, assignments, methods, and so forth.
- Test cases list the specific items that will be tested and describe the detailed steps that will be followed to verify the software.
- Bug reports describe the problems found as the test cases are followed. These could be done on paper but are often tracked in a database.
- Test tools and automation are described in detail in Chapter 15, "Automated Testing and Test Tools." If your team is using automated methods to test your software, the tools you use, either purchased or written in-house, must be documented.
- Metrics, statistics, and summaries convey the progress being made as the test work progresses. They take the form of graphs, charts, and written reports.

WHAT PARTS MAKE UP A SOFTWARE PRODUCT?

So far in this chapter you've learned about the effort that goes into creating a software product. It's also important to realize that when the product is ready to be boxed up and shipped out the door, it's not just the code that gets delivered. Numerous supporting parts go along with it (see Figure 2.3). Since all these parts are seen or used by the customer, they need to be tested too.

FIGURE 2.3. THE SOFTWARE CD-ROM IS JUST ONE OF THE MANY PIECES THAT MAKE UP A SOFTWARE PRODUCT.



It's unfortunate, but these components are often overlooked in the testing process. You've surely attempted to use a product's built-in help file and found it to be not so helpful or worse just plain wrong. Or, maybe you've checked the system requirements on a sticker on the side of a software box only to find out after you bought it that the software didn't work on your PC. These seem like simple things to test, but no one probably even gave them a second look before the product was Okayed for release. You will.

Later in this book you'll learn about these non-software pieces and how to properly test them. Until then, keep this list in mind as just a sampling of what more there is to a software product than just the code:

Help files	User's manual
Samples and examples	Labels and stickers
Product support info	Icons and art
Error messages	Ads and marketing material
Setup and installation	Readme file

DON'T FORGET TO TEST ERROR MESSAGES

Error messages are one of the most overlooked parts of a software product. Programmers, not trained writers, typically write them. They're seldom planned for and are usually hacked in while fixing bugs. It's also very difficult for testers to find and display all of them. Don't let error messages such as these creep into your software:

[View full width]

Error: Keyboard not found. Press F1 to continue. Can't instantiate the video thing. Windows has found an unknown device and is installing a driver for it. A Fatal Exception 006 has occurred at 0000:0000007.

SOFTWARE PROJECT STAFF

Now that you know what goes into a software product and what ships with one, it's time to learn about all the people who create software. Of course, this varies a great deal based on the company and the project, but for the most part the roles are the same, it's just the titles that are different.
The following lists, in no particular order, the major players and what they do. The most common names are given, but expect variations and additions:

- Project managers, program managers, or producers drive the project from beginning to end. They're usually responsible for writing the product spec, managing the schedule, and making the critical decisions and trade-offs.
- Architects or system engineers are the technical experts on the product team. They're usually very experienced and therefore are qualified to design the overall systems architecture or design for the software. They work very closely with the programmers.
- Programmers, developers, or coders design and write software and fix the bugs that are found. They work closely with the architects and project managers to create the software. Then, they work closely with the project managers and testers to get the bugs fixed.
- Testers or QA (Quality Assurance) Staff is responsible for finding and reporting problems in the software product. They work very closely with all members of the team as they develop and run their tests, and report the problems they find. Chapter 21, "Software Quality Assurance," thoroughly covers the differences between software testing and software quality assurance tasks.
- Technical writers, user assistance, user education, manual writers, or illustrators create the paper and online documentation that comes with a software product.
- Configuration management or builder handles the process of pulling together all the software written by the programmers and all the documentation created by the writers and putting it together into a single package.

As you can see, several groups of people contribute to a software product. On large teams there may be dozens or hundreds working together. To successfully communicate and organize their approach, they need a plan, a method for getting from point A to point B. That's what the next section is about

SOFTWARE DEVELOPMENT LIFECYCLE MODELS

A running joke in the computer industry is that three things should never be seen in the process of being created: laws, sausage, and software. Their creation process is so messy and disgusting that it's best to just wait and see the final result. That may or may not be totally true, but with most old sayings, there is

a grain of truth behind the words. Some software is developed with the rigor and discipline of a fine craftsman, some software with tightly controlled chaos, and other software is stuck together with duct tape and chewing gum. Usually, in the end, it's apparent to the customer what process was used. The process used to create a software product from its initial conception to its public release is known as the software development lifecycle model.
As discussed previously, there are many different methods that can be used for developing software, and no model is necessarily the best for a particular project. There are four frequently used models, with most others just variations of these:

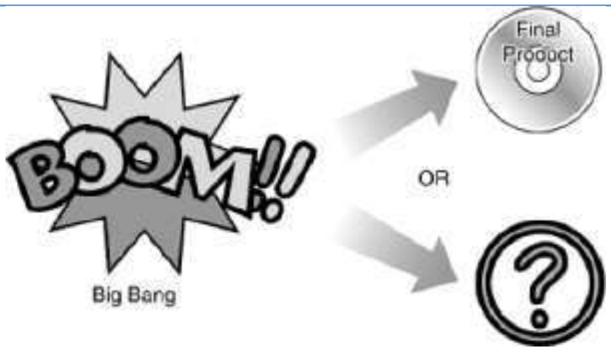
- Big-Bang
- Code-and-Fix
- Waterfall
- Spiral

Each model has its advantages and disadvantages. As a tester, you will likely encounter them all and will need to tailor your test approach to fit the model being used for your current project. Refer to these model descriptions as you read the rest of this book and think about how you would apply the various testing techniques you learn under each of them.

BIG-BANG MODEL

One theory of the creation of the universe is the big-bang theory. It states that billions of years ago, the universe was created in a single huge explosion of nearly infinite energy. Everything that exists is the result of energy and matter lining up to produce this book, DVDs, and Bill Gates. If the atoms didn't line up just right, these things might all be just quivering masses of goop.
The big-bang model for software development shown in Figure 2.4 follows much the same principle. A huge amount of matter (people and money) is put together, a lot of energy is expended often violently and out comes the perfect software product…or it doesn't.

FIGURE 2.4. THE BIG-BANG MODEL IS BY FAR THE SIMPLEST METHOD OF SOFTWARE DEVELOPMENT.



The beauty of the big-bang method is that it's simple. There is little if any planning, scheduling, or formal development process. All the effort is spent developing the software and writing the code. It's a process that is used if the product requirements aren't well understood and the final release date is completely flexible. It's also important to have very flexible customers, too, because they won't know what they're getting until the very end.

Notice that testing isn't shown in Figure 2.4. In most cases, there is little to no formal testing done under the big-bang model. If testing does occur, it's squeezed in just before the product is released. It's a mystery why testing is sometimes inserted into this model, but it's probably to make everyone feel good that some testing was performed.

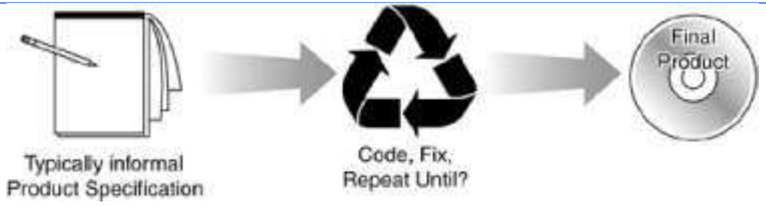
If you are called in to test a product under the big-bang model, you have both an easy and a difficult task. Because the software is already complete, you have the perfect specification the product itself. And, because it's impossible to go back and fix things that are broken, your job is really just to report what you find so the customers can be told about the problems.

The downside is that, in the eyes of project management, the product is ready to go, so your work is holding up delivery to the customer. The longer you take to do your job and the more bugs you find, the more contentious the situation will become. Try to stay away from testing in this model.

CODE-AND-FIX MODEL

The code-and-fix model shown in Figure 2.5 is usually the one that project teams fall into by default if they don't consciously attempt to use something else. It's a step up, procedurally, from the big-bang model, in that it at least requires some idea of what the product requirements are.

FIGURE 2.5. THE CODE-AND-FIX MODEL REPEATS UNTIL SOMEONE GIVES UP.



A wise man once said, "There's never time to do it right, but there's always time to do it over." That pretty much sums up this model. A team using this approach usually starts with a rough idea of what they want, does some simple design, and then proceeds into a long repeating cycle of coding, testing, and fixing bugs. At some point they decide that enough is enough and release the product.

As there's very little overhead for planning and documenting, a project team can show results immediately. For this reason, the code-and-fix model works very well for small projects intended to be created quickly and then thrown out shortly after they're done, such as prototypes and demos. Even so, code-and-fix has been used on many large and well-known software products. If your word processor or spreadsheet software has lots of little bugs or it just doesn't seem quite finished, it was likely created with the code-and-fix model.

Like the big-bang model, testing isn't specifically called out in the code-and-fix model but does play a significant role between the coding and the fixing.

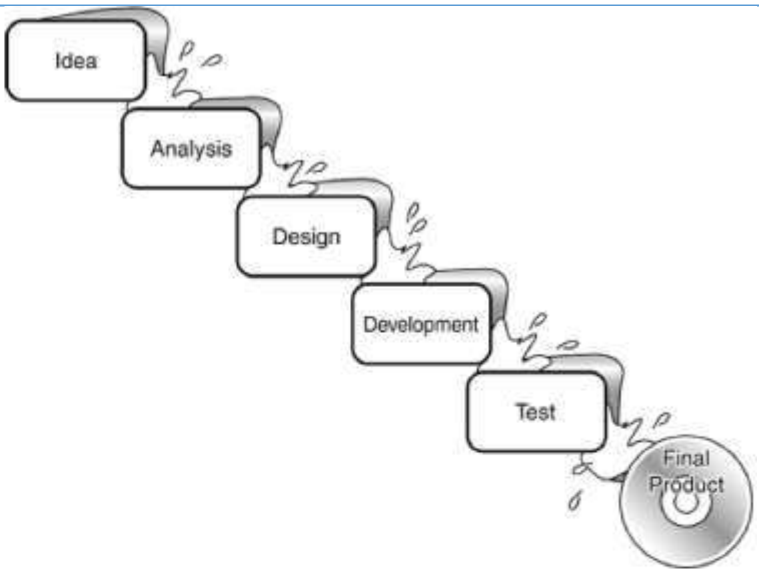
As a tester on a code-and-fix project, you need to be aware that you, along with the programmers, will be in a constant state of cycling. As often as every day you'll be given new or updated releases of the software and will set off to test it. You'll run your tests, report the bugs, and then get a new software release. You may not have finished testing the previous release when the new one arrives, and the new one may have new or changed features. Eventually, you'll get a chance to test most of the features, find fewer and fewer bugs, and then someone (or the schedule) will decide that it's time to release the product.

You will most likely encounter the code-and-fix model during your work as a software tester. It's a good introduction to software development and will help you appreciate the more formal methods.

WATERFALL MODEL

The waterfall method is usually the first one taught in programming school. It's been around forever. It's simple, elegant, and makes sense. And, it can work well on the right project. Figure 2.6 shows the steps involved in this model.

FIGURE 2.6. THE SOFTWARE DEVELOPMENT PROCESS FLOWS FROM ONE STEP TO THE NEXT IN THE WATERFALL MODEL.



A project using the waterfall model moves down a series of steps starting from an initial idea to a final product. At the end of each step, the project team holds a review to determine if they're ready to move to the next step. If the project isn't ready to progress, it stays at that level until it's ready.

Notice three important things about the waterfall method:

- There's a large emphasis on specifying what the product will be. Note that the development or coding phase is only a single block!
- The steps are discrete; there's no overlap.
- There's no way to back up. As soon as you're on a step, you need to complete the tasks for that step and then move on you can't go back.[1]

[1] Variations of the waterfall model loosen the rules a bit, allowing some overlap of the steps and the ability to back up one step if necessary.

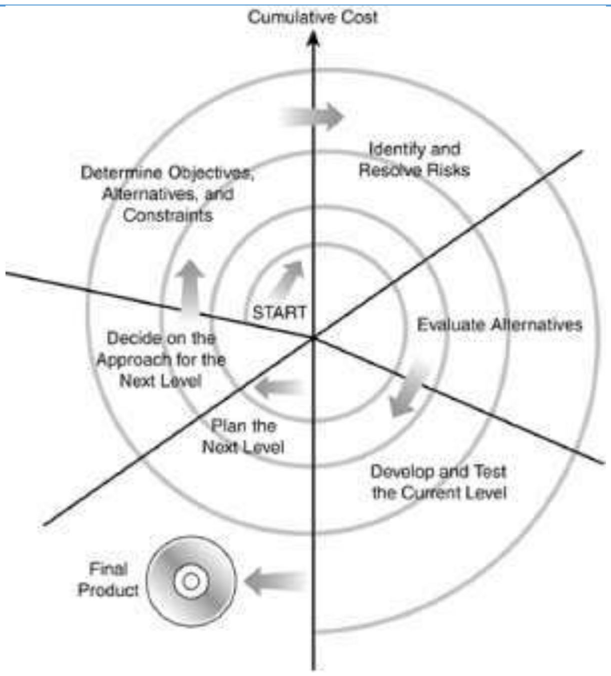
This may sound very limiting, and it is, but it works well for projects with a well-understood product definition and a disciplined development staff. The goal is to work out all the unknowns and nail down all the details before the first line of code is written. The drawback is that in today's fast moving culture,

with products being developed on Internet time, by the time a software product is so carefully thought out and defined, the original reason for its being may have changed.
From a testing perspective, the waterfall model offers one huge advantage over the other models presented so far. Everything is carefully and thoroughly specified. By the time the software is delivered to the test group, every detail has been decided on, written down, and turned into software. From that, the test group can create an accurate plan and schedule. They know exactly what they're testing, and there's no question about whether something is a feature or a bug.
But, with this advantage, comes a large disadvantage. Because testing occurs only at the end, a fundamental problem could creep in early on and not be detected until days before the scheduled product release. Remember from Chapter 1, "Software Testing Background," how the cost of bugs increases over time? What's needed is a model that folds the testing tasks in earlier to find problems before they become too costly.

SPIRAL MODEL

It's not quite utopia, but the spiral model (see Figure 2.7) goes a long way in addressing many of the problems inherent with the other models while adding a few of its own nice touches.

FIGURE2.7. THE SPIRAL MODEL STARTS SMALL AND GRADUALLY EXPANDS AS THE PROJECT BECOMES BETTER DEFINED AND GAINS STABILITY.


Cumulative Cost
Determine Objectives, Alternatives, and Constraints
Identify and Resolve Risks
START
Evaluate Alternatives
Decide on the Approach for the Next Level
Plan the Next Level
Develop and Test the Current Level
Final Product

The spiral model was introduced by Barry Boehm in 1986 in his Association for Computing Machinery (ACM) paper, "A Spiral Model of Software Development and Enhancement." It's used fairly often and has proven to be an effective approach to developing software.

The general idea behind the spiral model is that you don't define everything in detail at the very beginning. You start small, define your important features, try them out, get feedback from your customers, and then move on to the next level. You repeat this until you have your final product.

Each time around the spiral involves six steps:

1. Determine objectives, alternatives, and constraints.
2. Identify and resolve risks.
3. Evaluate alternatives.
4. Develop and test the current level.
5. Plan the next level.
6. Decide on the approach for the next level.

Built into the spiral model is a bit of waterfall (the steps of analysis, design, develop, test), a bit of code-and-fix (each time around the spiral), and a bit of big-bang (look at it from the outside). Couple this with the lower costs of finding problems early, and you have a pretty good development model.

If you're a tester, you'll like this model. You'll get a chance to influence the product early by being involved in the preliminary design phases. You'll see where the project has come from and where it's going. And, at the very end of the project, you won't feel as rushed to perform all your testing at the last minute. You've been testing all along, so the last push should only be a validation that everything is okay.

AGILE SOFTWARE DEVELOPMENT

A type of development process that has gained in popularity among a number of software companies is known as Agile Software Development. You might hear other names for it such as Rapid Prototyping, Extreme Programming, or Evolutionary Development, among others.
The goal of Agile Software Development, as described in its manifesto at www.agilemanifesto.org, is:

> "Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiations
> Responding to change over following a plan…
> That is, while there is value in the items on the right, we value the items on the left more."

Agile Software Development has developed a bit of a following and has had some successful projects but is far from main stream. It sounds like a great philosophy if you and your project seem to be bogged down in process and documentation but, in my opinion, it can too easily digress into chaos. It's a method to watch and learn about and your team may consider using it. For now, however, I won't be riding in a jet whose software was rapidly prototyped by an extreme programmer. Maybe someday.
For more information, check out: Agile Modeling, Teach Yourself Extreme Programming in 24 Hours, and Extreme Programming Practices in Action. All are published by Sams Publishing.

SUMMARY

You now have an understanding of how software products are created both what goes into them and the processes used to put them together. As you can see, there's no definitive approach. The four models presented here are just examples. There are many others and lots of variations of these. Each company, each project, and each team will choose what works for them. Sometimes they will choose right, sometimes they will choose wrong. Your job as a software tester is to work the best you can in the development model you're in, applying the testing skills you learn in the rest of this book to create the best software possible.

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. Name several tasks that should be performed before a programmer starts writing the first line of code.
2. What disadvantage is there to having a formal, locked-down specification?
3. What is the best feature of the big-bang model of software development?
4. When using the code-and-fix model, how do you know when the software is ready to release?
5. Why can the waterfall method be difficult to use?
6. Why would a software tester like the spiral model better than the others?

CHAPTER 3. THE REALITIES OF SOFTWARE TESTING

IN THIS CHAPTER

- **Testing Axioms**
- **Software Testing Terms and Definitions**

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.

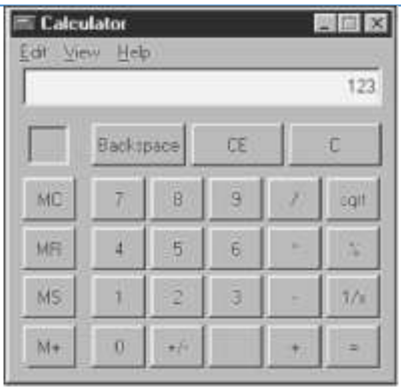
IT'S IMPOSSIBLE TO TEST A PROGRAM COMPLETELY

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.

FIGURE 3.1. EVEN A SIMPLE PROGRAM SUCH AS THE WINDOWS CALCULATOR IS TOO COMPLEX TO COMPLETELY TEST.



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 on screen you can press keys on your computer keyboard, too. Good values to try might be 1+a, z+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. 12+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

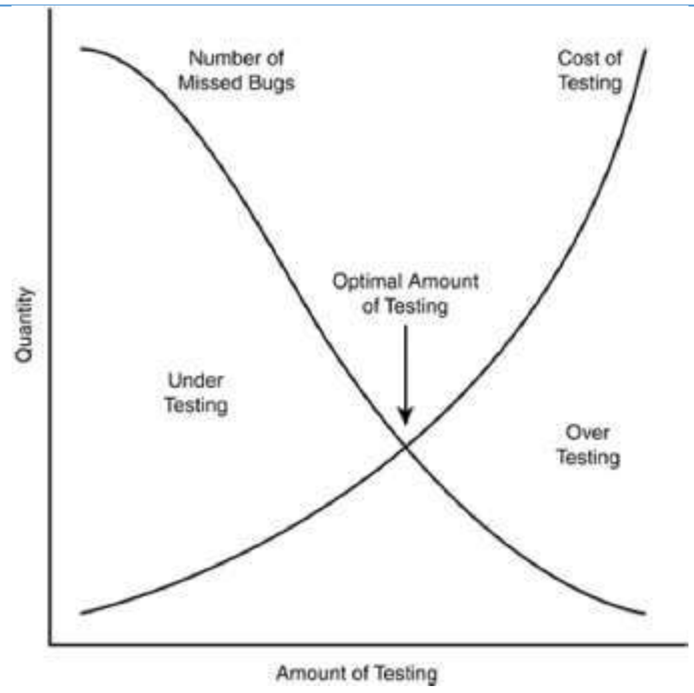
If you decide not to test every possible test scenario, you've chosen to take on risk. In the calculator example, what if you choose not to test that 1024+1024=2048? It's possible the programmer accidentally left in a bug for that situation. If you don't test it, a customer will eventually enter it, 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.

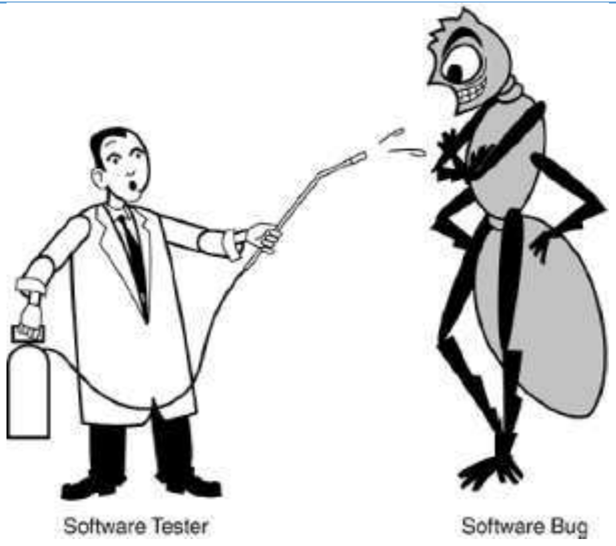
- Some bugs are really just the tip of the iceberg. Very often the software's design or architecture has a fundamental problem. A tester will find several bugs that at first may seem unrelated but eventually are discovered to have one primary serious cause.

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.

FIGURE 3.3. SOFTWARE UNDERGOING THE SAME REPETITIVE TESTS EVENTUALLY BUILDS UP RESISTANCE TO THEM.



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.
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 1exercising 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 other's 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.

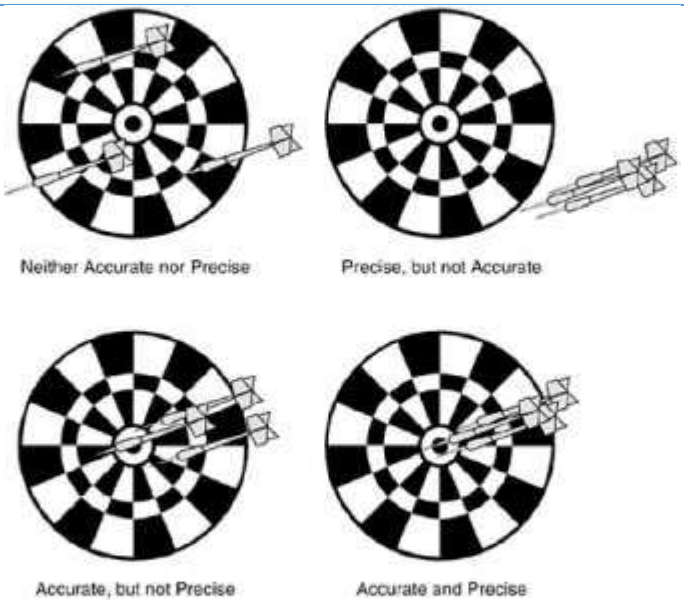
PRECISION AND ACCURACY

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.

FIGURE 3.4. DARTS ON A DARTBOARD DEMONSTRATE THE DIFFERENCE BETWEEN PRECISION AND ACCURACY.


Neither Accurate nor Precise
Precise, but not Accurate
Accurate, but not Precise
Accurate and Precise

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.
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. After that, the precision can vary. 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?

CHAPTER 4. EXAMINING THE SPECIFICATION

IN THIS CHAPTER

- Getting Started
- Performing a High-Level Review of the Specification
- Low-Level Specification Test Techniques

This chapter will introduce you to your first real hands-on testing - but it may not be what you expect. You won't be installing or running software and you won't be pounding on the keyboard hoping for a crash. In this chapter, you'll learn how to test the product's specification to find bugs before they make it into the software.

Testing the product spec isn't something that all software testers have the luxury of doing. Sometimes you might come into a project midway through the development cycle after the specification is written and the coding started. If that's the case, don't worry - you can still use the techniques presented here to test the final specification.

If you're fortunate enough to be involved on the project early and have access to a preliminary specification, this chapter is for you. Finding bugs at this stage can potentially save your project huge amounts of time and money.

Highlights of this chapter include

- What is black-box and white-box testing
- How static and dynamic testing differ
- What high-level techniques can be used for reviewing a product specification
- What specific problems you should look for when reviewing a product specification in detail

GETTING STARTED

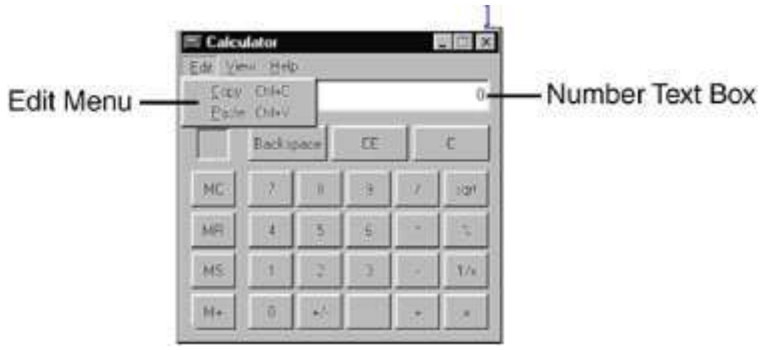
Think back to the four development models presented in Chapter 2, "The Software Development Process": big-bang, code-and-fix, waterfall, and spiral. In each model, except big-bang, the development team creates a product specification from the requirements document to define what the software will become.

Typically, the product specification is a written document using words and pictures to describe the intended product. An excerpt from the Windows Calculator (see Figure 4.1) product spec might read something like this:

The Edit menu will have two selections: Copy and Paste. These can be chosen by one of three methods: pointing and clicking to the menu items with the mouse, using access-keys (Alt+E and then C for Copy and P for Paste), or using the standard Windows shortcut keys of Ctrl+C for Copy and Ctrl+V for Paste.

The Copy function will copy the current entry displayed in the number text box into the Windows Clipboard. The Paste function will paste the value stored in the Windows Clipboard into the number text box.

FIGURE 4.1. THE STANDARD WINDOWS CALCULATOR DISPLAYING THE DROP-DOWN EDIT MENU.

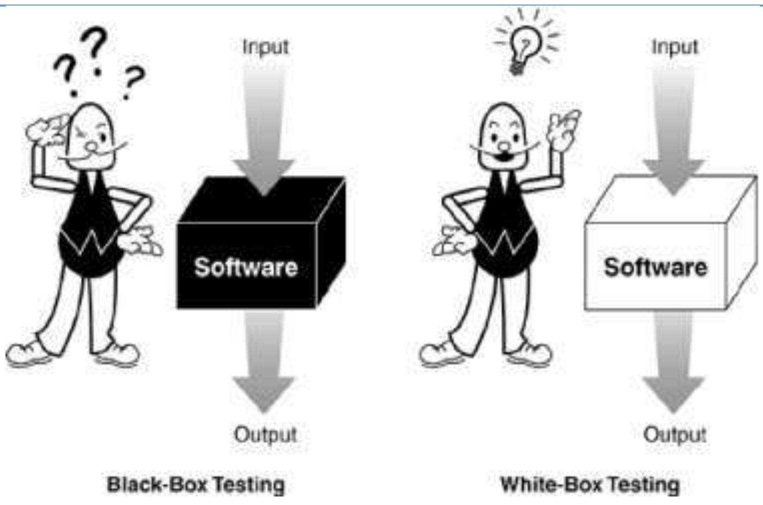


As you can see, it took quite a few words just to describe the operation of two menu items in a simple calculator program. A thoroughly detailed spec for the entire application could be a hundred pages long. It may seem like overkill to create a meticulous document for such simple software. Why not just let a programmer write a calculator program on his own? The problem is that you would have no idea what you'd eventually get. The programmer's idea of what it should look like, what functionality it should have, and how the user would use it could be completely different from yours. The only way to assure that the end product is what the customer required - and to properly plan the test effort - is to thoroughly describe the product in a specification.
The other advantage of having a detailed spec, and the basis of this chapter, is that as a tester you'll also have a document as a testable item. You can use it to find bugs before the first line of code is written.

BLACK-BOX AND WHITE-BOX TESTING

Two terms that software testers use to describe how they approach their testing are black-box testing and white-box testing. Figure 4.2 shows the difference between the two approaches. In black-box testing, the tester only knows what the software is supposed to do - he can't look in the box to see how it operates. If he types in a certain input, he gets a certain output. He doesn't know how or why it happens, just that it does.

FIGURE 4.2. WITH BLACK-BOX TESTING, THE SOFTWARE TESTER DOESN'T KNOW THE DETAILS OF HOW THE SOFTWARE WORKS.



TIP
Black-box testing is sometimes referred to as functional testing or behavioral testing. Don't get caught up in what the actual terms are, as your team may use something different. It's your job to understand their meaning and how they're used by your team.

Think about the Windows Calculator shown in Figure 4.1. If you type 3.14159 and press the sqrt button, you get 1.772453102341. With black-box testing, it doesn't matter what gyrations the software goes through to compute the square root of pi. It just does it. As a software tester, you can verify the result on another "certified" calculator and determine if the Windows Calculator is functioning correctly.
In white-box testing (sometimes called clear-box testing), the software tester has access to the program's code and can examine it for clues to help him with his testing - he can see inside the box. Based on what he sees, the tester may determine that certain numbers are more or less likely to fail and can tailor his testing based on that information.
NOTE
There is a risk to white-box testing. It's very easy to become biased and fail to objectively test the software because you might tailor the tests to match the code's operation.

STATIC AND DYNAMIC TESTING

Two other terms used to describe how software is tested are static testing and dynamic testing. Static testing refers to testing something that's not running - examining and reviewing it. Dynamic testing is what you would normally think of as testing - running and using the software.
The best analogy for these terms is the process you go through when checking out a used car. Kicking the tires, checking the paint, and looking under the hood are static testing techniques. Starting it up, listening to the engine, and driving down the road are dynamic testing techniques.

STATIC BLACK-BOX TESTING: TESTING THE SPECIFICATION

Testing the specification is static black-box testing. The specification is a document, not an executing program, so it's considered static. It's also something that was created using data from many sources - usability studies, focus groups, marketing input, and so on. You don't necessarily need to know how or why that information was obtained or the details of the process used to obtain it, just that it's been boiled down into a product specification. You can then take that document, perform static black-box testing, and carefully examine it for bugs.

Earlier you saw an example of a product specification for the Windows Calculator. This example used a standard written document with a picture to describe the software's operation. Although this is the most common method for writing a spec, there are lots of variations. Your development team may emphasize diagrams over words or it may use a self-documenting computer language such as Ada. Whatever their choice, you can still apply all the techniques presented in this chapter. You will have to tailor them to the spec format you have, but the ideas are still the same.

What do you do if your project doesn't have a spec? Maybe your team is using the big-bang model or a loose code-and-fix model. As a tester, this is a difficult position. Your goal is to find bugs early - ideally finding them before the software is coded - but if your product doesn't have a spec, this may seem impossible to do. Although the spec may not be written down, someone, or several people, know what they're trying to build. It may be the developer, a project manager, or a marketer. Use them as the walking, talking, product spec and apply the same techniques for evaluating this "mental" specification as though it was written on paper. You can even take this a step further by recording the information you gather and circulating it for review.

Tell your project team, "This is what I plan to test and submit bugs against." You'll be amazed at how many details they'll immediately fill in.

TIP

You can test a specification with static black-box techniques no matter what the format of the specification. It can be a written or graphical document or a combination of both. You can even test an unwritten specification by questioning the people who are designing and writing the software.

PERFORMING A HIGH-LEVEL REVIEW OF THE SPECIFICATION

Defining a software product is a difficult process. The spec must deal with many unknowns, take a multitude of changing inputs, and attempt to pull them all together into a document that describes a new product. The process is an inexact science and is prone to having problems.

The first step in testing the specification isn't to jump in and look for specific bugs. The first step is to stand back and view it from a high level. Examine the spec for large fundamental problems, oversights, and omissions. You might consider this more research than testing, but ultimately the research is a means to better understand what the software should do. If you have a better understanding of the whys and hows behind the spec, you'll be much better at examining it in detail.

PRETEND TO BE THE CUSTOMER

The easiest thing for a tester to do when he first receives a specification for review is to pretend to be the customer. Do some research about who the customers will be. Talk to your marketing or sales people to get hints on what they know about the end user. If the product is an internal software project, find out who will be using it and talk to them.

It's important to understand the customer's expectations. Remember that the definition of quality means "meeting the customer's needs." As a tester, you must understand those needs to test that the software meets them. To do this effectively doesn't mean that you must be an expert in the field of nuclear physics if you're testing software for a power plant, or that you must be a professional pilot if you're testing a flight simulator. But, gaining some familiarity with the field the software applies to would be a great help.

Above all else, assume nothing. If you review a portion of the spec and don't understand it, don't assume that it's correct and go on. Eventually, you'll have to use this specification to design your software tests, so you'll eventually have to understand it. There's no better time to learn than now. If you find bugs along the way (and you will), all the better.

TIP

Don't forget about software security when pretending to be the customer. Your users will assume that the software is secure, but you can't assume that the programmers will handle security issues properly. It must be specified - in detail. Chapter 13, "Testing for Software Security," discusses what you should look for when reviewing the product design and specification for security issues.

RESEARCH EXISTING STANDARDS AND GUIDELINES

Back in the days before Microsoft Windows and the Apple Macintosh, nearly every software product had a different user interface. There were different colors, different menu structures, unlimited ways to open a file, and myriad cryptic commands to get the same tasks done. Moving from one software product to another required complete retraining.

Thankfully, there has been an effort to standardize the hardware and the software. There has also been extensive research done on how people use computers. The result is that we now have products reasonably similar in their look and feel that have been designed with ergonomics in mind. You may argue that the adopted standards and guidelines aren't perfect, that there may be better ways to get certain tasks done, but efficiency has greatly improved because of this commonality.

Chapter 11, "Usability Testing," will cover this topic in more detail, but for now you should think about what standards and guidelines might apply to your product.

NOTE

The difference between standards and guidelines is a matter of degree. A standard is much more firm than a guideline. Standards should be strictly adhered to if your team has decided that it's important to comply with them completely. Guidelines are optional but should be followed. It's not uncommon for a team to decide to use a standard as a guideline - as long as everyone knows that is the plan.

Here are several examples of standards and guidelines to consider. This list isn't definitive. You should research what might apply to your software:

- Corporate Terminology and Conventions. If this software is tailored for a specific company, it should adopt the common terms and conventions used by the employees of that company.
- Industry Requirements. The medical, pharmaceutical, industrial, and financial industries have very strict standards that their software must follow.
- Government Standards. The government, especially the military, has strict standards.
- Graphical User Interface (GUI). If your software runs under Microsoft Windows or Apple Macintosh operating systems, there are published standards and guidelines for how the software should look and feel to a user.
- Security Standards. Your software and its interfaces and protocols may need to meet certain security standards or levels. It may also need to be independently certified that it does, indeed, meet the necessary criteria.

As a tester, your job isn't to define what guidelines and standards should be applied to your software. That job lies with the project manager or whoever is writing the specification. You do, however, need to perform your own investigation to "test" that the correct standards are being used and that none are overlooked. You also have to be aware of these standards and test against them when you verify and validate the software. Consider them as part of the specification.

REVIEW AND TEST SIMILAR SOFTWARE

One of the best methods for understanding what your product will become is to research similar software. This could be a competitor's product or something similar to what your team is creating. It's likely that the project manager or others who are specifying your product have already done this, so it should be relatively easy to get access to what products they used in their research. The software likely won't be an exact match (that's why you're creating new software, right?), but it should help you think about test situations and test approaches. It should also flag potential problems that may not have been considered.
Some things to look for when reviewing competitive products include

- Scale. Will there be fewer or greater features? Will there be less or more code? Will that size difference matter in your testing?
- Complexity. Will your software be more or less complex? Will this impact your testing?
- Testability. Will you have the resources, time, and expertise to test software such as this?

- Quality/Reliability. Is this software representative of the overall quality planned for your software? Will your software be more or less reliable?
- Security. How does the competitor's software security, both advertised and actual, compare to what you'll be offering?

There's no substitute for hands-on experience, so do whatever you can to get a hold of similar software, use it, bang on it, and put it through its paces. You'll gain a lot of experience that will help you when you review your specification in detail.
TIP
Don't forget about reading online and printed software reviews and articles about the competition. This can be especially helpful for security issues as you may not likely see the security flaws as you casually use the application. They will, though, be well known in the press.

LOW-LEVEL SPECIFICATION TEST TECHNIQUES

After you complete the high-level review of the product specification, you'll have a better understanding of what your product is and what external influences affect its design. Armed with this information, you can move on to testing the specification at a lower level. The remainder of this chapter explains the specifics for doing this.[1]

[1] The checklists are adapted from pp.294-295 and 303-308 of the Handbook of Walkthroughs, Inspections, and Technical Reviews, 3rd Edition Copyright 1990, 1982 by D.P. Freedman and G.M. Weinberg. Used by permission of Dorset House Publishing (www.dorsethouse.com). All rights reserved.

SPECIFICATION ATTRIBUTES CHECKLIST

A good, well-thought-out product specification, with "all its t's crossed and its i's dotted," has eight important attributes:

- Complete. Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
- Accurate. Is the proposed solution correct? Does it properly define the goal? Are there any errors?
- Precise, Unambiguous, and Clear. Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understand?
- Consistent. Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
- Relevant. Is the statement necessary to specify the feature? Is it extra information that should be left out? Is the feature traceable to an original customer need?
- Feasible. Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
- Code-free. Does the specification stick with defining the product and not the underlying software design, architecture, and code?
- Testable. Can the feature be tested? Is enough information provided that a tester could create tests to verify its operation?

When you're testing a product spec, reading its text, or examining its figures, carefully consider each of these traits. Ask yourself if the words and pictures you're reviewing have these attributes. If they don't, you've found a bug that needs to be addressed.

SPECIFICATION TERMINOLOGY CHECKLIST

A complement to the previous attributes list is a list of problem words to look for while reviewing a specification. The appearance of these words often signifies that a feature isn't yet completely thought out - it likely falls under one of the preceding attributes. Look for these words in the specification and carefully review how they're used in context. The spec may go on to clarify or elaborate on them, or it may leave them ambiguous - in which case, you've found a bug.

- Always, Every, All, None, Never. If you see words such as these that denote something as certain or absolute, make sure that it is, indeed, certain. Put on your tester's hat and think of cases that violate them.
- Certainly, Therefore, Clearly, Obviously, Evidently. These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- Some, Sometimes, Often, Usually, Ordinarily, Customarily, Most, Mostly. These words are too vague. It's impossible to test a feature that operates "sometimes."
- Etc., And So Forth, And So On, Such As. Lists that finish with words such as these aren't testable. Lists need to be absolute or explained so that there's no confusion as to how the series is generated and what appears next in the list.
- Good, Fast, Cheap, Efficient, Small, Stable. These are unquantifiable terms. They aren't testable. If they appear in a specification, they must be further defined to explain exactly what they mean.
- Handled, Processed, Rejected, Skipped, Eliminated. These terms can hide large amounts of functionality that need to be specified.
- If…Then…(but missing Else). Look for statements that have "If…Then" clauses but don't have a matching "Else." Ask yourself what will happen if the "if" doesn't happen.

SUMMARY

After completing this chapter, you may have decided that testing a specification is a very subjective process. High-level review techniques will flush out oversights and omissions, and low-level techniques help assure that all the details are defined. But, these techniques aren't really step-by-step processes to follow, for two reasons:

- This is an introductory book whose aim is to get you rapidly up the testing curve. The material presented here will do just that. Armed with the information presented in this chapter, you will make a big dent in any software spec you're given to test.
- The format of specifications can vary widely. You'll be able to apply the techniques from this chapter whether you're pulling the spec out of someone's brain, looking at a high-level diagram, or parsing through sentences. You will find bugs.

If you're interested in pursuing more advanced techniques for reviewing specifications, do some research on the work of Michael Fagan. While at IBM, Mr. Fagan pioneered a detailed and methodical approach called software inspections that many companies use, especially companies creating mission-critical software, to formally review their software specifications and code. You can find more information on his website: www.mfagan.com.

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. Can a software tester perform white-box testing on a specification?
2. Cite a few examples of Mac or Windows standards or guidelines.
3. Explain what's wrong with this specification statement: When the user selects the Compact Memory option, the program will compress the mailing list data as small as possible using a Huffman-sparse-matrix approach.
4. Explain what a tester should worry about with this line from a spec: The software will allow up to 100 million simultaneous connections, although no more than 1 million will normally be used.

CHAPTER 5. TESTING THE SOFTWARE WITH BLINDERS ON

IN THIS CHAPTER

- Dynamic Black-Box Testing: Testing the Software While Blindfolded
- Test-to-Pass and Test-to-Fail
- Equivalence Partitioning
- Data Testing
- State Testing
- Other Black-Box Test Techniques

Okay, now for the good stuff! This chapter covers what most people imagine when they think of software testing. It's time to crack your knuckles, sit in front of your computer, and start looking for bugs.

As a new software tester, this may be the first job you're assigned to do. If you're interviewing for a software test position, you will no doubt be asked how you'd approach testing a new software program or a new program's features.

It's very easy to jump right in, start pounding on keys, and hope that something breaks. Such an approach might work for a little while. If the software is still under development, it's very easy to get lucky and find a few bugs right away. Unfortunately, those easy pickings will quickly disappear and you'll need a more structured and targeted approach to continue finding bugs and to be a successful software tester.

This chapter describes the most common and effective techniques for testing software. It doesn't matter what kind of program you're testing - the same techniques will work whether it's a custom accounting package for your company, an industrial automation program, or a mass-market shoot-'em-up computer game.

You also don't need to be a programmer to use these techniques. Although they're all based on fundamental programming concepts, they don't require you to write code. A few techniques have some background information that explains why they're effective, but any code samples are short and written in a simple macro language to easily demonstrate the point. If you're into programming and want to learn more low-level test techniques, after you finish reading this chapter, move on to Chapter 6, "Examining the Code," and Chapter 7, "Testing the Software with X-Ray Glasses," the white-box testing chapters.

Topics covered in this chapter include

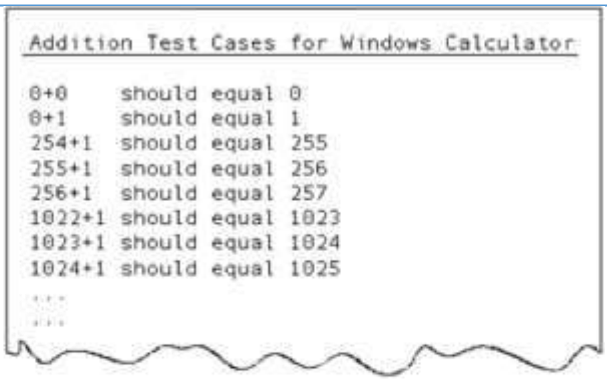
- What is dynamic black-box testing?
- How to reduce the number of test cases by equivalence partitioning
- How to identify troublesome boundary conditions
- Good data values to use to induce bugs
- How to test software states and state transitions
- How to use repetition, stress, and high loads to locate bugs
- A few secret places where bugs hide

DYNAMIC BLACK-BOX TESTING: TESTING THE SOFTWARE WHILE BLINDFOLDED

Testing software without having an insight into the details of underlying code is dynamic black-box testing. It's dynamic because the program is running - you're using it as a customer would. And, it's black-

box because you're testing it without knowing exactly how it works - with blinders on. You're entering inputs, receiving outputs, and checking the results. Another name commonly used for dynamic black-box testing is behavioral testing because you're testing how the software actually behaves when it's used.
To do this effectively requires some definition of what the software does - namely, a requirements document or product specification. You don't need to be told what happens inside the software "box" - you just need to know that inputting A outputs B or that performing operation C results in D. A good product spec will provide you with these details.
Once you know the ins and outs of the software you're about to test, your next step is to start defining the test cases. Test cases are the specific inputs that you'll try and the procedures that you'll follow when you test the software. Figure 5.1 shows an example of several cases that you might use for testing the addition function of the Windows Calculator.

FIGURE 5.1. TEST CASES SHOW THE DIFFERENT INPUTS AND THE STEPS TO TEST A PROGRAM.



```
Addition Test Cases for Windows Calculator

0+0     should equal 0
0+1     should equal 1
254+1   should equal 255
255+1   should equal 256
256+1   should equal 257
1022+1  should equal 1023
1023+1  should equal 1024
1024+1  should equal 1025
...
...
```

NOTE
Selecting test cases is the single most important task that software testers do. Improper selection can result in testing too much, testing too little, or testing the wrong things. Intelligently weighing the risks and reducing the infinite possibilities to a manageable effective set is where the magic is.

The rest of this chapter and much of the rest of the book will teach you how to strategically select good test cases. Chapter 18, "Writing and Tracking Test Cases," discusses the specific techniques for writing and managing test cases.

USE EXPLORATORY TESTING IF YOU DON'T HAVE A SPEC

A professional, mature software development process will have a detailed specification for the software. If you're stuck in a big-bang model or a sloppy code-and-fix model, you may not have a software spec to base your tests on. That's not an ideal situation for a software tester, but you can use a workable solution known as exploratory testing - simultaneously learning the software, designing tests, and executing those tests.
You need to treat the software as the specification. Methodically explore the software feature by feature. Take notes on what the software does, map out the features, and apply some of the static black-box techniques you learned in Chapter 4, "Examining the Specification." Analyze the software as though it is the specification. Then apply the dynamic black-box techniques from this chapter.
You won't be able to test the software as thoroughly as you would if you had a spec - you won't necessarily know if a feature is missing, for example. But, you will be able to systematically test it. In this situation, finding any bugs would be a positive thing.

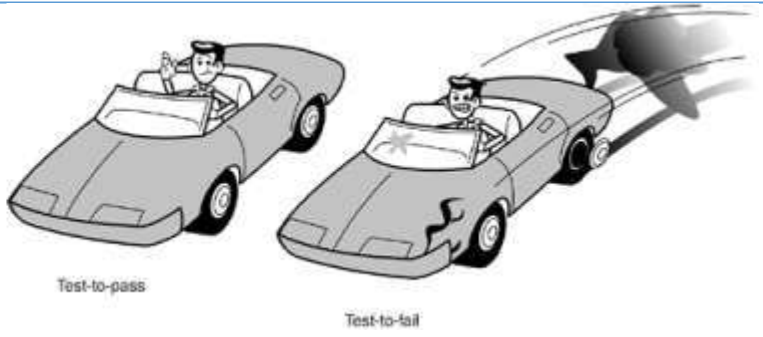
TEST-TO-PASS AND TEST-TO-FAIL

There are two fundamental approaches to testing software: test-to-pass and test-to-fail. When you test-to-pass, you really assure only that the software minimally works. You don't push its capabilities. You don't see what you can do to break it. You treat it with kid gloves, applying the simplest and most straightforward test cases.

You may be thinking that if your goal is to find bugs, why would you test-to-pass? Wouldn't you want to find bugs by any means possible? The answer is no, not initially.

Think about an analogy with a newly designed car (see Figure 5.2). You're assigned to test the very first prototype that has just rolled off the assembly line and has never been driven. You probably wouldn't get in, start it up, head for the test track, and run it wide open at full speed as hard as you could. You'd probably crash and die. With a new car, there'd be all kinds of bugs that would reveal themselves at low speed under normal driving conditions. Maybe the tires aren't the right size, or the brakes are inadequate, or the engine is too loud. You could discover these problems and have them fixed before getting on the track and pushing the limits.

FIGURE 5.2. USE TEST-TO-PASS TO REVEAL BUGS BEFORE YOU TEST-TO-FAIL.



NOTE

When designing and running your test cases, always run the test-to-pass cases first. It's important to see if the software fundamentally works before you throw the kitchen sink at it. You might be surprised how many bugs you find just using the software normally.

After you assure yourself that the software does what it's specified to do in ordinary circumstances, it's time to put on your sneaky, conniving, devious hat and attempt to find bugs by trying things that should force them out. Designing and running test cases with the sole purpose of breaking the software is called testing-to-fail or error-forcing. You'll learn later in this chapter that test-to-fail cases often don't appear intimidating. They often look like test-to-pass cases, but they're strategically chosen to probe for common weaknesses in the software.

ERROR MESSAGES: TEST-TO-PASS OR TEST-TO-FAIL

A common class of test cases is one that attempts to force error messages. You know the ones - like saving a file to a floppy disk but not having one inserted in the drive. These cases actually straddle the line between test-to-pass and test-to-fail. The specification probably states that certain input conditions should result in an error message. That seems pretty clear as a test-to-pass case. But, you're also forcing an error, so it could be viewed as test-to-fail. In the end, it's probably both.

Don't worry about the distinction. What's important is to try to force the error messages that are specified and to invent test cases to force errors that were never considered. You'll likely end up finding both test-to-pass and test-to-fail bugs.

EQUIVALENCE PARTITIONING

Selecting test cases is the single most important task that software testers do and equivalence partitioning, sometimes called equivalence classing, is the means by which they do it. Equivalence partitioning is the process of methodically reducing the huge (infinite) set of possible test cases into a much smaller, but still equally effective, set.
Remember the Windows Calculator example from Chapter 3? It's impossible to test all the cases of adding two numbers together. Equivalence partitioning provides a systematic means for selecting the values that matter and ignoring the ones that don't.
For example, without knowing anything more about equivalence partitioning, would you think that if you tested 1+1, 1+2, 1+3, and 1+4 that you'd need to test 1+5 and 1+6? Do you think you could safely assume that they'd work?

How about 1+99999999999999999999999999999999 (the maximum number you can type in)? Is this test case maybe a little different than the others, maybe in a different class, a different equivalence partition? If you had the choice, would you include it or 1+13?
See, you're already starting to think like a software tester!
NOTE
An equivalence class or equivalence partition is a set of test cases that tests the same thing or reveals the same bug.

What is the difference between 1+99999999999999999999999999999999 and 1+13? In the case of 1+13, it looks like a standard simple addition, a lot like 1+5 or 1+392. However, 1+999… is way out there, on the edge. If you enter the largest possible number and then add 1 to it, something bad might happen - possibly a bug. This extreme case is in a unique partition, a different one from the normal partition of regular numbers.
NOTE
When looking for equivalence partitions, think about ways to group similar inputs, similar outputs, and similar operation of the software. These groups are your equivalence partitions.

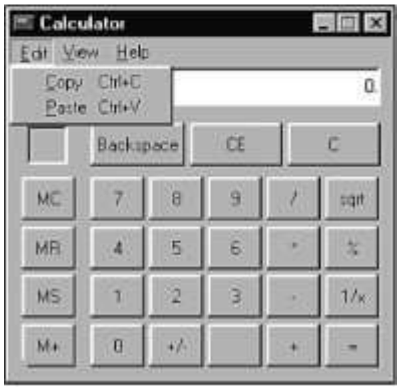
Look at a few examples:

- In the case of adding two numbers together, there seemed to be a distinct difference between testing 1+13 and 1+99999999999999999999999999999999. Call it a gut feeling, but one seemed to be normal addition and the other seemed to be risky. That gut feeling is right. A program would have to handle the addition of 1 to a maxed-out number differently than the addition of two small numbers. It would need to handle an overflow condition. These two cases, because the software most likely operates on them differently, are in different equivalence partitions.

  If you have some programming experience, you might be thinking of several more "special" numbers that could cause the software to operate differently. If you're not a programmer, don't worry - you'll learn the techniques very shortly and be able to apply them without having to understand the code in detail.

- Figure 5.3 shows the Calculator's Edit menu selected to display the copy and paste commands. There are five ways to perform each function. For copy, you click the Copy menu item, type c or C when the menu is displayed, or press Ctrl+c or Ctrl+Shift+c. Each input path copies the current number into the Clipboard - they perform the same output and produce the same result.

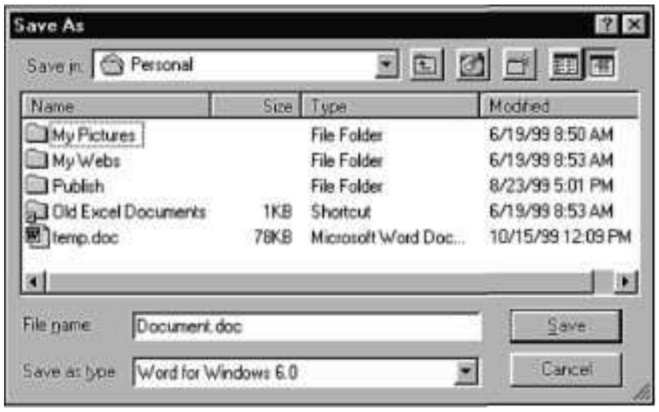
FIGURE 5.3. THE MULTIPLE WAYS TO INVOKE THE COPY FUNCTION ALL HAVE THE SAME RESULT.



If your job is to test the copy command, you could partition these five input paths down to three: Clicking the command on the menu, typing a c, or pressing Ctrl+c. As you grow more confident with the software's quality and know that the copy function, no matter how it's enabled, is working properly, you might even partition these down into a single partition, maybe Ctrl+c.

- As a third example, consider the possibilities for entering a filename in the standard Save As dialog box (see Figure 5.4).

FIGURE 5.4. THE FILE NAME TEXT BOX IN THE SAVE AS DIALOG BOX ILLUSTRATES SEVERAL EQUIVALENCE PARTITION POSSIBILITIES.



Name	Size	Type	Modified
My Pictures		File Folder	6/19/99 8:50 AM
My Webs		File Folder	6/19/99 8:53 AM
Publish		File Folder	8/23/99 5:01 PM
Old Excel Documents	1KB	Shortcut	6/19/99 8:53 AM
temp.doc	78KB	Microsoft Word Doc...	10/15/99 12:09 PM

A Windows filename can contain any characters except \ / : * ? " < > and |. Filenames can have from 1 to 255 characters. If you're creating test cases for filenames, you will have equivalence partitions for valid characters, invalid characters, valid length names, names that are too short, and names that are too long.

Remember, the goal of equivalence partitioning is to reduce the set of possible test cases into a smaller, manageable set that still adequately tests the software. You're taking on risk because you're choosing not to test everything, so you need to be careful how you choose your classes.
NOTE

If you equivalence partition too far in your effort to reduce the number of test cases, you risk eliminating tests that could reveal bugs. If you're new to testing, always get someone with more experience to review your proposed classes.

A final point about equivalence partitioning is that it can be subjective. It's science but it's also art. Two testers who test a complex program may arrive at two different sets of partitions. That's okay as long as the partitions are reviewed and everyone agrees that they acceptably cover the software being tested.

DATA TESTING

The simplest view of software is to divide its world into two parts: the data (or its domain) and the program. The data is the keyboard input, mouse clicks, disk files, printouts, and so on. The program is the executable flow, transitions, logic, and computations. A common approach to software testing is to divide up the test work along the same lines.
When you perform software testing on the data, you're checking that information the user inputs, results that he receives, and any interim results internal to the software are handled correctly.
Examples of data would be

- The words you type into a word processor
- The numbers entered into a spreadsheet
- The number of shots you have remaining in your space game
- The picture printed by your photo software
- The backup files stored on your floppy disk
- The data being sent by your modem over the phone lines

The amount of data handled by even the simplest programs can be overwhelming. Remember all the possibilities of input data for performing simple addition on a calculator? Consider a word processor, a missile guidance system, or a stock trading program. The trick (if you can call it that) to making any of these testable is to intelligently reduce the test cases by equivalence partitioning based on a few key concepts: boundary conditions, sub-boundary conditions, nulls, and bad data.

BOUNDARY CONDITIONS

The best way to describe boundary condition testing is shown in Figure 5.5. If you can safely and confidently walk along the edge of a cliff without falling off, you can almost certainly walk in the middle of a field. If software can operate on the edge of its capabilities, it will almost certainly operate well under normal conditions.

FIGURE 5.5. A SOFTWARE BOUNDARY IS MUCH LIKE THE EDGE OF A CLIFF.

Boundary conditions are special because programming, by its nature, is susceptible to problems at its edges. Software is very binary - something is either true or it isn't. If an operation is performed on a range of numbers, odds are the programmer got it right for the vast majority of the numbers in the middle, but maybe made a mistake at the edges. Listing 5.1 shows how a boundary condition problem can make its way into a very simple program.

LISTING 5.1. A SIMPLE BASIC PROGRAM DEMONSTRATING A BOUNDARY CONDITION BUG

```
Rem Create a 10 element integer array

Rem Initialize each element to -1

Dim data(10) As Integer

Dim i As Integer

For i = 1 To 10

   data(i) = -1

   Next i

End
```

The purpose of this code is to create a 10-element array and initialize each element of the array to 1. It looks fairly simple. An array (data) of 10 integers and a counter (i) are created. A For loop runs from 1 to 10, and each element of the array from 1 to 10 is assigned a value of 1. Where's the boundary problem?

In most BASIC scripts, when an array is dimensioned with a stated range - in this case, Dim data(10) as Integer - the first element created is 0, not 1. This program actually creates a data array of 11 elements from data(0) to data(10). The program loops from 1 to 10 and initializes those values of the array to 1, but since the first element of our array is data(0), it doesn't get initialized. When the program completes, the array values look like this:

data(0) = 0	data(6) = 1
data(1) = 1	data(7) = 1
data(2) = 1	data(8) = 1
data(3) = 1	data(9) = 1
data(4) = 1	data(10) = 1
data(5) = 1	

Notice that data(0)'s value is 0, not 1. If the same programmer later forgot about, or a different programmer wasn't aware of how this data array was initialized, he might use the first element of the array, data(0), thinking it was set to 1. Problems such as this are very common and, in large complex software, can result in very nasty bugs.

TYPES OF BOUNDARY CONDITIONS

Now it's time to open your mind and really think about what constitutes a boundary. Beginning testers often don't realize how many boundaries a given set of data can have. Usually there are a few obvious ones, but if you dig deeper you'll find the more obscure, interesting, and often bug-prone boundaries.

NOTE

Boundary conditions are those situations at the edge of the planned operational limits of the software.

When you're presented with a software test problem that involves identifying boundaries, look for the following types:

Numeric	Speed
Character	Location
Position	Size
Quantity	

And, think about the following characteristics of those types:

First/Last	Min/Max

First/Last	Min/Max
Start/Finish	Over/Under
Empty/Full	Shortest/Longest
Slowest/Fastest	Soonest/Latest
Largest/Smallest	Highest/Lowest
Next-To/Farthest-From	

These are not by any means definitive lists. They cover many of the possible boundary conditions, but each software testing problem is different and may involve very different data with very unique boundaries.
TIP
If you have a choice of what data you're going to include in your equivalence partition, choose data that lies on the boundary.

TESTING THE BOUNDARY EDGES

What you've learned so far is that you need to create equivalence partitions of the different data sets that your software operates on. Since software is susceptible to bugs at the boundaries, if you're choosing what data to include in your equivalence partition, you'll find more bugs if you choose data from the boundaries.
But testing the data points just at the edge of the boundary line isn't usually sufficient. As the words to the "Hokey Pokey" imply ("Put your right hand in, put your right hand out, put your right hand in, and you shake it all about…"), it's a good idea to test on both sides of the boundary - to shake things up a bit.
You'll find the most bugs if you create two equivalence partitions. The first should contain data that you would expect to work properly - values that are the last one or two valid points inside the boundary. The second partition should contain data that you would expect to cause an error - the one or two invalid points outside the boundary.
TIP
When presented with a boundary condition, always test the valid data just inside the boundary, test the last possible valid data, and test the invalid data just outside the boundary.

Testing outside the boundary is usually as simple as adding one, or a bit more, to the maximum value and subtracting one, or a bit more, from the minimum value. For example:

- First1/Last+1
- Start1/Finish+1
- Less than Empty/More than Full
- Even Slower/Even Faster
- Largest+1/Smallest1
- Min1/Max+1
- Just Over/Just Under
- Even Shorter/Longer
- Even Sooner/Later
- Highest+1/Lowest1

Look at a few examples so you can start thinking about all the boundary possibilities:

- If a text entry field allows 1 to 255 characters, try entering 1 character and 255 characters as the valid partition. You might also try 254 characters as a valid choice. Enter 0 and 256 characters as the invalid partitions.
- If a program reads and writes to a CD-R, try saving a file that's very small, maybe with one entry. Save a file that's very large - just at the limit for what the disc holds. Also try saving an empty file and a file that's too large to fit on the disc.
- If a program allows you to print multiple pages onto a single page, try printing just one (the standard case) and try printing the most pages that it allows. If you can, try printing zero pages and one more than it allows.
- Maybe the software has a data-entry field for a 9-digit ZIP code. Try 00000-0000, the simplest and smallest. Try entering 99999-9999 as the largest. Try entering one more or one less digit than what's allowed.
- If you're testing a flight simulator, try flying right at ground level and at the maximum allowed height for your plane. Try flying below ground level and below sea level as well as into outer space.

Since you can't test everything, performing equivalence partitioning around boundary conditions, such as in these examples, to create your test cases is critical. It's the most effective way to reduce the amount of testing you need to perform.

NOTE

It's vitally important that you continually look for boundaries in every piece of software you work with. The more you look, the more boundaries you'll discover, and the more bugs you'll find.

NOTE

Buffer Overruns are caused by boundary condition bugs. They are the number one cause of software security issues. Chapter 13, "Testing for Software Security," discusses the specific situations that cause buffer overruns and how you can test for them.

SUB-BOUNDARY CONDITIONS

The normal boundary conditions just discussed are the most obvious to find. They're the ones defined in the specification or evident when using the software. Some boundaries, though, that are internal to the software aren't necessarily apparent to an end user but still need to be checked by the software tester. These are known as sub-boundary conditions or internal boundary conditions.

These boundaries don't require that you be a programmer or that you be able to read the raw code that you're testing, but they do require a bit of general knowledge about how software works. Two examples are powers-of-two and the ASCII table. The software that you're testing can have many others, so you should talk with your team's programmers to see if they can offer suggestions for other sub-boundary conditions that you should check.

POWERS-OF-TWO

Computers and software are based on binary numbers - bits representing 0s and 1s, bytes made up of 8 bits, words (on 32-bit systems) made up of 4 bytes, and so on. Table 5.1 shows the common powers-of-two units and their equivalent values.

TABLE 5.1. SOFTWARE POWERS-OF-TWO

Term	Range or Value
Bit	0 or 1

TABLE 5.1. SOFTWARE POWERS-OF-TWO

Term	Range or Value
Nibble	015
Byte	0255
Word	04,294,967,295
Kilo	1,024
Mega	1,048,576
Giga	1,073,741,824
Tera	1,099,511,627,776

The ranges and values shown in Table 5.1 are critical values to treat as boundary conditions. You likely won't see them specified in a requirements document unless the software presents the same range to the user. Often, though, they're used internally by the software and are invisible, unless of course they create a situation for a bug.

AN EXAMPLE OF POWERS-OF-TWO

An example of how powers-of-two come into play is with communications software. Bandwidth, or the transfer capacity of your information, is always limited. There's always a need to send and receive information faster than what's possible. For this reason, software engineers try to pack as much data into communications strings as they can.

One way they do this is to compress the information into the smallest units possible, send the most common information in these small units, and then expand to the next size units as necessary.

Suppose that a communications protocol supports 256 commands. The software could send the most common 15 commands encoded into a small nibble of data. For the 16th through 256th commands, the software could then switch over to send the commands encoded into the longer bytes.

The software user knows only that he can issue 256 commands; he doesn't know that the software is performing special calculations and different operations on the nibble/byte boundary.

When you create your equivalence partitions, consider whether powers-of-two boundary conditions need to be included in your partition. For example, if your software accepts a range of numbers from 1 to 1000, you've learned to include in your valid partition 1 and 1000, maybe 2 and 999. To cover any possible powers-of-two sub-boundaries, also include the nibble boundaries of 14, 15, and 16, and the byte boundaries of 254, 255, and 256.

ASCII TABLE

Another common sub-boundary condition is the ASCII character table. Table 5.2 is a partial listing of the ASCII table.

TABLE 5.2. A PARTIAL ASCII TABLE OF VALUES

Character	ASCII Value	Character	ASCII Value
Null	0	B	66
Space	32	Y	89
/	47	Z	90
0	48	[	91
1	49	'	96
2	50	a	97
9	57	b	98
:	58	y	121
@	64	z	122
A	65	{	123

Notice that Table 5.2 is not a nice, contiguous list. 0 through 9 are assigned to ASCII values 48 through 57. The slash character, /, falls before 0. The colon, :, comes after 9. The uppercase letters A through Z go from 65 to 90. The lowercase letters span 97 to 122. All these cases represent sub-boundary conditions. If you're testing software that performs text entry or text conversion, you'd be very wise to reference a copy of the ASCII table and consider its boundary conditions when you define what values to include in your data partitions. For example, if you are testing a text box that accepts only the characters AZ and az, you should include in your invalid partition the values just below and above those in the ASCII table@, [, ', and {.

ASCII AND UNICODE

Although ASCII is still very popular as the common means for software to represent character data, it's being replaced by a new standard called Unicode. Unicode was developed by the Unicode Consortium in 1991 to solve ASCII's problem of not being able to represent all characters in all written languages.

ASCII, using only 8 bits, can represent only 256 different characters. Unicode, which uses 16 bits, can represent 65,536 characters. To date, more than 39,000 characters have been assigned, with more than 21,000 being used for Chinese ideographs.

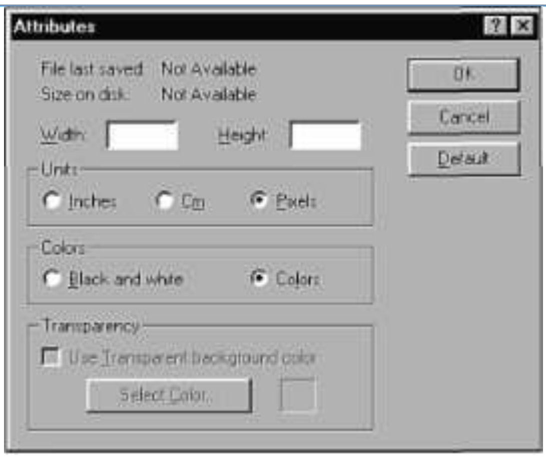
DEFAULT, EMPTY, BLANK, NULL, ZERO, AND NONE

Another source of bugs that may seem obvious is when the software requests an entry - say, in a text box - but rather than type the correct information, the user types nothing. He may just press Enter. This situation is often overlooked in the specification or forgotten by the programmer but is a case that typically happens in real life.

Well-behaved software will handle this situation. It will usually default to the lowest valid boundary limit or to some reasonable value in the middle of the valid partition, or return an error.

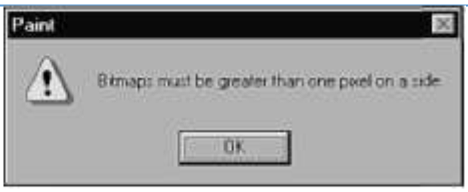
The Windows Paint Attributes dialog box (see Figure 5.6) normally places default values in the Width and Height text fields. If the user accidentally or purposely deletes them so that the fields are blank and then clicks OK, what happens?

FIGURE 5.6. THE WINDOWS PAINT ATTRIBUTES DIALOG BOX WITH THE WIDTH AND HEIGHT TEXT FIELDS BLANKED OUT.



Ideally, the software would handle this by defaulting to some valid width and height. If it didn't do that, some error should be returned, which is exactly what you get (see Figure 5.7). The error "Bitmaps must be greater than one pixel on a side" isn't the most descriptive one ever written, but that's another topic.

FIGURE 5.7. THE ERROR MESSAGE RETURNED IF ENTER IS PRESSED WITH THE WIDTH AND HEIGHT TEXT FIELDS BLANKED OUT.



Bitmaps must be greater than one pixel on a side

TIP
Always consider creating an equivalence partition that handles the default, empty, blank, null, zero, or none conditions.

You should create a separate equivalence partition for these values rather than lump them into the valid cases or the invalid cases because the software usually handles them differently. It's likely that in this default case, a different software path is followed than if the user typed 0 or 1 as invalid values. Since you expect different operation of the software, they should be in their own partition.

INVALID, WRONG, INCORRECT, AND GARBAGE DATA

The final type of data testing is garbage data. This is where you test-to-fail. You've already proven that the software works as it should by testing-to-pass with boundary testing, sub-boundary testing, and default testing. Now it's time to throw the trash at it.
Software testing purists might argue that this isn't necessary, that if you've tested everything discussed so far you've proven the software will work. In the real world, however, there's nothing wrong with seeing if the software will handle whatever a user can do to it.

If you consider that software today can sell hundreds of millions of copies, it's conceivable that some percentage of the users will use the software incorrectly. If that results in a crash or data loss, users won't blame themselves - they will blame the software. If the software doesn't do what they expect, it has a bug. Period.
So, with invalid, wrong, incorrect, and garbage data testing, have some fun. If the software wants numbers, give it letters. If it accepts only positive numbers, enter negative numbers. If it's date sensitive, see if it'll work correctly on the year 3000. Pretend to have "fat fingers" and press multiple keys at a time. There are no real rules for this testing other than to try to break the software. Be creative. Be devious. Have fun.

STATE TESTING

So far what you've been testing is the data - the numbers, words, inputs, and outputs of the software. The other side of software testing is to verify the program's logic flow through its various states. A software state is a condition or mode that the software is currently in. Consider Figures 5.8 and 5.9.

FIGURE 5.8. THE WINDOWS PAINT PROGRAM IN THE PENCIL DRAWING STATE.

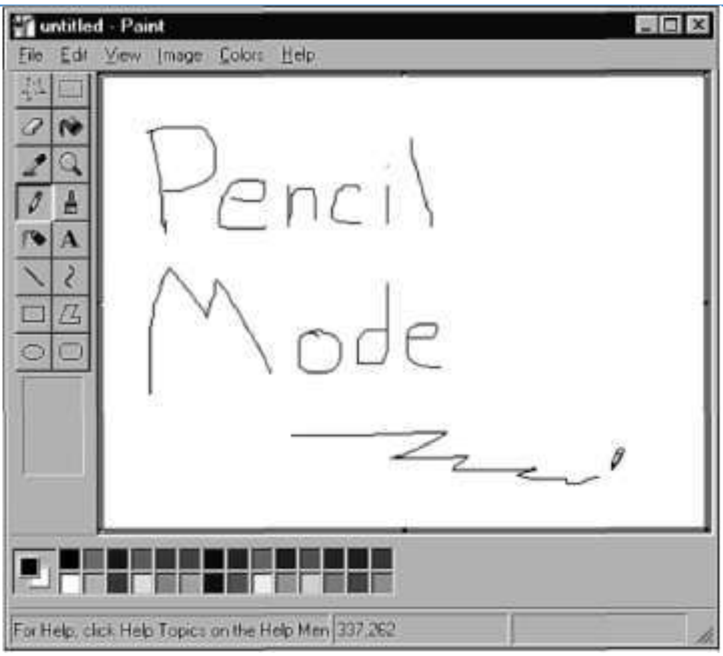

untitled - Paint
File Edit View Image Colors Help
Pencil
Mode
For Help, click Help Topics on the Help Menu
337,262

FIGURE 5.9. THE WINDOWS PAINT PROGRAM IN THE AIRBRUSHING STATE.

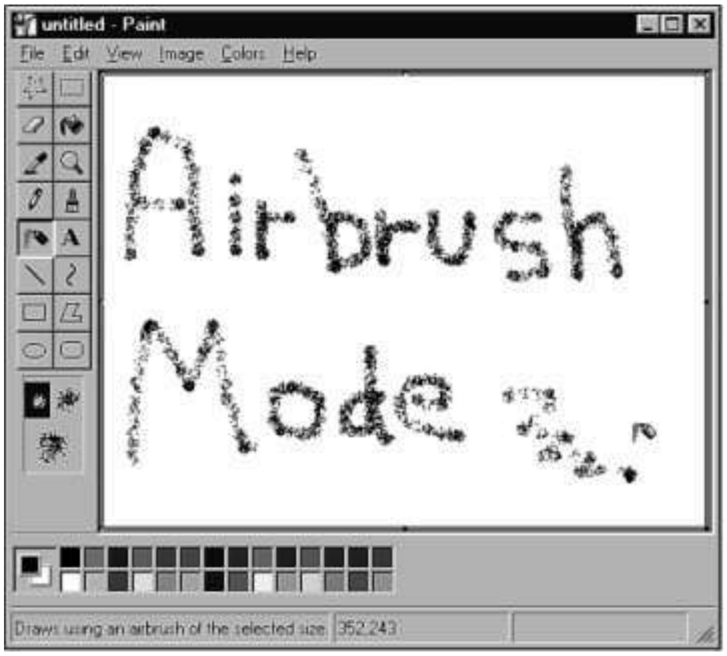

untitled - Paint
File Edit View Image Colors Help
Airbrush Mode
Draws using an airbrush of the selected size
352,243

Figure 5.8 shows the Windows Paint program in the pencil drawing state. This is the initial state in which the software starts. Notice that the pencil tool is selected, the cursor looks like a pencil, and a fine line is used to draw onscreen. Figure 5.9 shows the same program in the airbrush state. In this state, the airbrush tool is selected, airbrush sizes are provided, the cursor looks like a spray-paint can, and drawing results in a spray-paint look.

Take a closer look at all the available options that Paint provides - all the tools, menu items, colors, and so on. Whenever you select one of these and make the software change its look, its menus, or its operation, you're changing its state. The software follows a path through the code, toggles some bits, sets some variables, loads some data, and arrives at a new state of being.

NOTE

A software tester must test a program's states and the transitions between them.

TESTING THE SOFTWARE'S LOGIC FLOW

Remember the example in Chapter 3 that showed the infinite data possibilities for testing the Windows Calculator? You learned earlier in this chapter that to make the testing manageable, you must reduce the data possibilities by creating equivalence partitions of only the most vital numbers.
Testing the software's states and logic flow has the same problems. It's usually possible to visit all the states (after all, if you can't get to them, why have them?). The difficulty is that except for the simplest programs, it's often impossible to traverse all paths to all states. The complexity of the software, especially due to the richness of today's user interfaces, provides so many choices and options that the number of paths grows exponentially.
The problem is similar to the well-known traveling salesman problem: Given a fixed number of cities and the distance between each pair of them, find the shortest route to visit all of them once, returning to your starting point. If there were only five cities, you could do some quick math and discover that there are 120 different routes. Traversing each of them and finding the shortest route to all wouldn't be that

difficult or take that much time. If you increase that to hundreds or thousands of cities - or, in our case, hundreds or thousands of software states - you soon have a difficult-to-solve problem.

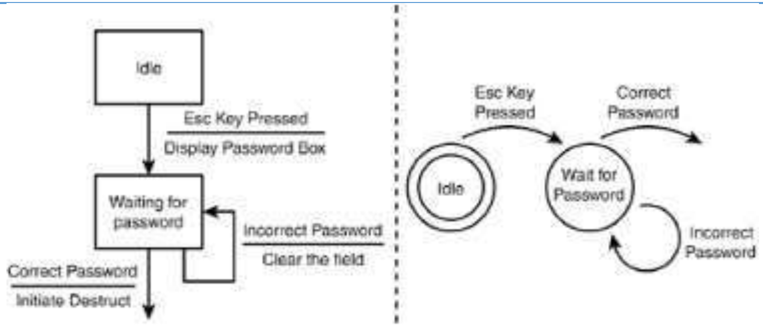
The solution for software testing is to apply equivalence partition techniques to the selection of the states and paths, assuming some risk because you will choose not to test all of them, but reducing that risk by making intelligent choices.

CREATING A STATE TRANSITION MAP

The first step is to create your own state transition map of the software. Such a map may be provided as part of the product specification. If it is, you should statically test it as described in Chapter 4, "Examining the Specification." If you don't have a state map, you'll need to create one.

There are several different diagramming techniques for state transition diagrams. Figure 5.10 shows two examples. One uses boxes and arrows and the other uses circles (bubbles) and arrows. The technique you use to draw your map isn't important as long as you and the other members of your project team can read and understand it.

FIGURE 5.10. STATE TRANSITION DIAGRAMS CAN BE DRAWN BY USING DIFFERENT TECHNIQUES.



NOTE
State transition diagrams can become quite large. Many development teams cover their office walls with the printouts. If you expect that your diagrams will become that complex, look for commercial software that helps you draw and manage them.

A state transition map should show the following items:

- Each unique state that the software can be in. A good rule of thumb is that if you're unsure whether something is a separate state, it probably is. You can always collapse it into another state if you find out later that it isn't.
- The input or condition that takes it from one state to the next. This might be a key press, a menu selection, a sensor input, a telephone ring, and so on. A state can't be exited without some reason. The specific reason is what you're looking for here.
- Set conditions and produced output when a state is entered or exited. This would include a menu and buttons being displayed, a flag being set, a printout occurring, a calculation being performed, and so on. It's anything and everything that happens on the transition from one state to the next.

REMINDER
Because you are performing black-box testing, you don't need to know what low-level variables are being set in the code. Create your map from the user's view of the software.

REDUCING THE NUMBER OF STATES AND TRANSITIONS TO TEST

Creating a map for a large software product is a huge undertaking. Hopefully, you'll be testing only a portion of the overall software so that making the map is a more reasonable task. Once you complete the map, you'll be able to stand back and see all the states and all the ways to and from those states. If you've done your job right, it'll be a scary picture!
If you had infinite time, you would want to test every path through the software - not just each line connecting two states, but each set of lines, back to front, round and round. As in the traveling salesman problem, it would be impossible to hit them all.
Just as you learned with equivalence partitioning for data, you need to reduce the huge set of possibilities to a set of test cases of workable size. There are five ways to do this:

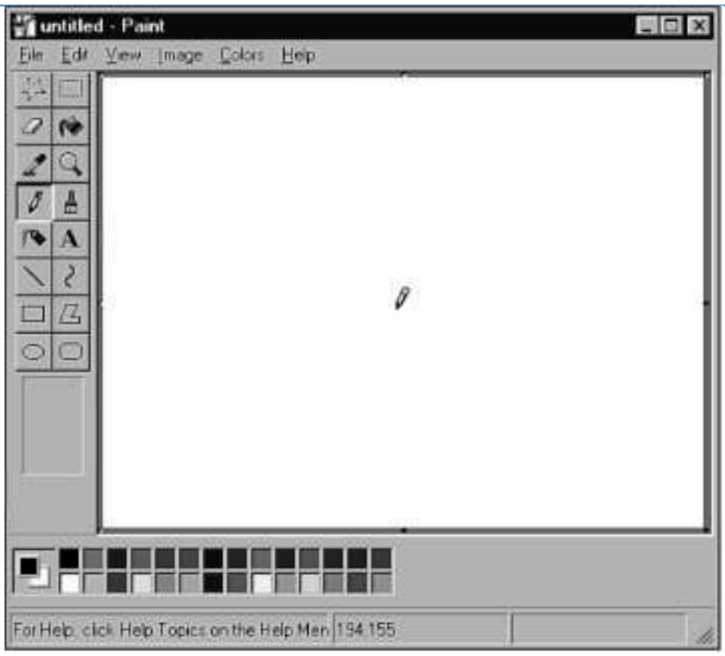
- Visit each state at least once. It doesn't matter how you get there, but each state needs to be tested.
- Test the state-to-state transitions that look like the most common or popular. This sounds subjective, and it is, but it should be based on the knowledge you gained when you performed static black-box analysis (in Chapter 3) of the product specification. Some user scenarios will be more frequently used than others. You want those to work!

- Test the least common paths between states. It's likely that these paths were overlooked by the product designers and the programmers. You may be the first one to try them.
- Test all the error states and returning from the error states. Many times error conditions are difficult to create. Very often programmers write the code to handle specific errors but can't test the code themselves. There are often cases when errors aren't properly handled, when the error messages are incorrect, or when the software doesn't recover properly when the error is fixed.
- Test random state transitions. If you have a printed state map, throw darts at it and try to move from dart to dart. If you have time to do more, read Chapter 15, "Automated Testing and Test Tools," for information on how to automate your random state transition testing.

WHAT TO SPECIFICALLY TEST

After you identify the specific states and state transitions that you want to test, you can begin defining your test cases.
Testing states and state transitions involves checking all the state variables - the static conditions, information, values, functionality, and so on that are associated with being in that state or moving to and from that state. Figure 5.11 shows an example of Windows Paint in the startup state.

FIGURE 5.11. THE WINDOWS PAINT OPENING SCREEN IN THE STARTUP STATE.



Here's a partial list of the state variables that define Paint's startup state:

- The window looks as shown in Figure 5.11.
- The window size is set to what it was the last time Paint was used.
- The drawing area is blank.
- The tool box, color box, and status bar are displayed.
- The pencil tool is selected. All the others are not.
- The default colors are black foreground on a white background.
- The document name is untitled.

There are many, many more state variables to consider for Paint, but these should give you an idea of what's involved in defining a state. Keep in mind that the same process of identifying state conditions is used whether the state is something visible such as a window or a dialog box, or invisible such as one that's part of a communications program or a financial package.

It's a good idea to discuss your assumptions about the states and state transitions with your team's spec writers and programmers. They can offer insights into states that happen behind the scenes that you may not have considered.

THE DIRTY DOCUMENT FLAG

State variables can be invisible but very important. A common example is the dirty document flag.
When a document is loaded into an editor, such as a word processor or painting program, an internal state variable called the dirty document flag is cleared and the software is in the "clean" state. The software stays in this state as long as no changes are made to the document. It can be viewed and scrolled and the state stays the same. As soon as something is typed or the document is modified in some way, the software changes state to the "dirty" state.
If an attempt is made to close or exit the software in the clean state, it shuts down normally. If the document is dirty, users will get a message asking if they want to save their work before quitting.
Some software is so sophisticated that if an edit is made that dirties the document and then the edit is undone to restore the document to its original condition, the software is returned to the clean state. Exiting the program will occur without a prompt to save the document.

TESTING STATES TO FAIL

Everything discussed so far regarding state testing has been about testing-to-pass. You're reviewing the software, sketching out the states, trying many valid possibilities, and making sure the states and state transitions work. The flip side to this, just as in data testing, is to find test cases that test the software to fail. Examples of such cases are race conditions, repetition, stress, and load.

RACE CONDITIONS AND BAD TIMING

Most operating systems today, whether for personal computers or for specialized equipment, can do multitasking. Multitasking means that an operating system is designed to run separate processes concurrently. These processes can be separate programs such as a spreadsheet and email. Or they can be part of the same program such as printing in the background while allowing new words to be typed into a word processor.
Designing a multitasking operating system isn't a trivial exercise, and designing applications software to take advantage of multitasking is a difficult task. In a truly multitasking environment, the software can't take anything for granted. It must handle being interrupted at any moment, be able to run concurrently with everything else on the system, and share resources such as memory, disk, communications, and other hardware.
The results of all this are race condition problems. These are when two or more events line up just right and confuse software that didn't expect to be interrupted in the middle of its operation. In other words, it's bad timing. The term race condition comes from just what you'd think - multiple processes racing to a finish line, not knowing which will get there first.
NOTE
Race condition testing is difficult to plan for, but you can get a good start by looking at each state in your state transition map and thinking about what outside influences might interrupt that state. Consider what the state might do if the data it uses isn't ready or is changing when it's needed. What if two or more of the connecting arcs or lines occur at exactly the same time?

Here are a few examples of situations that might expose race conditions:

- Saving and loading the same document at the same time with two different programs
- Sharing the same printer, communications port, or other peripheral

- Pressing keys or sending mouse clicks while the software is loading or changing states
- Shutting down or starting up two or more instances of the software at the same time
- Using different programs to simultaneously access a common database

These may sound like harsh tests, but they aren't, and the user often causes them by accident. Software must be robust enough to handle these situations. Years ago they may have been out of the ordinary but today, users expect their software to work properly under these conditions.

REPETITION, STRESS, AND LOAD

Three other test-to-fail state tests are repetition, stress, and load. These tests target state handling problems where the programmer didn't consider what might happen in the worst-case scenarios.
Repetition testing involves doing the same operation over and over. This could be as simple as starting up and shutting down the program over and over. It could also mean repeatedly saving and loading data or repeatedly selecting the same operation. You might find a bug after only a couple repetitions or it might take thousands of attempts to reveal a problem.
The main reason for doing repetition testing is to look for memory leaks. A common software problem happens when computer memory is allocated to perform a certain operation but isn't completely freed when the operation completes. The result is that eventually the program uses up memory that it depends on to work reliably. If you've ever used a program that works fine when you first start it up, but then becomes slower and slower or starts to behave erratically over time, it's likely due to a memory leak bug. Repetition testing will flush these problems out.
Stress testing is running the software under less-than-ideal conditions - low memory, low disk space, slow CPUs, slow modems, and so on. Look at your software and determine what external resources and dependencies it has. Stress testing is simply limiting them to their bare minimum. Your goal is to starve the software. Does this sound like boundary condition testing? It is.
Load testing is the opposite of stress testing. With stress testing, you starve the software; with load testing, you feed it all that it can handle. Operate the software with the largest possible data files. If the software operates on peripherals such as printers or communications ports, connect as many as you can. If you're testing an Internet server that can handle thousands of simultaneous connections, do it. Max out the software's capabilities. Load it down.
Don't forget about time as a load testing variable. With most software, it's important for it to run over long periods. Some software should be able to run forever without being restarted.
NOTE
There's no reason that you can't combine repetition, stress, and load, running all the tests at the same time. This is a sure way to expose severe bugs that might otherwise be difficult to find.

There are two important considerations with repetition, stress, and load testing:

- Your team's programmers and project managers may not be completely receptive to your efforts to break the software this way. You'll probably hear them complain that no customer will use the system this way or stress it to the point that you are. The short answer is that yes, they will. Your job is to make sure that the software does work in these situations and to report bugs if it doesn't. Chapter 19, "Reporting What You Find," discusses how to best report your bugs to make sure that they're taken seriously and are fixed.
- Opening and closing your program a million times is probably not possible if you're doing it by hand. Likewise, finding a few thousand people to connect to your Internet server might be difficult to organize. Chapter 15 covers test automation and will give you ideas on how to perform testing such as this without requiring people to do the dirty work.

OTHER BLACK-BOX TEST TECHNIQUES

The remaining categories of black-box test techniques aren't standalone methods as much as they are variations of the data testing and state testing that has already been described. If you've done thorough

equivalence partitioning of your program's data, created a detailed state map, and developed test cases from these, you'll find most software bugs that a user would find.
What's left are techniques for finding the stragglers, the ones that, if they were real living bugs, might appear to have a mind of their own, going their own way. Finding them might appear a bit subjective and not necessarily based on reason, but if you want to flush out every last bug, you'll have to be a bit creative.

BEHAVE LIKE A DUMB USER

The politically correct term might be inexperienced user or new user, but in reality, they're all the same thing. Put a person who's unfamiliar with the software in front of your program and they'll do and try things that you never imagined. They'll enter data that you never thought of. They'll change their mind in mid-stream, back up, and do something different. They'll surf through your website, clicking things that shouldn't be clicked. They'll discover bugs that you completely missed.
It can be frustrating, as a tester, to watch someone who has no experience in testing spend five minutes using a piece of software and crash it. How do they do it? They weren't operating on any rules or making any assumptions.
When you're designing your test cases or looking at the software for the first time, try to think like a dumb user. Throw out any preconceived ideas you had about how the software should work. If you can, bring in a friend who isn't working on the project to brainstorm ideas with you. Assume nothing. Adding these test cases to your designed library of test cases will create a very comprehensive set.

LOOK FOR BUGS WHERE YOU'VE ALREADY FOUND THEM

There are two reasons to look for bugs in the areas where you've already found them:

- As you learned in Chapter 3, the more bugs you find, the more bugs there are. If you discover that you're finding lots of bugs at the upper boundary conditions across various features, it would be wise to emphasize testing these upper boundaries on all features. Of course you're going to test these anyway, but you might want to throw in a few special cases to make sure the problem isn't pervasive.
- Many programmers tend to fix only the specific bug you report. No more, no less. If you report a bug that starting, stopping, and restarting a program 255 times results in a crash, that's what the programmer will fix. There may have been a memory leak that caused the problem and the programmer found and fixed it. When you get the software back to retest, make sure you rerun the same test for 256 times and beyond. There could very well be yet another memory leak somewhere out there.

THINK LIKE A HACKER

As you'll learn in Chapter 13, no software is 100% secure. The hackers of the world know this and will seek to find vulnerabilities in your software and exploit them. As a tester, you'll need to be devious and conniving. Try to think of what things of value your software contains, why someone might want to gain access to them, and what the methods of entry might be for them to get in. Don't be gentle. They won't be.

FOLLOW EXPERIENCE, INTUITION, AND HUNCHES

There's no better way to improve as a software tester than to gain experience. There's no better learning tool than just doing it, and there's no better lesson than getting that first phone call from a customer who found a bug in the software you just finished testing.
Experience and intuition can't be taught. They must be gained over time. You can apply all the techniques you've learned so far and still miss important bugs. It's the nature of the business. As you progress through your career, learning to test different types and sizes of products, you'll pick up little tips and tricks that steer you toward those tough-to-find bugs. You'll be able to start testing a new piece of software and quickly find bugs that your peers would have missed.

Take notes of what works and what doesn't. Try different approaches. If you think something looks suspicious, take a closer look. Go with your hunches until you prove them false.

> Experience is the name everyone gives to their mistakes.
>
> Oscar Wilde

SUMMARY

It's been a long chapter. Dynamic black-box testing covers a lot of ground. For new testers, this may be the single most important chapter in the book. It's likely that at your interviews or your first day on the job you'll be given software and asked to test it. Applying this chapter's techniques is a sure way to immediately find bugs.

Don't assume, though, that this is all there is to software testing. If it was, you could stop reading right now and ignore the remaining chapters. Dynamic black-box testing will just get you in the door. There's so much more to software testing, and you're just getting started.

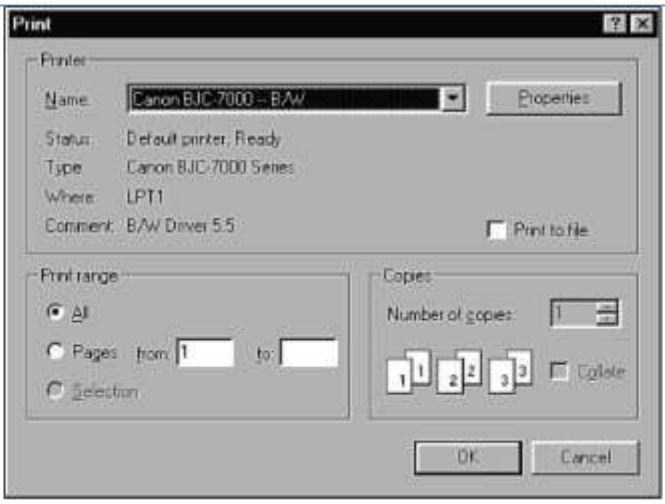
The next two chapters introduce you to software testing when you have access to the code and can see how it works and what it does on the lowest levels. The same black-box techniques are still valid, but you'll be able to complement them with new techniques that will help you become an even more effective software tester

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. True or False: You can perform dynamic black-box testing without a product specification or requirements document.
2. If you're testing a program's ability to print to a printer, what generic test-to-fail test cases might be appropriate?
3. Start up Windows WordPad and select Print from the File menu. You'll get the dialog shown in Figure 5.12. What boundary conditions exist for the Print Range feature shown in the lower-left corner?

FIGURE 5.12. THE WINDOWS PRINT DIALOG BOX SHOWING THE PRINT RANGE FEATURE.



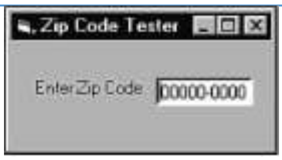
Printer
Name: Canon BJC-7000 - B/W
Status: Default printer; Ready
Type: Canon BJC-7000 Series
Where: LPT1
Comment: B/W Driver 5.5
Print to file [ ]

Print range
[✓ All]
Pages from: 1 to:
Selection

Copies
Number of copies: 1
Collate [ ]

4. Assume that you have a 10-character-wide ZIP code text box, such as the one shown in Figure 5.13. What equivalence partitions would you create for this text box?

FIGURE 5.13. A SAMPLE ZIP CODE TEXT BOX THAT HOLDS UP TO 10 CHARACTERS.



5. True or False: Visiting all the states that a program has assures that you've also traversed all the transitions among them.

6. There are many different ways to draw state transition diagrams, but there are three things that they all show. What are they?

7. What are some of the initial state variables for the Windows Calculator?

8. What actions do you perform on software when attempting to expose race condition bugs?

9.True or False: It's an unfair test to perform stress testing at the same time you perform load testing.

CHAPTER 6. EXAMINING THE CODE

IN THIS CHAPTER

- Static White-Box Testing: Examining the Design and Code
- Formal Reviews
- Coding Standards and Guidelines
- Generic Code Review Checklist

Software testing isn't limited to treating the specification or the program like a black box as described in Chapters 4, "Examining the Specification," and 5, "Testing the Software with Blinders On." If you have some programming experience, even if it's just a little, you can also perform testing on the software's architecture and code.
In some industries, such verification isn't as common as black-box testing. However, if you're testing military, financial, factory automation, or medical software, or if you're lucky enough to be working in a highly disciplined development model, it may be routine to verify the product at this level. If you're testing software for security issues, it's imperative.
This chapter introduces you to the basics of performing verification on the design and code. As a new software tester, it may not be your first task, but it's one that you can eventually move into if your interests lie in programming.
Highlights from this chapter include

- The benefits of static white-box testing
- The different types of static white-box reviews
- Coding guidelines and standards
- How to generically review code for errors

STATIC WHITE-BOX TESTING: EXAMINING THE DESIGN AND CODE

Remember the definitions of static testing and white-box testing from Chapter 4? Static testing refers to testing something that isn't running - examining and reviewing it.

White-box (or clear-box) testing implies having access to the code, being able to see it and review it. Static white-box testing is the process of carefully and methodically reviewing the software design, architecture, or code for bugs without executing it. It's sometimes referred to as structural analysis.
The obvious reason to perform static white-box testing is to find bugs early and to find bugs that would be difficult to uncover or isolate with dynamic black-box testing. Having a team of testers concentrate their efforts on the design of the software at this early stage of development is highly cost effective.
A side benefit of performing static white-box testing is that it gives the team's black-box testers ideas for test cases to apply when they receive the software for testing. They may not necessarily understand the details of the code, but by listening to the review comments they can identify feature areas that sound troublesome or bug-prone.
NOTE
Development teams vary in who has the responsibility for static white-box testing. In some teams the programmers are the ones who organize and run the reviews, inviting the software testers as independent observers. In other teams the software testers are the ones who perform this task, asking the programmer who wrote the code and a couple of his peers to assist in the reviews. Ultimately, either approach can work. It's up to the development team to choose what works best for them.

The unfortunate thing about static white-box testing is that it's not always done. Many teams have the misconception that it's too time-consuming, too costly, or not productive. All of these are untrue - compared to the alternative of testing, finding, and even not finding bugs at the back end of the project. The problem lies in the perception that a programmer's job is to write lines of code and that any task that takes away from his efficiency of churning out those lines is slowing down the process.
Fortunately, the tide is changing. Many companies are realizing the benefits of testing early and are hiring and training their programmers and testers to perform white-box testing. It's not rocket science (unless you're designing rockets), but getting started requires knowing a few basic techniques. If you're interested in taking it further, the opportunities are huge.

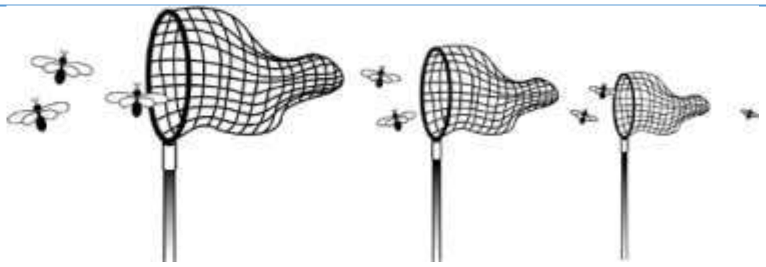
FORMAL REVIEWS

A formal review is the process under which static white-box testing is performed. A formal review can range from a simple meeting between two programmers to a detailed, rigorous inspection of the software's design or its code.
There are four essential elements to a formal review:

- Identify Problems. The goal of the review is to find problems with the software - not just items that are wrong, but missing items as well. All criticism should be directed at the design or code, not the person who created it. Participants shouldn't take any criticism personally. Leave your egos, emotions, and sensitive feelings at the door.
- Follow Rules. A fixed set of rules should be followed. They may set the amount of code to be reviewed (usually a couple hundred lines), how much time will be spent (a couple hours), what can be commented on, and so on. This is important so that the participants know what their roles are and what they should expect. It helps the review run more smoothly.
- Prepare. Each participant is expected to prepare for and contribute to the review. Depending on the type of review, participants may have different roles. They need to know what their duties and responsibilities are and be ready to actively fulfill them at the review. Most of the problems found through the review process are found during preparation, not at the actual review.
- Write a Report. The review group must produce a written report summarizing the results of the review and make that report available to the rest of the product development team. It's imperative that others are told the results of the meeting - how many problems were found, where they were found, and so on.

What makes formal reviews work is following an established process. Haphazardly "getting together to go over some code" isn't sufficient and may actually be detrimental. If a process is run in an ad-hoc fashion, bugs will be missed and the participants will likely feel that the effort was a waste of time.
If the reviews are run properly, they can prove to be a great way to find bugs early. Think of them as one of the initial nets (see Figure 6.1) that catches the big bugs at the beginning of the process. Sure, smaller bugs will still get through, but they'll be caught in the next testing phases with the smaller nets with the tighter weave.

FIGURE 6.1. FORMAL REVIEWS ARE THE FIRST NETS USED IN CATCHING BUGS.



In addition to finding problems, holding formal reviews has a few indirect results:

- Communications. Information not contained in the formal report is communicated. For example, the black-box testers can get insight into where problems may lie. Inexperienced programmers may learn new techniques from more experienced programmers. Management may get a better feel for how the project is tracking its schedule.
- Quality. A programmer's code that is being gone over in detail, function by function, line by line, often results in the programmer being more careful. That's not to say that he would otherwise be sloppy - just that if he knows that his work is being carefully reviewed by his peers, he might make an extra effort to triple-check it to make sure that it's right.
- Team Camaraderie. If a review is run properly, it can be a good place for testers and programmers to build respect for each other's skills and to better understand each other's jobs and job needs.
- Solutions. Solutions may be found for tough problems, although whether they are discussed depends on the rules for the review. It may be more effective to discuss solutions outside the review.

These indirect benefits shouldn't be relied on, but they do happen. On many teams, for whatever reasons, the members end up working in isolation. Formal reviews are a great way to get them in the same room, all discussing the same project problems.

PEER REVIEWS

The easiest way to get team members together and doing their first formal reviews of the software is through peer reviews, the least formal method. Sometimes called buddy reviews, this method is really more of an "I'll show you mine if you show me yours" type discussion.
Peer reviews are often held with just the programmer who designed the architecture or wrote the code and one or two other programmers or testers acting as reviewers. That small group simply reviews the code together and looks for problems and oversights. To assure that the review is highly effective (and doesn't turn into a coffee break) all the participants need to make sure that the four key elements of a formal review are in place: Look for problems, follow rules, prepare for the review, and write a report. Because peer reviews are informal, these elements are often scaled back. Still, just getting together to discuss the code can find bugs.

WALKTHROUGHS

Walkthroughs are the next step up in formality from peer reviews. In a walkthrough, the programmer who wrote the code formally presents (walks through) it to a small group of five or so other programmers and testers. The reviewers should receive copies of the software in advance of the review so they can examine it and write comments and questions that they want to ask at the review. Having at least one senior programmer as a reviewer is very important.

The presenter reads through the code line by line, or function by function, explaining what the code does and why. The reviewers listen and question anything that looks suspicious. Because of the larger number of participants involved in a walkthrough compared to a peer review, it's much more important for them to prepare for the review and to follow the rules. It's also very important that after the review the presenter write a report telling what was found and how he plans to address any bugs discovered.

INSPECTIONS

Inspections are the most formal type of reviews. They are highly structured and require training for each participant. Inspections are different from peer reviews and walkthroughs in that the person who presents the code, the presenter or reader, isn't the original programmer. This forces someone else to learn and understand the material being presented, potentially giving a different slant and interpretation at the inspection meeting.

The other participants are called inspectors. Each is tasked with reviewing the code from a different perspective, such as a user, a tester, or a product support person. This helps bring different views of the product under review and very often identifies different bugs. One inspector is even tasked with reviewing the code backward - that is, from the end to the beginning - to make sure that the material is covered evenly and completely.

Some inspectors are also assigned tasks such as moderator and recorder to assure that the rules are followed and that the review is run effectively.

After the inspection meeting is held, the inspectors might meet again to discuss the defects they found and to work with the moderator to prepare a written report that identifies the rework necessary to address the problems. The programmer then makes the changes and the moderator verifies that they were properly made. Depending on the scope and magnitude of the changes and on how critical the software is, a re-inspection may be needed to locate any remaining bugs.

Inspections have proven to be very effective in finding bugs in any software deliverable, especially design documents and code, and are gaining popularity as companies and product development teams discover their benefits.

CODING STANDARDS AND GUIDELINES

In formal reviews, the inspectors are looking for problems and omissions in the code. There are the classic bugs where something just won't work as written. These are best found by careful analysis of the code - senior programmers and testers are great at this.

There are also problems where the code may operate properly but may not be written to meet a specific standard or guideline. It's equivalent to writing words that can be understood and get a point across but don't meet the grammatical and syntactical rules of the English language. Standards are the established, fixed, have-to-follow-them rules - the do's and don'ts. Guidelines are the suggested best practices, the recommendations, the preferred way of doing things. Standards have no exceptions, short of a structured waiver process. Guidelines can be a bit loose.

It may sound strange that some piece of software may work, may even be tested and shown to be very stable, but still be incorrect because it doesn't meet some criteria. It's important, though, and there are three reasons for adherence to a standard or guideline:

- Reliability. It's been shown that code written to a specific standard or guideline is more reliable and secure than code that isn't.
- Readability/Maintainability. Code that follows set standards and guidelines is easier to read, understand, and maintain.

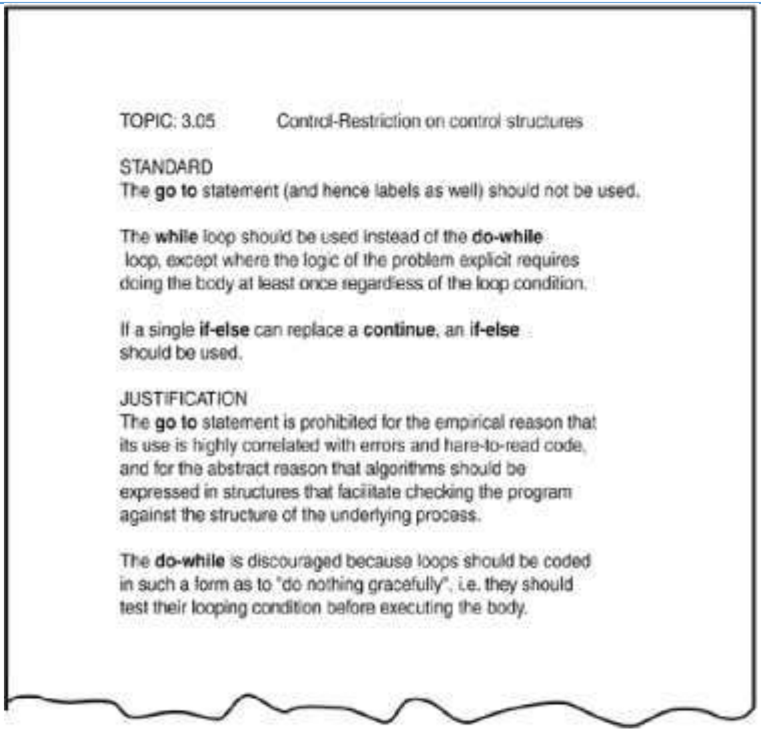
- Portability. Code often has to run on different hardware or be compiled with different compilers. If it follows a set standard, it will likely be easier - or even completely painless - to move it to a different platform.

The requirements for your project may range from strict adherence to national or international standards to loose following of internal team guidelines. What's important is that your team has some standards or guidelines for programming and that these are verified in a formal review.

EXAMPLES OF PROGRAMMING STANDARDS AND GUIDELINES

Figure 6.2 shows an example of a programming standard that deals with the use of the C language goto, while, and if-else statements. Improper use of these statements often results in buggy code, and most programming standards explicitly set rules for using them.

FIGURE 6.2. A SAMPLE CODING STANDARD EXPLAINS HOW SEVERAL LANGUAGE CONTROL STRUCTURES SHOULD BE USED. (ADAPTED FROM C++ PROGRAMMING GUIDELINES BY THOMAS PLUM AND DAN SAKS. COPYRIGHT 1991, PLUM HALL, INC.)



TOPIC: 3.05 Control-Restriction on control structures

STANDARD
The **go to** statement (and hence labels as well) should not be used.

The **while** loop should be used instead of the **do-while** loop, except where the logic of the problem explicit requires doing the body at least once regardless of the loop condition.

If a single **if-else** can replace a **continue**, an **if-else** should be used.

JUSTIFICATION
The **go to** statement is prohibited for the empirical reason that its use is highly correlated with errors and hare-to-read code, and for the abstract reason that algorithms should be expressed in structures that facilitate checking the program against the structure of the underlying process.

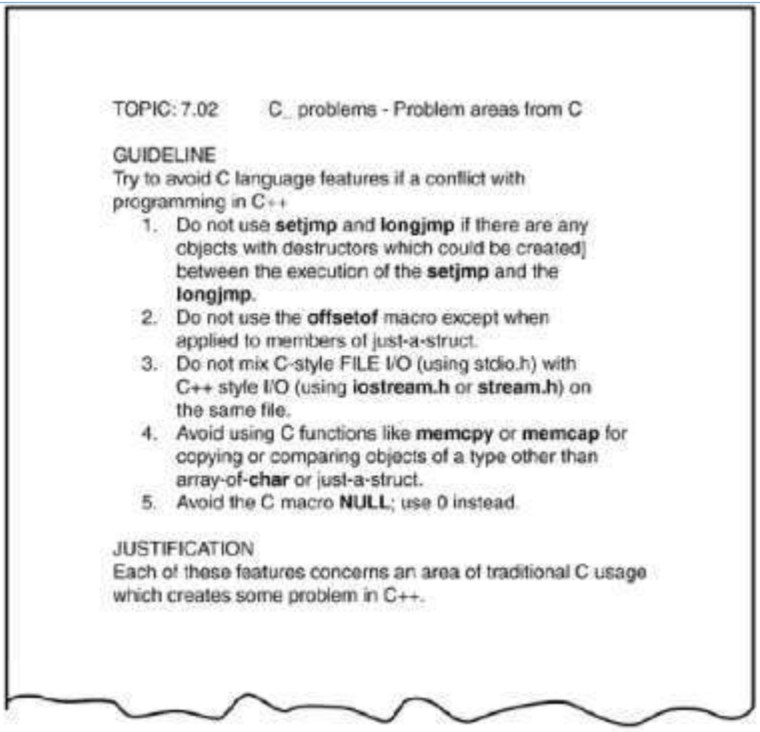
The **do-while** is discouraged because loops should be coded in such a form as to "do nothing gracefully", i.e. they should test their looping condition before executing the body.

The standard has four main parts:

- Title describes what topic the standard covers.
- Standard (or guideline) describes the standard or guideline explaining exactly what's allowed and not allowed.
- Justification gives the reasoning behind the standard so that the programmer understands why it's good programming practice.
- Example shows simple programming samples of how to use the standard. This isn't always necessary.

Figure 6.3 is an example of a guideline dealing with C language features used in C++. Note how the language is a bit different. In this case, it starts out with "Try to avoid." Guidelines aren't as strict as standards, so there is some room for flexibility if the situation calls for it.

FIGURE 6.3. AN EXAMPLE OF A PROGRAMMING GUIDELINE SHOWS HOW TO USE CERTAIN ASPECTS OF C IN C++. (ADAPTED FROM C++ PROGRAMMING GUIDELINES BY THOMAS PLUM AND DAN SAKS. COPYRIGHT 1991, PLUM HALL, INC.)



TOPIC: 7.02 C_ problems - Problem areas from C

GUIDELINE
Try to avoid C language features if a conflict with programming in C++

1. Do not use **setjmp** and **longjmp** if there are any objects with destructors which could be created between the execution of the **setjmp** and the **longjmp**.
2. Do not use the **offsetof** macro except when applied to members of just-a-struct.
3. Do not mix C-style FILE I/O (using stdio.h) with C++ style I/O (using **iostream.h** or **stream.h**) on the same file.
4. Avoid using C functions like **memcpy** or **memcap** for copying or comparing objects of a type other than array-of-**char** or just-a-struct.
5. Avoid the C macro **NULL**; use 0 instead.

JUSTIFICATION
Each of these features concerns an area of traditional C usage which creates some problem in C++.

IT'S A MATTER OF STYLE

There are standards, there are guidelines, and then there is style. From a software quality and testing perspective, style doesn't matter.

Every programmer, just like every book author and artist, has his or her own unique style. The rules may be followed, the language usage may be consistent, but it's still easy to tell who created what software.

That differentiating factor is style. In programming, it could be how verbose the commenting is or how the variables are named. It could be what indentation scheme is used in the loop constructs. It's the look and feel of the code.

Some teams do institute standards and guidelines for style aspects (such as indenting) so that the look and feel of the code doesn't become too random. As a software tester, when performing formal reviews on a piece of software, test and comment only on things that are wrong, missing, or don't adhere to the team's accepted standards or guidelines. Ask yourself if what you're about to report is really a problem or just difference of opinion, a difference of style. The latter isn't a bug.

OBTAINING STANDARDS

If your project, because of its nature, must follow a set of programming standards, or if you're just interested in examining your software's code to see how well it meets a published standard or guideline, several sources are available for you to reference.

National and international standards for most computer languages and information technology can be obtained from:

- American National Standards Institute (ANSI), www.ansi.org
- International Engineering Consortium (IEC), www.iec.org
- International Organization for Standardization (ISO), www.iso.ch
- National Committee for Information Technology Standards (NCITS), www.ncits.org

There are also documents that demonstrate programming guidelines and best practices available from professional organizations such as

- Association for Computing Machinery (ACM), www.acm.org
- Institute of Electrical and Electronics Engineers, Inc (IEEE), www.ieee.org

You may also obtain information from the software vendor where you purchased your programming software. They often have published standards and guidelines available for free or for a small fee.

GENERIC CODE REVIEW CHECKLIST

The rest of this chapter on static white-box testing covers some problems you should look for when verifying software for a formal code review. These checklists[1] are in addition to comparing the code against a standard or a guideline and to making sure that the code meets the project's design requirements.

[1] These checklist items were adapted from Software Testing in the Real World: Improving the Process, pp. 198-201. Copyright 1995 by Edward Kit. Used by permission of Pearson Education Limited, London. All rights reserved.

To really understand and apply these checks, you should have some programming experience. If you haven't done much programming, you might find it useful to read an introductory book such as Sams Teach Yourself Beginning Programming in 24 Hours by Sams Publishing before you attempt to review program code in detail.

DATA REFERENCE ERRORS

Data reference errors are bugs caused by using a variable, constant, array, string, or record that hasn't been properly declared or initialized for how it's being used and referenced.

- Is an uninitialized variable referenced? Looking for omissions is just as important as looking for errors.
- Are array and string subscripts integer values and are they always within the bounds of the array's or string's dimension?
- Are there any potential "off by one" errors in indexing operations or subscript references to arrays? Remember the code in Listing 5.1 from Chapter 5.
- Is a variable used where a constant would actually work better - for example, when checking the boundary of an array?
- Is a variable ever assigned a value that's of a different type than the variable? For example, does the code accidentally assign a floating-point number to an integer variable?
- Is memory allocated for referenced pointers?
- If a data structure is referenced in multiple functions or subroutines, is the structure defined identically in each one?

NOTE

Data reference errors are the primary cause of buffer overruns - the bug behind many software security issues. Chapter 13, "Testing for Software Security," covers this topic in more detail.

DATA DECLARATION ERRORS

Data declaration bugs are caused by improperly declaring or using variables or constants.

- Are all the variables assigned the correct length, type, and storage class? For example, should a variable be declared as a string instead of an array of characters?
- If a variable is initialized at the same time as it's declared, is it properly initialized and consistent with its type?
- Are there any variables with similar names? This isn't necessarily a bug, but it could be a sign that the names have been confused with those from somewhere else in the program.
- Are any variables declared that are never referenced or are referenced only once?
- Are all the variables explicitly declared within their specific module? If not, is it understood that the variable is shared with the next higher module?

COMPUTATION ERRORS

Computational or calculation errors are essentially bad math. The calculations don't result in the expected result.

- Do any calculations that use variables have different data types, such as adding an integer to a floating-point number?
- Do any calculations that use variables have the same data type but are different lengths - adding a byte to a word, for example?
- Are the compiler's conversion rules for variables of inconsistent type or length understood and taken into account in any calculations?
- Is the target variable of an assignment smaller than the right-hand expression?
- Is overflow or underflow in the middle of a numeric calculation possible?
- Is it ever possible for a divisor/modulus to be zero?
- For cases of integer arithmetic, does the code handle that some calculations, particularly division, will result in loss of precision?
- Can a variable's value go outside its meaningful range? For example, could the result of a probability be less than 0% or greater than 100%?
- For expressions containing multiple operators, is there any confusion about the order of evaluation and is operator precedence correct? Are parentheses needed for clarification?

COMPARISON ERRORS

Less than, greater than, equal, not equal, true, false. Comparison and decision errors are very susceptible to boundary condition problems.

- Are the comparisons correct? It may sound pretty simple, but there's always confusion over whether a comparison should be less than or less than or equal to.
- Are there comparisons between fractional or floating-point values? If so, will any precision problems affect their comparison? Is 1.00000001 close enough to 1.00000002 to be equal?
- Does each Boolean expression state what it should state? Does the Boolean calculation work as expected? Is there any doubt about the order of evaluation?
- Are the operands of a Boolean operator Boolean? For example, is an integer variable containing integer values being used in a Boolean calculation?

CONTROL FLOW ERRORS

Control flow errors are the result of loops and other control constructs in the language not behaving as expected. They are usually caused, directly or indirectly, by computational or comparison errors.

- If the language contains statement groups such as begin...end and do...while, are the ends explicit and do they match their appropriate groups?
- Will the program, module, subroutine, or loop eventually terminate? If it won't, is that acceptable?
- Is there a possibility of premature loop exit?
- Is it possible that a loop never executes? Is it acceptable if it doesn't?
- If the program contains a multiway branch such as a switch...case statement, can the index variable ever exceed the number of branch possibilities? If it does, is this case handled properly?
- Are there any "off by one" errors that would cause unexpected flow through the loop?

SUBROUTINE PARAMETER ERRORS

Subroutine parameter errors are due to incorrect passing of data to and from software subroutines.

- Do the types and sizes of parameters received by a subroutine match those sent by the calling code? Is the order correct?
- If a subroutine has multiple entry points (yuck), is a parameter ever referenced that isn't associated with the current point of entry?
- If constants are ever passed as arguments, are they accidentally changed in the subroutine?
- Does a subroutine alter a parameter that's intended only as an input value?
- Do the units of each parameter match the units of each corresponding argument - English versus metric, for example?
- If global variables are present, do they have similar definitions and attributes in all referencing subroutines?

INPUT/OUTPUT ERRORS

These errors include anything related to reading from a file, accepting input from a keyboard or mouse, and writing to an output device such as a printer or screen. The items presented here are very simplified and generic. You should adapt and add to them to properly cover the software you're testing.

- Does the software strictly adhere to the specified format of the data being read or written by the external device?
- If the file or peripheral isn't present or ready, is that error condition handled?
- Does the software handle the situation of the external device being disconnected, not available, or full during a read or write?
- Are all conceivable errors handled by the software in an expected way?
- Have all error messages been checked for correctness, appropriateness, grammar, and spelling?

OTHER CHECKS

This best-of-the-rest list defines a few items that didn't fit well in the other categories. It's not by any means complete, but should give you ideas for specific items that should be added to a list tailored for your software project.

- Will the software work with languages other than English? Does it handle extended ASCII characters? Does it need to use Unicode instead of ASCII?
- If the software is intended to be portable to other compilers and CPUs, have allowances been made for this? Portability, if required, can be a huge issue if not planned and tested for.
- Has compatibility been considered so that the software will operate with different amounts of available memory, different internal hardware such as graphics and sound cards, and different peripherals such as printers and modems?

- Does compilation of the program produce any "warning" or "informational" messages? They usually indicate that something questionable is being done. Purists would argue that any warning message is unacceptable.

SUMMARY

Examining the code - static white-box testing - has proven to be an effective means for finding bugs early. It's a task that requires a great deal of preparation to make it a productive exercise, but many studies have shown that the time spent is well worth the benefits gained. To make it even more attractive, commercial software products, known as static analyzers, are available to automate a great deal of the work. The software reads in a program's source files and checks them against published standards and your own customizable guidelines. Compilers have also improved to the point that if you enable all their levels of error checking, they will catch many of the problems listed previously in the generic code review checklist. Some will even disallow use of functions with known security issues. These tools don't eliminate the tasks of code reviews or inspections - they just make it easier to accomplish and give testers more time to look even deeper for bugs.
If your team currently isn't doing testing at this level and you have some experience at programming, you might try suggesting it as a process to investigate. Programmers and managers may be apprehensive at first, not knowing if the benefits are that great - it's hard to claim, for example, that finding a bug during an inspection saved your project five days over finding it months later during black-box testing. But, static white-box testing is gaining momentum, and in some circles, projects can't ship reliable software without it.

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. Name several advantages to performing static white-box testing
2. True or False: Static white-box testing can find missing items as well as problems.
3. What key element makes formal reviews work?
4. Besides being more formal, what's the big difference between inspections and other types of reviews?
5. If a programmer was told that he could name his variables with only eight characters and the first character had to be capitalized, would that be a standard or a guideline?
6. Should you adopt the code review checklist from this chapter as your team's standard to verify its code?
7. A common security issue known as a buffer overrun is in a class of errors known as what? They are caused by what?

CHAPTER 7. TESTING THE SOFTWARE WITH X-RAY GLASSES

IN THIS CHAPTER

- Dynamic White-Box Testing
- Dynamic White-Box Testing Versus Debugging
- Testing the Pieces
- Data Coverage
- Code Coverage

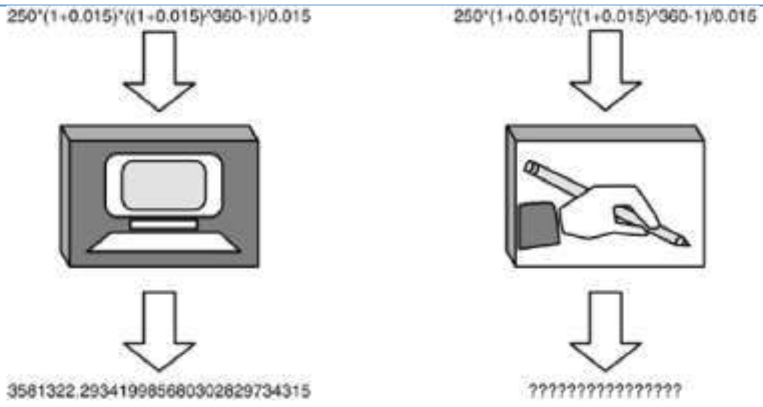
So far in Part II, "Testing Fundamentals," you've learned about three of the four fundamental testing techniques: static black box (testing the specification), dynamic black box (testing the software), and static white box (examining the code). In this chapter, you'll learn the fourth fundamental technique - dynamic white-box testing. You'll look into the software "box" with your X-ray glasses as you test the software.
In addition to your X-ray specs, you'll also need to wear your programmer's hat - if you have one. If you don't own one, don't be scared off. The examples used aren't that complex and if you take your time, you'll be able to follow them. Gaining even a small grasp of this type of testing will make you a much more effective black-box tester.
If you do have some programming experience, consider this chapter an introduction to a very wide-open testing field. Many software companies are hiring testers specifically to perform low-level testing of their software. They're looking for people with both programming and testing skills, which is often a rare mix and highly sought after.
Highlights from this chapter include

- What dynamic white-box testing is
- The difference between debugging and dynamic white-box testing
- What unit and integration testing are
- How to test low-level functions
- The data areas that need to be tested at a low level
- How to force a program to operate a certain way
- What different methods you can use to measure the thoroughness of your testing

DYNAMIC WHITE-BOX TESTING

By now you should be very familiar with the terms static, dynamic, white box, and black box. Knowing that this chapter is about dynamic white-box testing should tell you exactly what material it covers. Since it's dynamic, it must be about testing a running program and since it's white-box, it must be about looking inside the box, examining the code, and watching it as it runs. It's like testing the software with X-ray glasses.
Dynamic white-box testing, in a nutshell, is using information you gain from seeing what the code does and how it works to determine what to test, what not to test, and how to approach the testing. Another name commonly used for dynamic white-box testing is structural testing because you can see and use the underlying structure of the code to design and run your tests.
Why would it be beneficial for you to know what's happening inside the box, to understand how the software works? Consider Figure 7.1. This figure shows two boxes that perform the basic calculator operations of addition, subtraction, multiplication, and division.

FIGURE 7.1. YOU WOULD CHOOSE DIFFERENT TEST CASES IF YOU KNEW THAT ONE BOX CONTAINED A COMPUTER AND THE OTHER A PERSON WITH A PENCIL AND PAPER.



If you didn't know how the boxes worked, you would apply the dynamic black-box testing techniques you learned in Chapter 5, "Testing the Software with Blinders On." But, if you could look in the boxes and see that one contained a computer and the other contained a person with a pencil and paper, you would probably choose a completely different test approach for each one. Of course, this example is very simplistic, but it makes the point that knowing how the software operates will influence how you test. Dynamic white-box testing isn't limited just to seeing what the code does. It also can involve directly testing and controlling the software. The four areas that dynamic white-box testing encompasses are

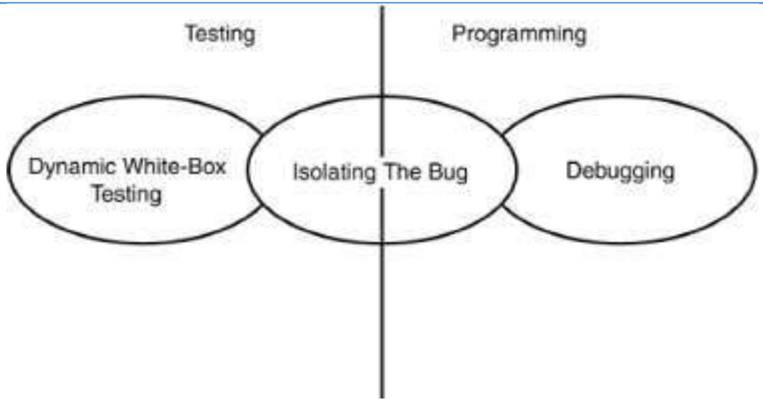
- Directly testing low-level functions, procedures, subroutines, or libraries. In Microsoft Windows, these are called Application Programming Interfaces (APIs).
- Testing the software at the top level, as a completed program, but adjusting your test cases based on what you know about the software's operation.
- Gaining access to read variables and state information from the software to help you determine whether your tests are doing what you thought. And, being able to force the software to do things that would be difficult if you tested it normally.
- Measuring how much of the code and specifically what code you "hit" when you run your tests and then adjusting your tests to remove redundant test cases and add missing ones.

Each area is discussed in the remainder of this chapter. Think about them as you read on and consider how they might be used to test software that you're familiar with.

DYNAMIC WHITE-BOX TESTING VERSUS DEBUGGING

It's important not to confuse dynamic white-box testing with debugging. If you've done some programming, you've probably spent many hours debugging code that you've written. The two techniques may appear similar because they both involve dealing with software bugs and looking at the code, but they're very different in their goals (see Figure 7.2).

FIGURE 7.2. DYNAMIC WHITE-BOX TESTING AND DEBUGGING HAVE DIFFERENT GOALS BUT THEY DO OVERLAP IN THE MIDDLE.



The goal of dynamic white-box testing is to find bugs. The goal of debugging is to fix them. They do overlap, however, in the area of isolating where and why the bug occurs. You'll learn more about this in Chapter 19, "Reporting What You Find," but for now, think of the overlap this way. As a software tester, you should narrow down the problem to the simplest test case that demonstrates the bug. If it's white-box testing, that could even include information about what lines of code look suspicious. The programmer who does the debugging picks the process up from there, determines exactly what is causing the bug, and attempts to fix it.

NOTE

If you're performing this low-level testing, you will use many of the same tools that programmers use. If the program is compiled, you will use the same compiler but possibly with different settings to enable better error detection. You will likely use a code-level debugger to single-step through the program, watch variables, set break conditions, and so on. You may also write your own programs to test separate code modules given to you to validate.

TESTING THE PIECES

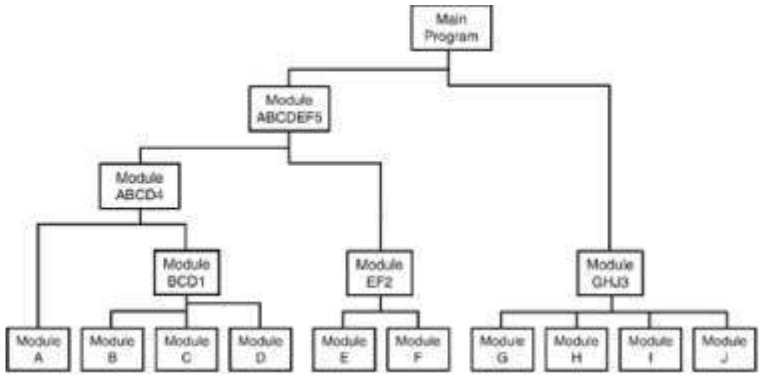
Recall from Chapter 2, "The Software Development Process," the various models for software development. The big-bang model was the easiest but the most chaotic. Everything was put together at once and, with fingers crossed, the team hoped that it all worked and that a product would be born. By now you've probably deduced that testing in such a model would be very difficult. At most, you could perform dynamic black-box testing, taking the near final product in one entire blob and exploring it to see what you could find.
You've learned that this approach is very costly because the bugs are found late in the game. From a testing perspective, there are two reasons for the high cost:

- It's difficult and sometimes impossible to figure out exactly what caused the problem. The software is a huge Rube Goldberg machine that doesn't work - the ball drops in one side, but buttered toast and hot coffee doesn't come out the other. There's no way to know which little piece is broken and causing the entire contraption to fail.
- Some bugs hide others. A test might fail. The programmer confidently debugs the problem and makes a fix, but when the test is rerun, the software still fails. So many problems were piled one on top the other that it's impossible to get to the core fault.

UNIT AND INTEGRATION TESTING

The way around this mess is, of course, to never have it happen in the first place. If the code is built and tested in pieces and gradually put together into larger and larger portions, there won't be any surprises when the entire product is linked together (see Figure 7.3).

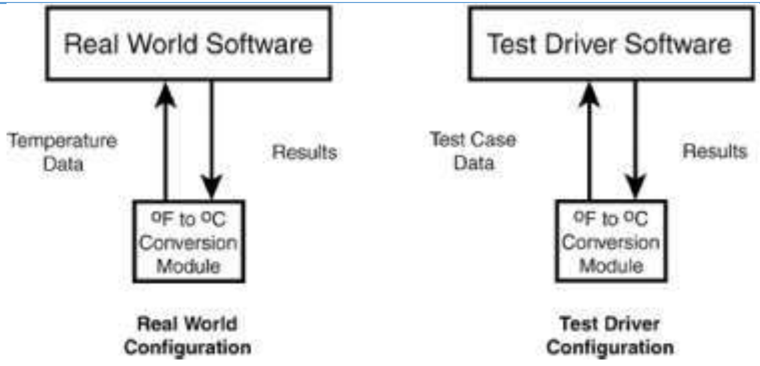
FIGURE 7.3. INDIVIDUAL PIECES OF CODE ARE BUILT UP AND TESTED SEPARATELY, AND THEN INTEGRATED AND TESTED AGAIN.



Testing that occurs at the lowest level is called unit testing or module testing. As the units are tested and the low-level bugs are found and fixed, they are integrated and integration testing is performed against groups of modules. This process of incremental testing continues, putting together more and more pieces of the software until the entire product - or at least a major portion of it - is tested at once in a process called system testing.
With this testing strategy, it's much easier to isolate bugs. When a problem is found at the unit level, the problem must be in that unit. If a bug is found when multiple units are integrated, it must be related to how the modules interact. Of course, there are exceptions to this, but by and large, testing and debugging is much more efficient than testing everything at once.

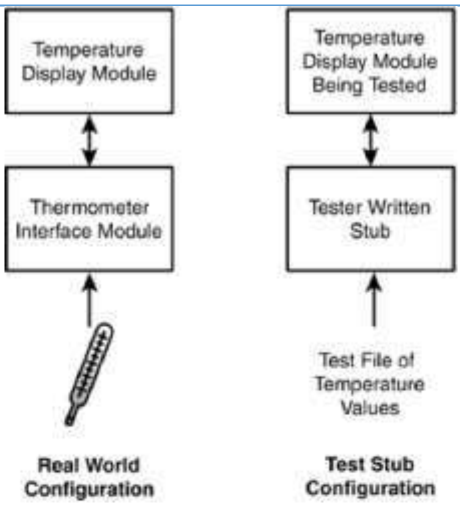
There are two approaches to this incremental testing: bottom-up and top-down. In bottom-up testing (see Figure 7.4), you write your own modules, called test drivers, that exercise the modules you're testing. They hook in exactly the same way that the future real modules will. These drivers send test-case data to the modules under test, read back the results, and verify that they're correct. You can very thoroughly test the software this way, feeding it all types and quantities of data, even ones that might be difficult to send if done at a higher level.

FIGURE 7.4. A TEST DRIVER CAN REPLACE THE REAL SOFTWARE AND MORE EFFICIENTLY TEST A LOW-LEVEL MODULE.



Top-down testing may sound like big-bang testing on a smaller scale. After all, if the higher-level software is complete, it must be too late to test the lower modules, right? Actually, that's not quite true. Look at Figure 7.5. In this case, a low-level interface module is used to collect temperature data from an electronic thermometer. A display module sits right above the interface, reads the data from the interface, and displays it to the user. To test the top-level display module, you'd need blow torches, water, ice, and a deep freeze to change the temperature of the sensor and have that data passed up the line.

FIGURE 7.5. A TEST STUB SENDS TEST DATA UP TO THE MODULE BEING TESTED.


Temperature Display Module
Thermometer Interface Module
Real World Configuration
Temperature Display Module Being Tested
Tester Written Stub
Test File of Temperature Values
Test Stub Configuration

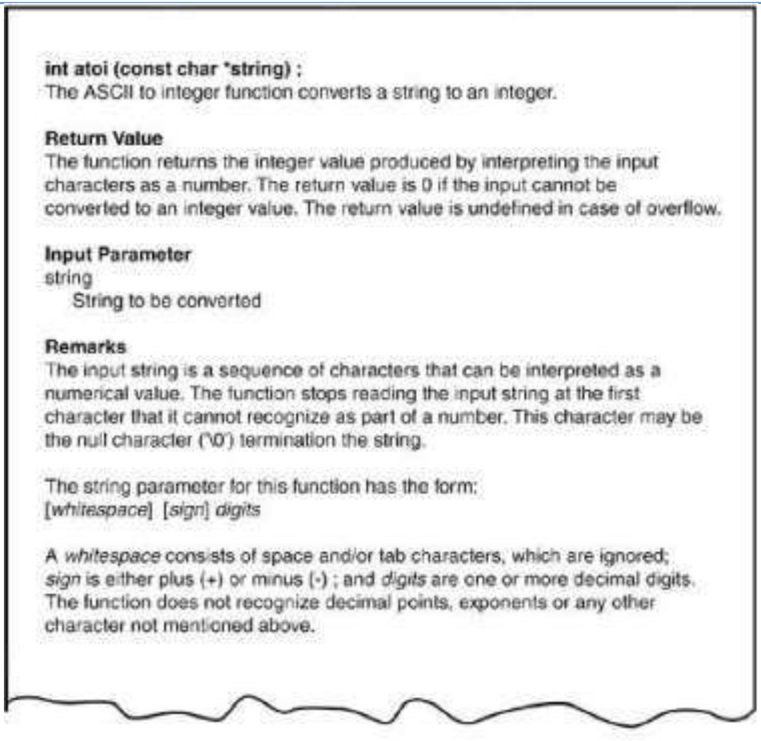
Rather than test the temperature display module by attempting to control the temperature of the thermometer, you could write a small piece of code called a stub that acts just like the interface module by feeding "fake" temperature values from a file directly to the display module. The display module would read the data and show the temperature just as though it was reading directly from a real thermometer interface module. It wouldn't know the difference. With this test stub configuration, you could quickly run through numerous test values and validate the operation of the display module.

AN EXAMPLE OF MODULE TESTING

A common function available in many compilers is one that converts a string of ASCII characters into an integer value.
What this function does is take a string of numbers, or + signs, and possible extraneous characters such as spaces and letters, and converts them to a numeric value - for example, the string "12345" gets converted to the number 12,345. It's a fairly common function that's often used to process values that a user might type into a dialog box - for example, someone's age or an inventory count.

The C language function that performs this operation is atoi(), which stands for "ASCII to Integer." Figure 7.6 shows the specification for this function. If you're not a C programmer, don't fret. Except for the first line, which shows how to make the function call, the spec is in English and could be used for defining the same function for any computer language.

FIGURE 7.6. THE SPECIFICATION SHEET FOR THE C LANGUAGE ATOI() FUNCTION.



int atoi (const char *string) ;
The ASCII to integer function converts a string to an integer.

Return Value
The function returns the integer value produced by interpreting the input characters as a number. The return value is 0 if the input cannot be converted to an integer value. The return value is undefined in case of overflow.

Input Parameter
string
String to be converted

Remarks
The input string is a sequence of characters that can be interpreted as a numerical value. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character may be the null character ('\0') termination the string.

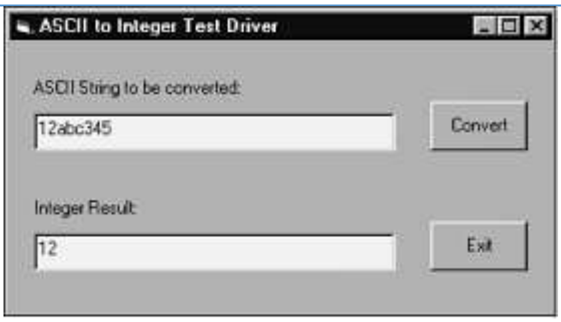
The string parameter for this function has the form:
[*whitespace*] [*sign*] *digits*

A *whitespace* consists of space and/or tab characters, which are ignored; *sign* is either plus (+) or minus (-) ; and *digits* are one or more decimal digits. The function does not recognize decimal points, exponents or any other character not mentioned above.

If you're the software tester assigned to perform dynamic white-box testing on this module, what would you do?
First, you would probably decide that this module looks like a bottom module in the program, one that's called by higher up modules but doesn't call anything itself. You could confirm this by looking at the internal code. If this is true, the logical approach is to write a test driver to exercise the module independently from the rest of the program.

This test driver would send test strings that you create to the atoi() function, read back the return values for those strings, and compare them with your expected results. The test driver would most likely be written in the same language as the function - in this case, C - but it's also possible to write the driver in other languages as long as they interface to the module you're testing.
This test driver can take on several forms. It could be a simple dialog box, as shown in Figure 7.7, that you use to enter test strings and view the results. Or it could be a standalone program that reads test strings and expected results from a file. The dialog box, being user driven, is very interactive and flexible - it could be given to a black-box tester to use. But the standalone driver can be very fast reading and writing test cases directly from a file.

FIGURE 7.7. A DIALOG BOX TEST DRIVER CAN BE USED TO SEND TEST CASES TO A MODULE BEING TESTED.



Next, you would analyze the specification to decide what black-box test cases you should try and then apply some equivalence partitioning techniques to reduce the total set (remember Chapter 5?). Table 7.1 shows examples of a few test cases with their input strings and expected output values. This table isn't intended to be a comprehensive list.

TABLE 7.1. SAMPLE ASCII TO INTEGER CONVERSION TEST CASES

Input String	Output Integer Value
"1"	1
"1"	1
"+1"	1
"0"	0
"0"	0
"+0"	0
"1.2"	1
"23"	2
"abc"	0
"a123"	0
and so on	

Lastly, you would look at the code to see how the function was implemented and use your white-box knowledge of the module to add or remove test cases.

NOTE

Creating your black-box testing cases based on the specification, before your white-box cases, is important. That way, you are truly testing what the module is intended to do. If you first create your test cases based on a white-box view of the module, by examining the code, you will be biased into creating test cases based on how the module works. The programmer could have misinterpreted the specification and your test cases would then be wrong. They would be precise, perfectly testing the module, but they wouldn't be accurate because they wouldn't be testing the intended operation.

Adding and removing test cases based on your white-box knowledge is really just a further refinement of the equivalence partitions done with inside information. Your original black-box test cases might have assumed an internal ASCII table that would make cases such as "a123" and "z123" different and important. After examining the software, you could find that instead of an ASCII table, the programmer simply checked for numbers, and + signs, and blanks. With that information, you might decide to remove one of these cases because both of them are in the same equivalence partition.

With close inspection of the code, you could discover that the handling of the + and signs looks a little suspicious. You might not even understand how it works. In that situation, you could add a few more test cases with embedded + and signs, just to be sure.

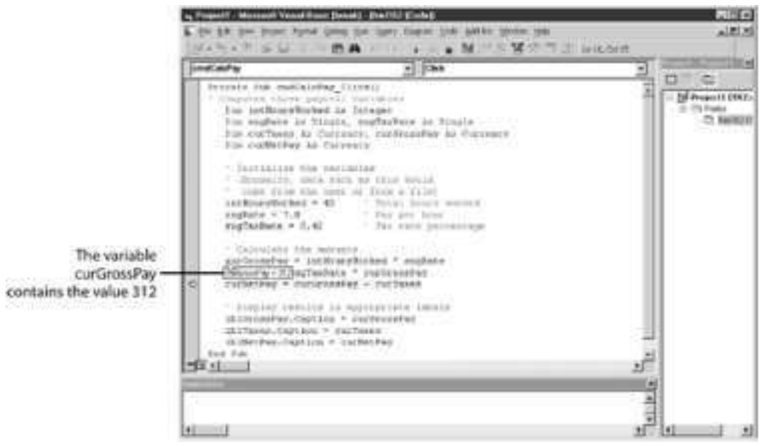
DATA COVERAGE

The previous example of white-box testing the atoi() function was greatly simplified and glossed over some of the details of looking at the code to decide what adjustments to make to the test cases. In reality, there's quite a bit more to the process than just perusing the software for good ideas.
The logical approach is to divide the code just as you did in black-box testing - into its data and its states (or program flow). By looking at the software from the same perspective, you can more easily map the white-box information you gain to the black-box cases you've already written.
Consider the data first. Data includes all the variables, constants, arrays, data structures, keyboard and mouse input, files and screen input and output, and I/O to other devices such as modems, networks, and so on.

DATA FLOW

Data flow coverage involves tracking a piece of data completely through the software. At the unit test level this would just be through an individual module or function. The same tracking could be done through several integrated modules or even through the entire software product - although it would be more time-consuming to do so.
If you test a function at this low level, you would use a debugger and watch variables to view the data as the program runs (see Figure 7.8). With black-box testing, you only know what the value of the variable is at the beginning and at the end. With dynamic white-box testing you could also check intermediate values during program execution. Based on what you see you might decide to change some of your test cases to make sure the variable takes on interesting or even risky interim values.

FIGURE 7.8. A DEBUGGER AND WATCH VARIABLES CAN HELP YOU TRACE A VARIABLE'S VALUES THROUGH A PROGRAM.



SUB-BOUNDARIES

Sub-boundaries were discussed in Chapter 5 in regard to embedded ASCII tables and powers-of-two. These are probably the most common examples of sub-boundaries that can cause bugs, but every piece of software will have its own unique sub-boundaries, too. Here are a few more examples:

- A module that computes taxes might switch from using a data table to using a formula at a certain financial cut-off point.
- An operating system running low on RAM may start moving data to temporary storage on the hard drive. This sub-boundary may not even be fixed. It may change depending on how much space remains on the disk.
- To gain better precision, a complex numerical analysis program may switch to a different equation for solving the problem depending on the size of the number.

If you perform white-box testing, you need to examine the code carefully to look for sub-boundary conditions and create test cases that will exercise them. Ask the programmer who wrote the code if she knows about any of these situations and pay special attention to internal tables of data because they're especially prone to sub-boundary conditions.

FORMULAS AND EQUATIONS

Very often, formulas and equations are buried deep in the code and their presence or effect isn't always obvious from the outside. A financial program that computes compound interest will definitely have this formula somewhere in the software:

$$A=P(1+r/n)^{nt}$$

where

P = principal amount

r = annual interest rate

n = number of times the interest is compounded per year

t = number of years

A = amount of money after time t

A good black-box tester would hopefully choose a test case of n=0, but a white-box tester, after seeing the formula in the code, would know to try n=0 because that would cause the formula to blow up with a divide-by-zero error.
But, what if n was the result of another computation? Maybe the software sets the value of n based on other user input or algorithmically tries different n values in an attempt to find the lowest payment. You need to ask yourself if there's any way that n can ever become zero and figure out what inputs to feed the program to make that happen.
TIP
Scour your code for formulas and equations, look at the variables they use, and create test cases and equivalence partitions for them in addition to the normal inputs and outputs of the program.

ERROR FORCING

The last type of data testing covered in this chapter is error forcing. If you're running the software that you're testing in a debugger, you don't just have the ability to watch variables and see what values they hold - you can also force them to specific values.
In the preceding compound interest calculation, if you couldn't find a direct way to set the number of compoundings (n) to zero, you could use your debugger to force it to zero. The software would then have to handle it…or not.
NOTE
Be careful if you use error forcing and make sure you aren't creating a situation that can never happen in the real world. If the programmer checked that n was greater than zero at the top of the function and n was never used until the formula, setting it to zero and causing the software to fail would be an invalid test case.

If you take care in selecting your error forcing scenarios and double-check with the programmer to assure that they're valid, error forcing can be an effective tool. You can execute test cases that would otherwise be difficult to perform.

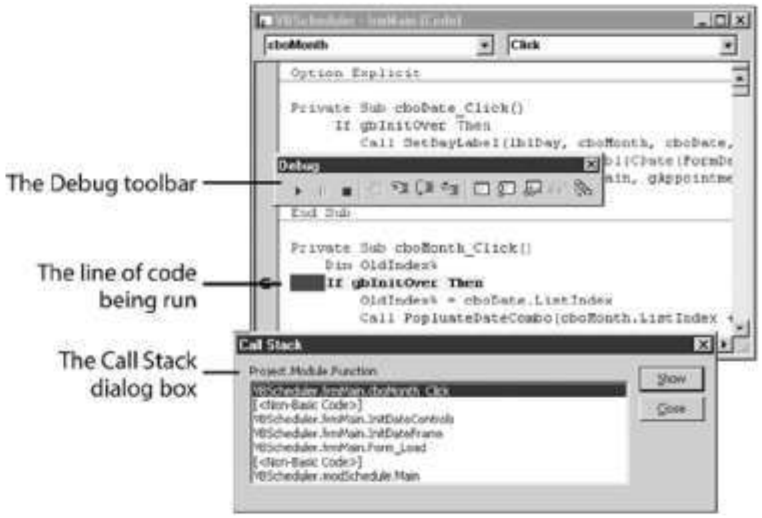
FORCING ERROR MESSAGES

A great way to use error forcing is to cause all the error messages in your software to appear. Most software uses internal error codes to represent each error message. When an internal error condition flag is set, the error handler takes the variable that holds the error code, looks up the code in a table, and displays the appropriate message.
Many errors are difficult to create - like hooking up 2,049 printers. But if all you want to do is test that the error messages are correct (spelling, language, formatting, and so on), using error forcing can be a very efficient way to see all of them. Keep in mind, though, that you aren't testing the code that detects the error, just the code that displays it.

CODE COVERAGE

As with black-box testing, testing the data is only half the battle. For comprehensive coverage you must also test the program's states and the program's flow among them. You must attempt to enter and exit every module, execute every line of code, and follow every logic and decision path through the software. This type of testing is known as code coverage testing.
Code coverage is dynamic white-box testing because it requires you to have full access to the code to view what parts of the software you pass through when you run your test cases.
The simplest form of code coverage testing is using your compiler's debugger to view the lines of code you visit as you single-step through the program. Figure 7.9 shows an example of the Visual Basic debugger in operation.

FIGURE 7.9. THE DEBUGGER ALLOWS YOU TO SINGLE-STEP THROUGH THE SOFTWARE TO SEE WHAT LINES OF CODE AND MODULES YOU EXECUTE WHILE RUNNING YOUR TEST CASES.

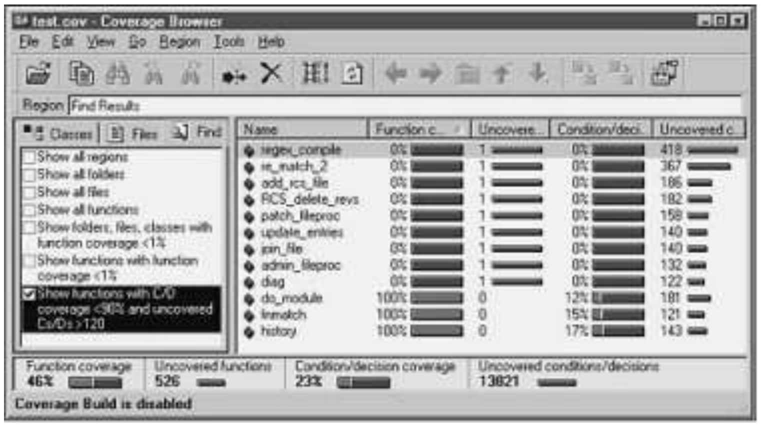


```
Option Explicit

Private Sub cboDate_Click()
    If gbInitOver Then
        Call SetDayLabel(lblDay, cboMonth, cboDate,
[illegible]bl(CDate(FormDe
[illegible]ain, gAppointme
End Sub

Private Sub cboMonth_Click()
    Dim OldIndex%
    If gbInitOver Then
        OldIndex% = cboDate.ListIndex
        Call PopluateDateCombo(cboMonth.ListIndex +
```

Project.Module.Function
VBScheduler.frmMain.cboMonth_Click
[<Non-Basic Code>]
VBScheduler.frmMain.InitDateControls
VBScheduler.frmMain.InitDateFrame
VBScheduler.frmMain.Form_Load
[<Non-Basic Code>]
VBScheduler.modSchedule.Main

For very small programs or individual modules, using a debugger is often sufficient. However, performing code coverage on most software requires a specialized tool known as a code coverage analyzer. Figure 7.10 shows an example of such a tool.

FIGURE 7.10. A CODE COVERAGE ANALYZER PROVIDES DETAILED INFORMATION ABOUT HOW EFFECTIVE YOUR TEST CASES ARE. (THIS FIGURE IS COPYRIGHT AND COURTESY OF BULLSEYE TESTING TECHNOLOGY.)


test.cov - Coverage Browser
File Edit View Go Region Tools Help
Region
Classes
Files
Find

Region: Find Results

- [ ] Show all regions
- [ ] Show all folders
- [ ] Show all files
- [ ] Show all functions
- [ ] Show folders, files, classes with function coverage <1%
- [ ] Show functions with function coverage <1%
- [x] Show functions with C/D coverage <90% and uncovered Cs/Ds >120

Name	Function c...	Uncovere...	Condition/deci...	Uncovered c...
regex_compile	0%	1	0%	418
re_match_2	0%	1	0%	367
add_rcs_file	0%	1	0%	186
RCS_delete_revs	0%	1	0%	182
patch_fileproc	0%	1	0%	158
update_entries	0%	1	0%	140
join_file	0%	1	0%	140
admin_fileproc	0%	1	0%	132
diag	0%	1	0%	122
do_module	100%	0	12%	181
lnmatch	100%	0	15%	121
history	100%	0	17%	143

Function coverage	Uncovered functions	Condition/decision coverage	Uncovered conditions/decisions
46%	526	23%	13821

Coverage Build is disabled

Code coverage analyzers hook into the software you're testing and run transparently in the background while you run your test cases. Each time a function, a line of code, or a logic decision is executed, the analyzer records the information. You can then obtain statistics that identify which portions of the software were executed and which portions weren't. With this data you'll know

- What parts of the software your test cases don't cover. If a specific module is never executed, you know that you need to write additional test cases for testing that module's function.
- Which test cases are redundant. If you run a series of test cases and they don't increase the percentage of code covered, they are likely in the same equivalence partition.

- What new test cases need to be created for better coverage. You can look at the code that has low coverage, see how it works and what it does, and create new test cases that will exercise it more thoroughly.

You will also have a general feel for the quality of the software. If your test cases cover 90 percent of the software and don't find any bugs, the software is in pretty good shape. If, on the other hand, your tests cover only 50 percent of the software and you're still finding bugs, you know you still have work to do.
REMINDER
Don't forget about the Pesticide Paradox described in Chapter 3, "The Realities of Software Testing" - the more you test the software, the more it becomes immune to your tests. If your test cases cover 90 percent of the software and you're not finding any bugs, it could still be that your software isn't in good shape - it could be that it has become immune. Adding new test cases may reveal that the next 10 percent is very buggy!

PROGRAM STATEMENT AND LINE COVERAGE

The most straightforward form of code coverage is called statement coverage or line coverage. If you're monitoring statement coverage while you test your software, your goal is to make sure that you execute every statement in the program at least once. In the case of the short program shown in Listing 7.1, 100 percent statement coverage would be the execution of lines 1 through 4.

LISTING 7.1. IT'S VERY EASY TO TEST EVERY LINE OF THIS SIMPLE PROGRAM

```
1: PRINT "Hello World"

2: PRINT "The date is: "; Date$

3: PRINT "The time is: "; Time$

4: END
```

You might think this would be the perfect way to make sure that you tested your program completely. You could run your tests and add test cases until every statement in the program is touched. Unfortunately, statement coverage is misleading. It can tell you if every statement is executed, but it can't tell you if you've taken all the paths through the software.

BRANCH COVERAGE

Attempting to cover all the paths in the software is called path testing. The simplest form of path testing is called branch coverage testing. Consider the program shown in Listing 7.2.

LISTING 7.2. THE IF STATEMENT CREATES ANOTHER BRANCH THROUGH THE CODE

```
1: PRINT "Hello World"

2: IF Date$ = "01-01-2000" THEN

3:     PRINT "Happy New Year"

4:     END IF

5: PRINT "The date is: "; Date$

6: PRINT "The time is: "; Time$
```

```
7: END
```

If you test this program with the goal of 100 percent statement coverage, you would need to run only a single test case with the Date$ variable set to January 1, 2000. The program would then execute the following path:

Lines 1, 2, 3, 4, 5, 6, 7

Your code coverage analyzer would state that you tested every statement and achieved 100 percent coverage. You could quit testing, right? Wrong! You may have tested every statement, but you didn't test every branch.
Your gut may be telling you that you still need to try a test case for a date that's not January 1, 2000. If you did, the program would execute the other path through the program:

Lines 1, 2, 5, 6, 7

Most code coverage analyzers will account for code branches and report both statement coverage and branch coverage results separately, giving you a much better idea of your test's effectiveness.

CONDITION COVERAGE

Just when you thought you had it all figured out, there's yet another complication to path testing. Listing 7.3 shows a slight variation to Listing 7.2. An extra condition is added to the IF statement in line 2 that checks the time as well as the date. Condition coverage testing takes the extra conditions on the branch statements into account.

LISTING 7.3. THE MULTIPLE CONDITIONS IN THE IF STATEMENT CREATE MORE PATHS THROUGH THE CODE

```
1: PRINT "Hello World"

2: IF Date$ = "01-01-2000" AND Time$ = "00:00:00" THEN

3:     PRINT "Happy New Year"

4:     END IF

5: PRINT "The date is: "; Date$

6: PRINT "The time is: "; Time$

7: END
```

In this sample program, to have full condition coverage testing, you need to have the four sets of test cases shown in Table 7.2. These cases assure that each possibility in the IF statement are covered.

TABLE 7.2. TEST CASES TO ACHIEVE FULL CONDITION COVERAGE

Date$	**Time$**	**Line # Execution**

TABLE 7.2. TEST CASES TO ACHIEVE FULL CONDITION COVERAGE

Date$	Time$	Line # Execution
01-01-1999	11:11:11	1,2,5,6,7
01-01-1999	00:00:00	1,2,5,6,7
01-01-2000	11:11:11	1,2,5,6,7
01-01-2000	00:00:00	1,2,3,4,5,6,7

If you were concerned only with branch coverage, the first three conditions would be redundant and could be equivalence partitioned into a single test case. But, with condition coverage testing, all four cases are important because they exercise the different conditions of the IF statement in line 2False-False, False-True, True-False, and True-True.
As with branch coverage, code coverage analyzers can be configured to consider conditions when reporting their results. If you test for condition coverage, you will achieve branch coverage and therefore achieve statement coverage.
NOTE
If you manage to test every statement, branch, and condition (and that's impossible except for the smallest of programs), you still haven't tested the program completely. Remember, all the data errors discussed in the first part of this chapter are still possible. The program flow and the data together make up the operation of the software.

SUMMARY

This chapter showed you how having access to the software's source code while the program is running can open up a whole new area of software testing. Dynamic white-box testing is a very powerful approach that can greatly reduce your test work by giving you "inside" information about what to test. By knowing the details of the code, you can eliminate redundant test cases and add test cases for areas you didn't initially consider. Either way, you can greatly improve your testing effectiveness.
Chapters 4 through 7 covered the fundamentals of software testing:

- Static black-box testing involves examining the specification and looking for problems before they get written into the software.
- Dynamic black-box testing involves testing the software without knowing how it works.
- Static white-box testing involves examining the details of the written code through formal reviews and inspections.
- Dynamic white-box testing involves testing the software when you can see how it works and basing your tests on that information.

In a sense, this is all there is to software testing. Of course, reading about it in four chapters and putting it into practice are very different things. Being a good software tester requires lots of dedication and hard work. It takes practice and experience to know when and how to best apply these fundamental techniques.
In Part III, "Applying Your Testing Skills," you'll learn about different types of software testing and how you can apply the skills from your "black and white testing box" to real-world scenarios.

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. Why does knowing how the software works influence how and what you should test?
2. What's the difference between dynamic white-box testing and debugging?
3. What are two reasons that testing in a big-bang software development model is nearly impossible? How can these be addressed?
4. True or False: If your product development is in a hurry, you can skip module testing and proceed directly to integration testing.
5. What's the difference between a test stub and a test driver?
6. True or False: Always design your black-box test cases first.
7. Of the three code coverage measures described, which one is the best? Why?
8. What's the biggest problem of white-box testing, either static or dynamic?

CHAPTER 8. CONFIGURATION TESTING

IN THIS CHAPTER

- An Overview of Configuration Testing
- Approaching the Task
- Obtaining the Hardware
- Identifying Hardware Standards
- Configuration Testing Other Hardware

Life could be so simple. All computer hardware could be identical. All software could be written by the same company. There wouldn't be confusing option buttons to click or check boxes to check. Everything would interface perfectly the first time, every time. How boring.
In the real world, 50,000-square-foot computer superstores are offering PCs, printers, monitors, network cards, modems, scanners, digital cameras, peripherals, net-cams, and hundreds of other computer doodads from thousands of companies - all able to connect to your PC!
If you're just getting started at software testing, one of your first tasks may be configuration testing. You'll be making sure that your software works with as many different hardware combinations as possible.
If you're not testing software for a popular hardware platform such as a PC or a Mac - that is, if you're testing some specialized proprietary system - you will still need to consider configuration issues. You can easily tailor what you learn in this chapter to your situation.
The first part of this chapter deals with the generalities of PC configuration testing and then moves into the specifics of testing printers, display adapters (video cards), and sound cards for a PC. Although the examples are based on desktop computers, you can extrapolate the methods to just about any type of configuration test problem. New and different devices are released every day, and it will be your job to figure out how to test them.
Highlights of this chapter include

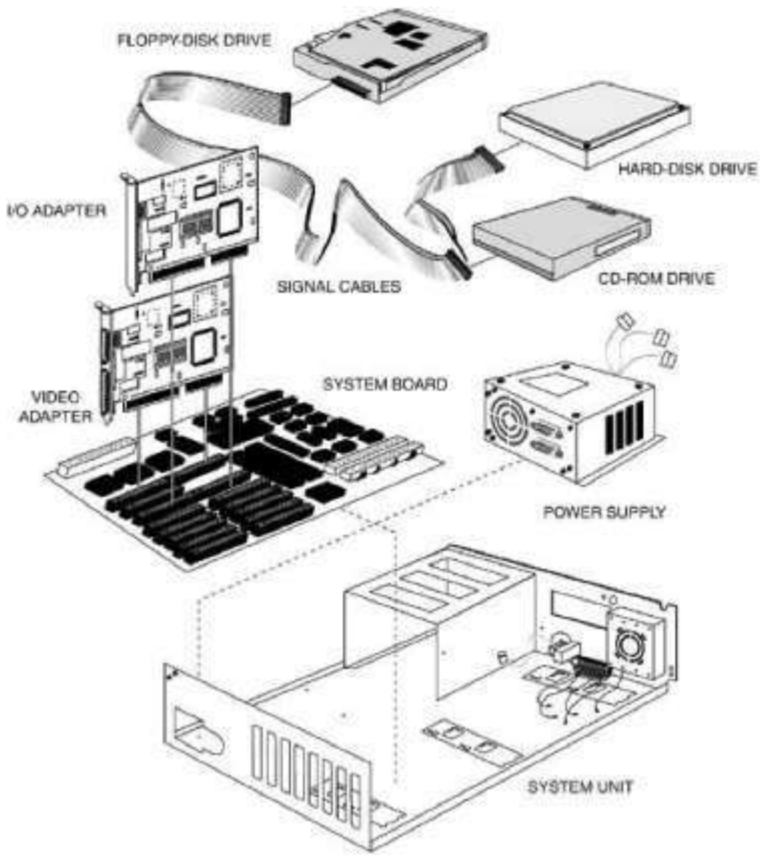
- Why configuration testing is necessary
- Why configuration testing can be a huge job
- A basic approach to configuration testing
- How to find the hardware you need to test with
- What to do if you're not testing software for a desktop computer

AN OVERVIEW OF CONFIGURATION TESTING

The next time you're in one of those computer superstores, look at a few software boxes and read over the system requirements. You'll see things such as PC with a Pentium 4 processor, 1024x768 32-bit color monitor, 32-bit audio card, game port, and so on. Configuration testing is the process of checking the operation of the software you're testing with all these various types of hardware. Consider the different configuration possibilities for a standard Windows-based PC used in homes and businesses:

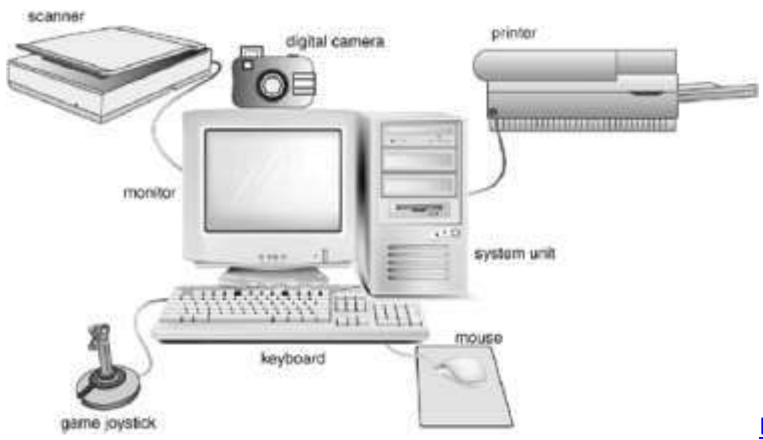
- The PC. There are several well-known computer manufacturers, such as Dell, Gateway, Hewlett Packard, and others. Each one builds PCs using components designed themselves or obtained from other manufacturers. Many hobbyists even build their own PCs using off-the-shelf components available at computer superstores.
- Components. Most PCs are modular and built up from various system boards, component cards, and other internal devices such as disk drives, CD-ROM drives, DVD burners, video, sound, fax modem, and network cards (see Figure 8.1). There are TV tuner cards and specialized cards for video capture and home automation. There are even input/output cards that can give a PC the ability to control a small factory! These internal devices are built by hundreds of different manufacturers.

FIGURE 8.1. NUMEROUS INTERNAL COMPONENTS MAKE UP A PC'S CONFIGURATION.



- Peripherals. Peripherals, shown in Figure 8.2, are the printers, scanners, mice, keyboards, monitors, cameras, joysticks, and other devices that plug into your system and operate externally to the PC.

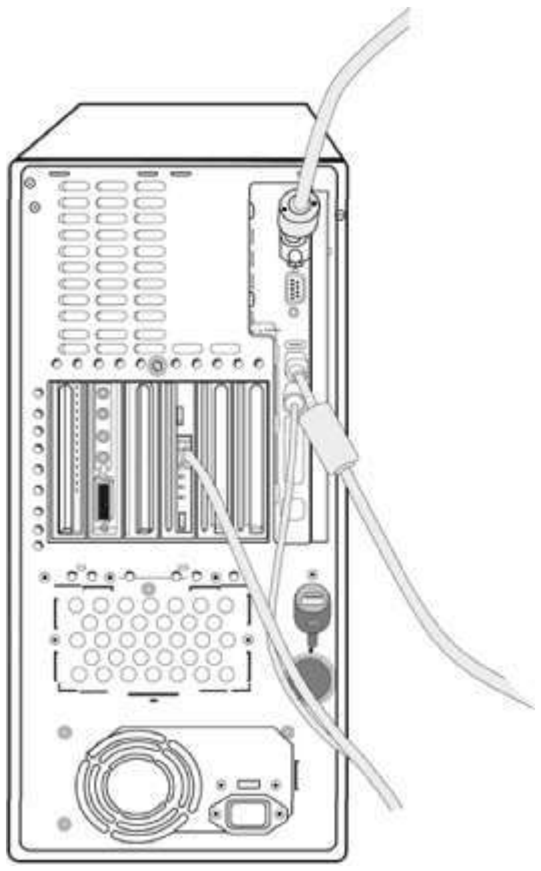
FIGURE 8.2. A PC CAN CONNECT TO A WIDE ASSORTMENT OF PERIPHERALS.



D:\Documents and Settings\jvazquez\My Documents\Vivek Jain\Local Settings\Temp\images\0672327988\graphics\08fig02_alt.jpg;446287

- Interfaces. The components and peripherals plug into your PC through various types of interface connectors (see Figure 8.3). These interfaces can be internal or external to the PC. Typical names for them are ISA, PCI, USB, PS/2, RS/232, RJ-11, RJ-45, and Firewire. There are so many different possibilities that hardware manufacturers will often create the same peripheral with different interfaces. It's possible to buy the exact same mouse in three different configurations!

FIGURE 8.3. THE BACK OF A PC SHOWS NUMEROUS INTERFACE CONNECTORS FOR ATTACHING PERIPHERALS.



- Options and memory. Many components and peripherals can be purchased with different hardware options and memory sizes. Printers can be upgraded to support extra fonts or accept more memory to speed up printing. Graphics cards with more memory can support additional colors and higher resolutions. The system board can have different versions of BIOS (its Basic Input Output System) and, of course, various amounts of memory.
- Device Drivers. All components and peripherals communicate with the operating system and the software applications through low-level software called device drivers. These drivers are often provided by the hardware device manufacturer and are installed when you set up the hardware. Although technically they are software, for testing purposes they are considered part of the hardware configuration.

If you're a tester gearing up to start configuration testing on a piece of software, you need to consider which of these configuration areas would be most closely tied to the program. A highly graphical computer game will require lots of attention to the video and sound areas. A greeting card program will be especially vulnerable to printer issues. A fax or communications program will need to be tested with numerous modems and network configurations.
You may be wondering why this is all necessary. After all, there are standards to meet for building hardware, whether it's for an off-the-shelf PC or a specialized computer in a hospital. You would expect that if everyone designed their hardware to a set of standards, software would just work with it without any problems. In an ideal world, that would happen, but unfortunately, standards aren't always followed. Sometimes, the standards are fairly loose - call them guidelines. Card and peripheral manufacturers are always in tight competition with one another and frequently bend the rules to squeeze in an extra feature or to get in a last little bit of performance gain. Often the device drivers are rushed and packed into the box as the hardware goes out the door. The result is software that doesn't work correctly with certain hardware configurations.

ISOLATING CONFIGURATION BUGS

Those configuration bugs can bite hard. Remember the Disney Lion King bug described in Chapter 1, "Software Testing Background"? That was a configuration problem. The software's sound didn't work only on a few, but very popular, hardware configurations. If you've ever been playing a game or using a graphics program and the colors suddenly go crazy or pieces of windows get left behind as you drag them, you've probably discovered a display adapter configuration bug. If you've ever spent hours (or days!) trying to get an old program to work with your new printer, it's probably a configuration bug.
NOTE
The sure way to tell if a bug is a configuration problem and not just a bug that would occur in any configuration is to perform the exact same operation that caused the problem, step by step, on another computer with a completely different hardware setup. If the bug doesn't occur, it's very likely a specific configuration problem that's revealed by the unique hardware used in the test.

Assume that you test your software on a unique configuration and discover a problem. Who should fix the bugyour team or the hardware manufacturer? That could turn out to be a million-dollar question.
First you need to figure out where the problem lies. This is usually a dynamic white-box testing and programmer-debugging effort. A configuration problem can occur for several reasons, all requiring someone to carefully examine the code while running the software under different configurations to find the bug:

- Your software may have a bug that appears under a broad class of configurations. An example is if your greeting card program works fine with laser printers but not with inkjet printers.

- Your software may have a bug specific only to one particular configuration - it doesn't work on the OkeeDoKee Model BR549 InkJet Deluxe printer.
- The hardware device or its device drivers may have a bug that only your software reveals. Maybe your software is the only one that uses a unique display card setting. When your software is run with a specific video card, the PC crashes.
- The hardware device or its device drivers may have a bug that can be seen with lots of other software - although it may be particularly obvious with yours. An example would be if a specific printer driver always defaulted to draft mode and your photo printing software had to set it to high-quality every time it printed.

In the first two cases, it seems fairly straightforward that your project team is responsible for fixing the bug. It's your problem. You should fix it.

In the last two cases, things get blurry. Say the bug is in a printer and that printer is the most popular in the world, with tens of millions in use. Your software obviously needs to work with that printer. It may take the printer vendor months to fix the problem (if it does at all) so your team will need to make changes to your software, even though the software is doing everything right, to work around the bug.

In the end, it's your team's responsibility to address the problem, no matter where it lies. Your customers don't care why or how the bug is happening, they just want the new software they purchased to work on their system's configuration.

OF PURPLE FUZZ AND SOUND CARDS

In 1997 Microsoft released its ActiMates Barney character and supporting CD-ROM learning software for kids. These animatronic dolls interacted with the software through a two-way radio in the doll and another radio connected to a PC.

The PC's radio connected to a seldom-used interface on most sound cards called an MIDI connector. This interface is used for music keyboards and other musical instruments. Microsoft assumed the connector would be a good choice because most people don't own musical devices. It would likely not have anything plugged into it and would be available for use with the ActiMates radio.

During configuration testing, a typical amount of bugs showed up. Some were due to sound card problems, some were in the ActiMates software. There was one bug, however, that could never quite be pinned down. It seemed that occasionally, randomly, the PC running the software would just lock up and would require rebooting. This problem occurred only with the most popular sound card on the market - of course.

With just weeks left in the schedule, a concerted effort was put together to resolve the problem. After a great deal of configuration testing and debugging, the bug was isolated to the sound card's hardware. It seems that the MIDI connector always had this bug, but, being so seldom used, no one had ever seen it. The ActiMates software exposed it for the first time.

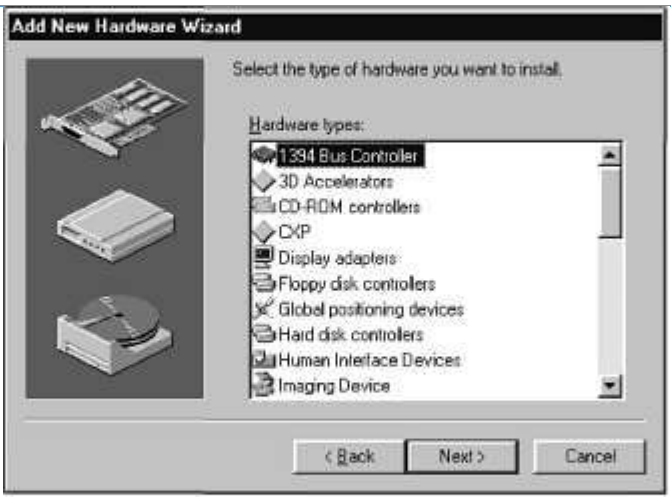
There was a mad scramble, lots of denials and finger pointing, and lots of late nights. In the end, the sound card manufacturer conceded that there was a problem and promised to work around the bug in updated versions of its device driver. Microsoft included a fixed driver on the ActiMates CD-ROM and made changes to the software that attempted to make the bug occur less frequently. Despite all those efforts, sound card compatibility problems were the top reason that people called in for assistance with the product.

SIZING UP THE JOB

The job of configuration testing can be a huge undertaking. Suppose that you're testing a new software game that runs under Microsoft Windows. The game is very graphical, has lots of sound effects, allows multiple players to compete against each other over the phone lines, and can print out game details for strategy planning.

At the least, you'll need to consider configuration testing with different graphics cards, sound cards, modems, and printers. The Windows Add New Hardware Wizard (see Figure 8.4) allows you to select hardware in each of these categories - and 25 others.

FIGURE 8.4. THE MICROSOFT WINDOWS ADD NEW HARDWARE WIZARD DIALOG BOX ALLOWS YOU TO ADD NEW HARDWARE TO YOUR PC'S CURRENT CONFIGURATION.

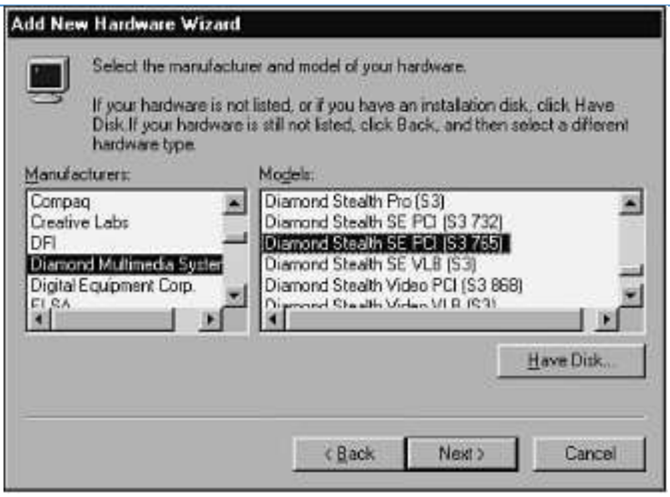

Add New Hardware Wizard

Select the type of hardware you want to install.

Hardware types:
- 1394 Bus Controller
- 3D Accelerators
- CD-ROM controllers
- CXP
- Display adapters
- Floppy disk controllers
- Global positioning devices
- Hard disk controllers
- Human Interface Devices
- Imaging Device

Under each hardware category are the different manufacturers and models (see Figure 8.5). Keep in mind, these are only the models with support built into Windows. Many other models provide their own setup disks with their hardware.

FIGURE 8.5. EACH TYPE OF HARDWARE HAS NUMEROUS MANUFACTURERS AND MODELS.


Add New Hardware Wizard

Select the manufacturer and model of your hardware.

If your hardware is not listed, or if you have an installation disk, click Have Disk. If your hardware is still not listed, click Back, and then select a different hardware type.

Manufacturers:
- Compaq
- Creative Labs
- DFI
- Diamond Multimedia Systen
- Digital Equipment Corp.
- ELSA

Models:
- Diamond Stealth Pro (S3)
- Diamond Stealth SE PCI (S3 732)
- Diamond Stealth SE PCI (S3 765)
- Diamond Stealth SE VLB (S3)
- Diamond Stealth Video PCI (S3 868)
- Diamond Stealth Video VLB (S3)

If you decided to perform a full, comprehensive configuration test, checking every possible make and model combination, you'd have a huge job ahead of you.
Say there are approximately 336 possible display cards, 210 sound cards, 1500 modems, and 1200 printers. The number of test combinations is 336 x 210 x 1500 x 1200, for a total in the billions - way too many to consider!
If you limited your testing to exclude combinations, just testing each card individually at about 30 minutes per configuration, you'd be at it for about a year. Keep in mind that's just one pass through the configurations. It's not uncommon with bug fixes to run two or three configuration test passes before a product is released.

The answer to this mess, as you've hopefully deduced, is equivalence partitioning. You need to figure out a way to reduce the huge set of possible configurations to the ones that matter the most. You'll assume some risk by not testing everything, but that's what software testing is all about.

APPROACHING THE TASK

The decision-making process that goes into deciding what devices to test with and how they should be tested is a fairly straightforward equivalence partition project.

What's important, and what makes the effort a success or not, is the information you use to make the decisions. If you're not experienced with the hardware that your software runs on, you should learn as much as you can and bring in other experienced testers or programmers to help you. Ask a lot of questions and make sure you get your plan approved.

The following sections show the general process that you should use when planning your configuration testing.

DECIDE THE TYPES OF HARDWARE YOU'LL NEED

Does your application print? If so, you'll need to test printers. If it has sound, you'll need to test sound cards. If it's a photo or graphics program, you'll likely need scanners and digital cameras. Look closely at your software feature set to make sure that you cover everything. Put your software disk on a table and ask yourself what hardware pieces you need to put together to make it work.

ONLINE REGISTRATION

An example of a feature that you can easily overlook when selecting what hardware to test with is online registration. Many programs today allow users to register their software during installation via modem or broadband connections. Users type in their name, address, and other personal data, click a button, and the modem dials out to a computer at the software company where it downloads the information and completes the registration. The software may not do anything else with online communications. But, if it has online registration, you will need to consider modems and network communications as part of your configuration testing.

DECIDE WHAT HARDWARE BRANDS, MODELS, AND DEVICE DRIVERS ARE AVAILABLE

If you're putting out a cutting-edge graphics program, you probably don't need to test that it prints well on a 1987 black-and-white dot-matrix printer. Work with your sales and marketing people to create a list of hardware to test with. If they can't or won't help, check out the recent equipment reviews from PC Magazine or Mac World to get an idea of what hardware is available and what is (and was) popular. Both magazines, as well as others, have annual reviews of printers, sound cards, display adapters and other peripherals.

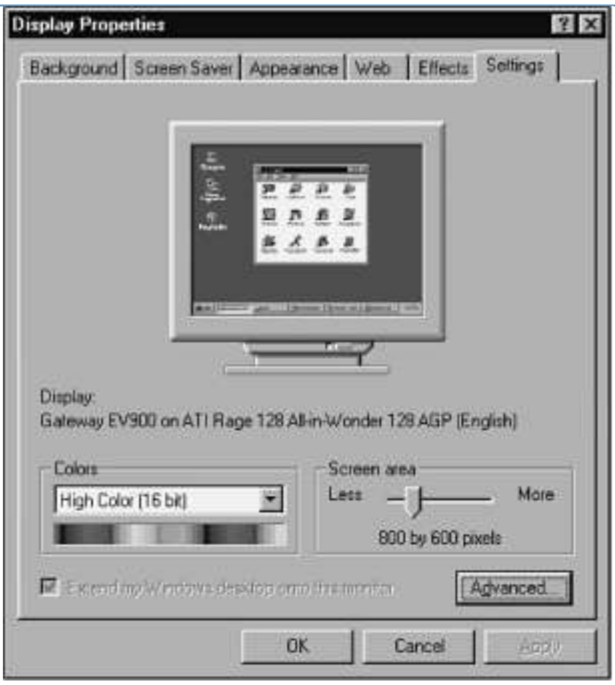
Do some research to see if some of the devices are clones of each other and therefore equivalent - falling under the same equivalence partition. For example, a printer manufacturer may license their printer to another company that puts a different cover and label on it. From your standpoint, it's the same printer.

Decide what device drivers you're going to test with. Your options are usually the drivers included with the operating system, the drivers included with the device, or the latest drivers available on the hardware or operating system company's website. Usually, all three are different. Ask yourself what customers have or what they can get.

DECIDE WHICH HARDWARE FEATURES, MODES, AND OPTIONS ARE POSSIBLE

Color printers can print in black and white or color, they can print in different quality modes, and can have settings for printing photos or text. Display cards, as shown in Figure 8.6, can have different color settings and screen resolutions.

FIGURE 8.6. THE DISPLAY PROPERTIES OF NUMBER OF COLORS AND SCREEN AREA ARE POSSIBLE CONFIGURATIONS FOR A DISPLAY CARD.



Display:
Gateway EV900 on ATI Rage 128 All-in-Wonder 128 AGP (English)

Colors: High Color (16 bit)

Screen area: Less — More, 800 by 600 pixels

Extend my Windows desktop onto this monitor.

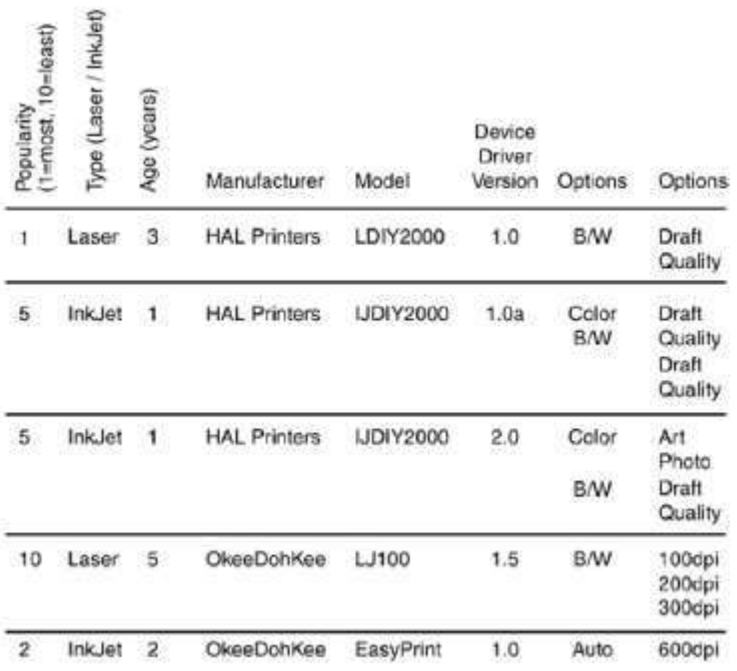
Every device has options, and your software may not need to support all of them. A good example of this is computer games. Many require a minimum number of display colors and resolution. If the configuration has less than that, they simply won't run.

PARE DOWN THE IDENTIFIED HARDWARE CONFIGURATIONS TO A MANAGEABLE SET

Given that you don't have the time or budget to test everything, you need to reduce the thousands of potential configurations into the ones that matter - the ones you're going to test.

One way to do this is to put all the configuration information into a spreadsheet with columns for the manufacturer, model, driver versions, and options. Figure 8.7 shows an example of a table that identifies various printer configurations. You and your team can review the chart and decide which configuration you want to test.

FIGURE 8.7. ORGANIZE YOUR CONFIGURATION INFORMATION INTO A SPREADSHEET.



Popularity (1=most, 10=least)	Type (Laser / InkJet)	Age (years)	Manufacturer	Model	Device Driver Version	Options	Options
1	Laser	3	HAL Printers	LDIY2000	1.0	B/W	Draft Quality
5	InkJet	1	HAL Printers	IJDIY2000	1.0a	Color B/W	Draft Quality Draft Quality
5	InkJet	1	HAL Printers	IJDIY2000	2.0	Color B/W	Art Photo Draft Quality
10	Laser	5	OkeeDohKee	LJ100	1.5	B/W	100dpi 200dpi 300dpi
2	InkJet	2	OkeeDohKee	EasyPrint	1.0	Auto	600dpi

Notice that Figure 8.7 also has columns for information about the device's popularity, its type, and its age. When creating your equivalence partitions, you might decide that you want to test only the most popular printers or ones that are less than five years old. With the type information - in this case, laser or inkjet - you could decide to test 75 percent lasers and 25 percent inkjets.

NOTE

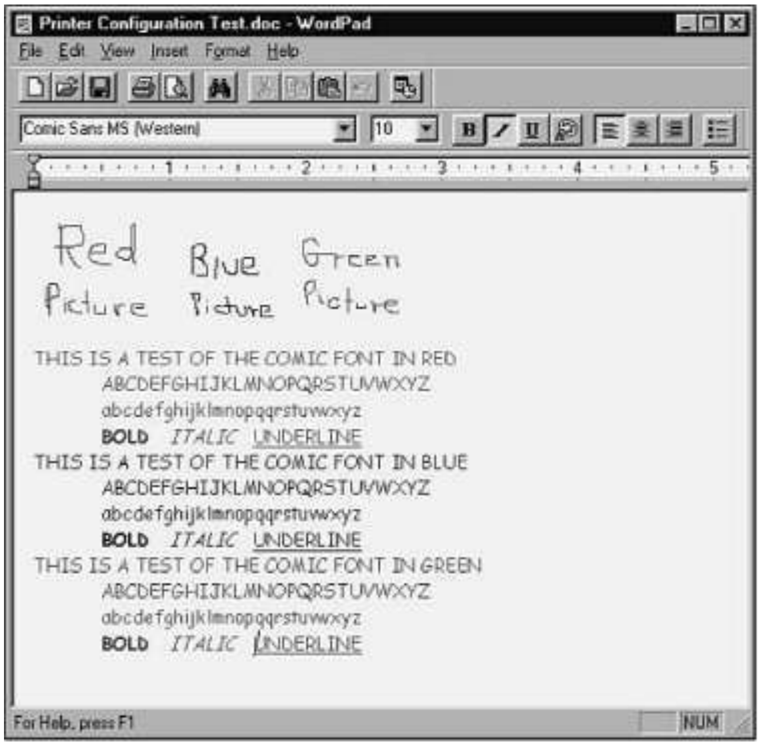
Ultimately, the decision-making process that you use to equivalence partition the configurations into smaller sets is up to you and your team. There is no right formula. Every software project is different and will have different selection criteria. Just make sure that everyone on the project team, especially your project manager, is aware of what configurations are being tested (and not tested) and what variables went into selecting them.

IDENTIFY YOUR SOFTWARE'S UNIQUE FEATURES THAT WORK WITH THE HARDWARE CONFIGURATIONS

The key word here is unique. You don't want to, nor do you need to, completely test your software on each configuration. You need to test only those features that are different from each other (different equivalence partitions) that interact with the hardware.

For example, if you're testing a word processor such as WordPad (see Figure 8.8), you don't need to test the file save and load feature in each configuration. File saving and loading has nothing to do with printing. A good test would be to create a document that contains different (selected by equivalence partitioning, of course) fonts, point sizes, colors, embedded pictures, and so on. You would then attempt to print this document on each chosen printer configuration.

FIGURE 8.8. YOU CAN USE A SAMPLE DOCUMENT MADE UP OF DIFFERENT FONTS AND STYLES TO CONFIGURATION TEST A PRINTER.


Printer Configuration Test.doc - WordPad
File Edit View Insert Format Help

Comic Sans MS (Western) 10

Red Blue Green
Picture Picture Picture

THIS IS A TEST OF THE COMIC FONT IN RED
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
BOLD *ITALIC* UNDERLINE
THIS IS A TEST OF THE COMIC FONT IN BLUE
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
BOLD *ITALIC* UNDERLINE
THIS IS A TEST OF THE COMIC FONT IN GREEN
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
BOLD *ITALIC* UNDERLINE

For Help, press F1 NUM

Selecting the unique features to try isn't as easy as it sounds. You should first make a black-box pass by looking at your product and pulling out the obvious ones. Then talk with others on your team, especially the programmers, to get a white-box view. You may be surprised at what features are even slightly tied to the configuration.

DESIGN THE TEST CASES TO RUN ON EACH CONFIGURATION

You'll learn the details of writing test cases in Chapter 18, "Writing and Tracking Test Cases," but, for now, consider that you'll need to write down the steps required to test each configuration. This can be as simple as

1.	Select and set up the next test configuration from the list.
2.	Start the software.
3.	Load in the file configtest.doc.
4.	Confirm that the file is displayed correctly.
5.	Print the document.
6.	Confirm that there are no error messages and that the printed document matches the standard.
7.	Log any discrepancies as a bug.

In reality, the steps would be much more involved, including more detail and specifics on exactly what to do and what to look for. The goal is to create steps that anyone can run. You'll learn more about writing test cases in Chapter 18.

EXECUTE THE TESTS ON EACH CONFIGURATION

You need to run the test cases and carefully log and report your results (see Chapter 19, "Reporting What You Find") to your team and to the hardware manufacturers if necessary. As described earlier in this chapter, it's often difficult and time-consuming to identify the specific source of configuration problems. You'll need to work closely with the programmers and white-box testers to isolate the cause and decide if the bugs you find are due to your software or to the hardware.

If the bug is specific to the hardware, consult the manufacturer's website for information on reporting problems to them. Be sure to identify yourself as a software tester and what company you work for. Many companies have separate staff set up to assist software companies writing software to work with their hardware. They may ask you to send copies of your test software, your test cases, and supporting details to help them isolate the problem.

RERUN THE TESTS UNTIL THE RESULTS SATISFY YOUR TEAM

It's not uncommon for configuration testing to run the entire course of a project. Initially a few configurations might be tried, then a full test pass, then smaller and smaller sets to confirm bug fixes. Eventually you will get to a point where there are no known bugs or to where the bugs that still exist are in uncommon or unlikely test configurations. At that point, you can call your configuration testing complete.

OBTAINING THE HARDWARE

One thing that hasn't been mentioned so far is where you obtain all this hardware. Even if you take great pains, and risk, to equivalence partition your configurations to the barest minimum, you still could require dozens of different hardware setups. It would be an expensive proposition to go out and buy everything at retail, especially if you will use the hardware only once for the one test pass. Here are a few ideas for overcoming this problem:

- Buy only the configurations that you can or will use most often. A great plan is for every tester on the team to have different hardware. This may drive your purchasing department and the group that maintains your company's PCs crazy (they like everyone to have exactly the same configuration) but it's a very efficient means of always having different configurations available to test on. Even if your test team is very small, three or four people having just a few configurations would be a great help.
- Contact the hardware manufacturers and ask if they will lend or even give you the hardware. If you explain that you're testing new software and you want to assure that it works on their hardware, many will do this for you. They have an interest in the outcome, too, so tell them that you'll furnish them with the results of the tests and, if you can, a copy of the finished software. It's good to build up these relationships, especially if you find a bug and need a contact person at the hardware company to report it to.
- Send a memo or email to everyone in your company asking what hardware they have in their office or even at home - and if they would allow you to run a few tests on it. To perform the configuration testing, you may need to drive around town, but it's a whole lot cheaper than attempting to buy all the hardware.

CONFIGURATION TESTING VCRS

The Microsoft ActiMates product line of animatronic dolls not only interfaced with a PC, but also a VCR. Coded commands, invisible to a viewer, were mixed in with the video on the tape.

A special box connected to the VCR decoded the commands and sent them by radio to the doll. The test team obviously needed to perform configuration testing on VCRs. They had many PC configurations but no VCRs.
They found two ways to get the job done:

- They asked about 300 employees to bring in their VCRs for a day of testing. The program manager awarded gift certificates as a means of persuading people to bring them in.
- They paid the manager of a local electronics superstore to stay at the store after hours (actually, all night) while they pulled each VCR off the shelf, connected their equipment, and ran the tests. They dusted and cleaned the VCRs and bought the manager dinner to show their thanks.

When it was all over, they had tested about 150 unique VCRs, which they determined was a very good equivalence partition of the VCRs in people's homes.

- If you have the budget, work with your project manager to contract out your test work to a professional configuration and compatibility test lab. These companies do nothing but configuration testing and have every piece of PC hardware known to man. Okay, maybe not that much, but they do have a lot.

  These labs can help you, based on their experience, select the correct hardware to test. Then, they will allow you to come in and use their equipment, or they will provide a complete turn-key service. You provide the software, the step-by-step test process, and the expected results. They'll take it from there, running the tests and reporting what passed and what failed. Of course this can be costly, but much less so than buying the hardware yourself or worse, not testing and having customers find the problems.

IDENTIFYING HARDWARE STANDARDS

If you're interested in performing a little static black-box analysis - that is, reviewing the specifications that the hardware companies use to create their products - you can look in a couple of places. Knowing some details of the hardware specifications can help you make more informed equivalence partition decisions.
For Apple hardware, visit the Apple Hardware website at developer.apple.com/hardware. There you'll find information and links about developing and testing hardware and device drivers for Apple computers. Another Apple link, developer.apple.com/testing, points you to specific testing information, including links to test labs that perform configuration testing.
For PCs, the best link is www.microsoft.com/whdc/system/platform. This site provides technical implementation guidelines, tips, and tools for developers and testers developing hardware for use with Windows.
Microsoft also publishes a set of standards for software and hardware to receive the Windows logo. That information is at msdn.microsoft.com/certification and www.microsoft.com/whdc/whql.

CONFIGURATION TESTING OTHER HARDWARE

So, what if you're not testing software that runs on a PC or a Mac? Was this chapter a waste of your time? No way! Everything you learned can be applied to testing generic or proprietary systems, too. It doesn't matter what the hardware is or what it's connected to; if it can have variations such as memory size, CPU speed, etc, or, if it connects to another piece of hardware, software configuration issues need to be tested.

If you're testing software for an industrial controller, a network, medical devices, or a phone system, ask yourself the same questions that you would if you were testing software for a desktop computer:

- What external hardware will operate with this software?
- What models and versions of that hardware are available?
- What features or options does that hardware support?

Create equivalence partitions of the hardware based on input from the people who work with the equipment, your project manager, or your sales people. Develop test cases, collect the selected hardware, and run the tests. Configuration testing follows the same testing techniques that you've already learned.

SUMMARY

This chapter got you thinking about how to approach configuration testing. It's a job that new software testers are frequently assigned because it is easily defined, is a good introduction to basic organization skills and equivalence partitioning, is a task that will get you working with lots of other project team members, and is one for which your manager can readily verify the results. The downside is that it can become overwhelming.
If you're assigned to perform configuration testing for your project, take a deep breath, reread this chapter, carefully plan your work, and take your time. When you're done, your boss will have another job for you: compatibility testing, the subject of the next chapter.

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. What's the difference between a component and a peripheral?
2. How can you tell if a bug you find is a general problem or a specific configuration problem?
3. How could you guarantee that your software would never have a configuration problem?
4. Some companies purchase generic hardware and put their names on it, selling it as their own. You'll often see this on lower-priced peripherals sold in computer superstores. The same "cloned" peripheral might be sold under different names in different stores. True or False: Only one version of a cloned sound card needs to be considered when selecting the configurations to test.
5. In addition to age and popularity, what other criteria might you use to equivalence partition hardware for configuration testing?
6. Is it acceptable to release a software product that has configuration bugs?

CHAPTER 9. COMPATIBILITY TESTING

IN THIS CHAPTER

- Compatibility Testing Overview
- Platform and Application Versions
- Standards and Guidelines
- Data Sharing Compatibility

In Chapter 8, "Configuration Testing," you learned about hardware configuration testing and how to assure that software works properly with the hardware it was designed to run on and connect with. This chapter

deals with a similar area of interaction testing - checking that your software operates correctly with other software.
Testing whether one program plays well with others has become increasingly important as consumers demand the ability to share data among programs of different types and from different vendors and take advantage of the ability to run multiple programs at once.
It used to be that a program could be developed as a standalone application. It would be run in a known, understood, benign environment, isolated from anything that could corrupt it. Today, that program likely needs to import and export data to other programs, run with different operating systems and Web browsers, and interoperate with other software being run simultaneously on the same hardware. The job of software compatibility testing is to make sure that this interaction works as users would expect.
The highlights of this chapter include

- What it means for software to be compatible
- How standards define compatibility
- What platforms are and what they mean for compatibility
- Why being able to transfer data among software applications is the key to compatibility

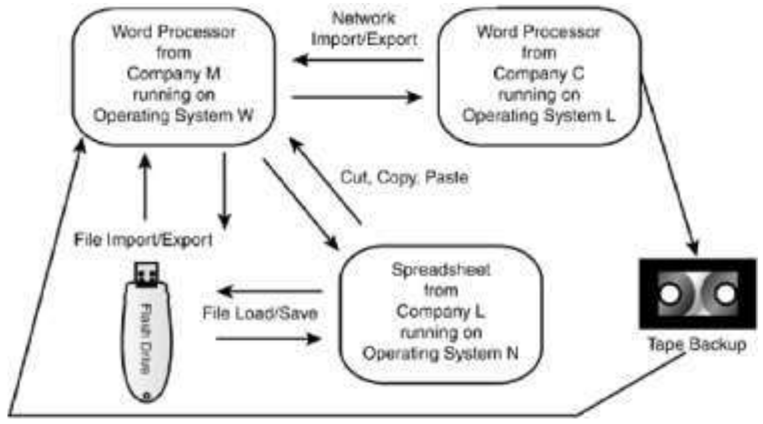
COMPATIBILITY TESTING OVERVIEW

Software compatibility testing means checking that your software interacts with and shares information correctly with other software. This interaction could occur between two programs simultaneously running on the same computer or even on different computers connected through the Internet thousands of miles apart. The interaction could also be as simple as saving data to a floppy disk and hand-carrying it to another computer across the room.
Examples of compatible software are

- Cutting text from a web page and pasting it into a document opened in your word processor
- Saving accounting data from one spreadsheet program and then loading it into a completely different spreadsheet program
- Having photograph touchup software work correctly on different versions of the same operating system
- Having your word processor load in the names and addresses from your contact management program and print out personalized invitations and envelopes
- Upgrading to a new database program and having all your existing databases load in and work just as they did with the old program

What compatibility means for your software depends on what your team decides to specify and what levels of compatibility are required by the system that your software will run on. Software for a standalone medical device that runs its own operating system, stores its data on its own memory cartridges, and doesn't connect to any other device would have no compatibility considerations. However, the fifth version of a word processor (see Figure 9.1) that reads and writes different files from other word processors, allows multiuser editing over the Internet, and supports inclusion of embedded pictures and spreadsheets from various applications has a multitude of compatibility considerations.

FIGURE 9.1. COMPATIBILITY ACROSS DIFFERENT SOFTWARE APPLICATIONS CAN QUICKLY BECOME VERY COMPLICATED.


Word Processor from Company M running on Operating System W
Network Import/Export
Word Processor from Company C running on Operating System L
Cut, Copy, Paste
File Import/Export
Flash Drive
File Load/Save
Spreadsheet from Company L running on Operating System N
Tape Backup

If you're assigned the task of performing software compatibility testing on a new piece of software, you'll need to get the answers to a few questions:

- What other platforms (operating system, web browser, or other operating environment) and other application software is your software designed to be compatible with? If the software you're testing is a platform, what applications are designed to run under it?
- What compatibility standards or guidelines should be followed that define how your software should interact with other software?
- What types of data will your software use to interact and share information with other platforms and software?

Gaining the answers to these questions is basic static testing - both black-box and white-box. It involves thoroughly analyzing the specification for the product and any supporting specifications. It could also entail discussions with the programmers and possibly close review of the code to assure that all links to and from your software are identified. The rest of this chapter discusses these questions in more detail.

PLATFORM AND APPLICATION VERSIONS

Selecting the target platforms or the compatible applications is really a program management or a marketing task. Someone who's very familiar with the customer base will decide whether your software is to be designed for a specific operating system, web browser, or some other platform. They'll also identify the version or versions that the software needs to be compatible with. For example, you've probably seen notices such as these on software packages or startup screens:

Works best with AOL 9.0

Requires Windows XP or greater

For use with Linux 2.6.10

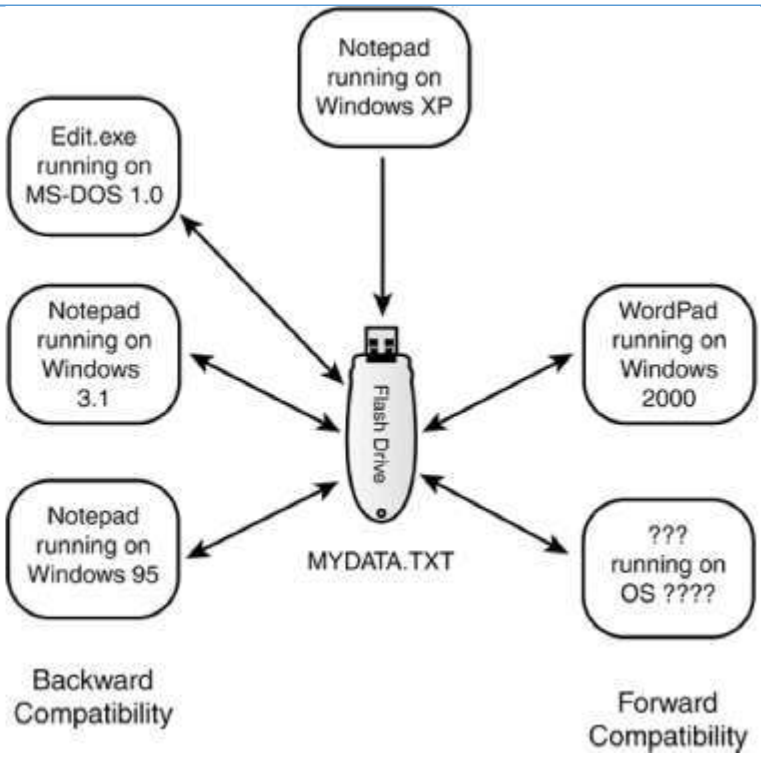
This information is part of the specification and tells the development and test teams what they're aiming for. Each platform has its own development criteria and it's important, from a project management standpoint, to make this platform list as small as possible but still fill the customer's needs.

BACKWARD AND FORWARD COMPATIBILITY

Two terms you'll hear regarding compatibility testing are backward compatible and forward compatible. If something is backward compatible, it will work with previous versions of the software. If something is forward compatible, it will work with future versions of the software.

The simplest demonstration of backward and forward compatibility is with a .txt or text file. As shown in Figure 9.2, a text file created using Notepad 98 running under Windows 98 is backward compatible all the way back to MS-DOS 1.0. It's also forward compatible to Windows XP service pack 2 and likely will be beyond that.

FIGURE 9.2. BACKWARD AND FORWARD COMPATIBILITY DEFINE WHAT VERSIONS WILL WORK WITH YOUR SOFTWARE OR DATA FILES.

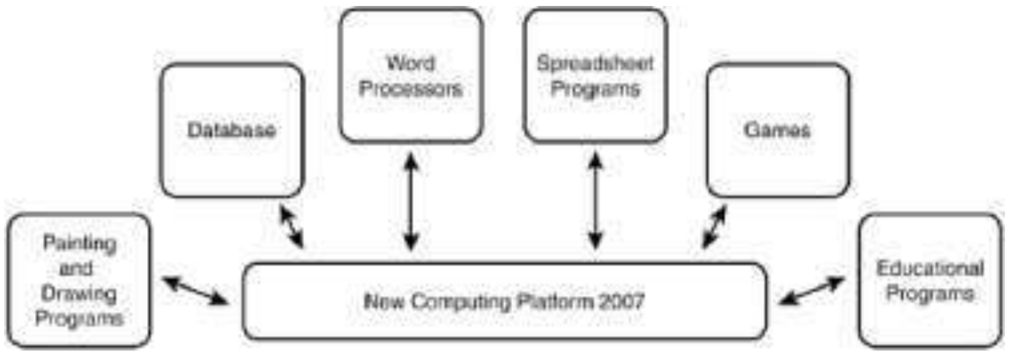

Notepad running on Windows XP
Edit.exe running on MS-DOS 1.0
Notepad running on Windows 3.1
WordPad running on Windows 2000
Flash Drive
Notepad running on Windows 95
MYDATA.TXT
??? running on OS ????
Backward Compatibility
Forward Compatibility

NOTE
It's not a requirement that all software or files be backward or forward compatible. That's a product feature decision your software designers need to make. You should, though, provide input on how much testing will be required to check forward and backward compatibility for the software.

THE IMPACT OF TESTING MULTIPLE VERSIONS

Testing that multiple versions of platforms and software applications work properly with each other can be a huge task. Consider the situation of having to compatibility test a new version of a popular operating system. The programmers have made numerous bug fixes and performance improvements and have added many new features to the code. There could be tens or hundreds of thousands of existing programs for the current versions of the OS. The project's goal is to be 100 percent compatible with them. See Figure 9.3.

FIGURE 9.3. IF YOU COMPATIBILITY TEST A NEW PLATFORM, YOU MUST CHECK THAT EXISTING SOFTWARE APPLICATIONS WORK CORRECTLY WITH IT.



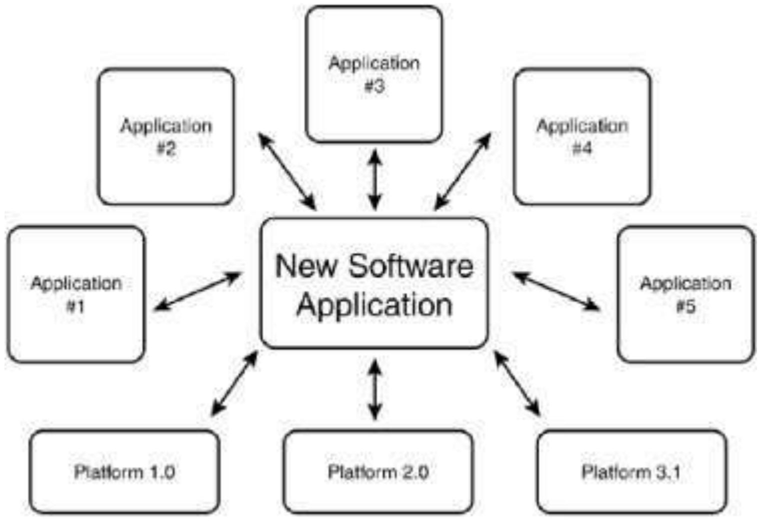
This is a big job, but it's just another example of how equivalence partitioning can be applied to reduce the amount of work.
NOTE
To begin the task of compatibility testing, you need to equivalence partition all the possible software combinations into the smallest, effective set that verifies that your software interacts properly with other software.

In short, you can't test all the thousands of software programs on your operating system, so you need to decide which ones are the most important to test. The key word is important. The criteria that might go into deciding what programs to choose could be

- Popularity. Use sales data to select the top 100 or 1,000 most popular programs.
- Age. You might want to select programs and versions that are less than three years old.
- Type. Break the software world into types such as painting, writing, accounting, databases, communications, and so on. Select software from each category for testing.
- Manufacturer. Another criteria would be to pick software based on the company that created it.

Just as in hardware configuration testing, there is no right "textbook" answer. You and your team will need to decide what matters most and then use that criteria to create equivalence partitions of the software you need to test with.
The previous example dealt with compatibility testing a new operating system platform. The same issues apply to testing a new application (see Figure 9.4). You need to decide what platform versions you should test your software on and what other software applications you should test your software with.

FIGURE 9.4. COMPATIBILITY TESTING A NEW APPLICATION MAY REQUIRE YOU TO TEST IT ON MULTIPLE PLATFORMS AND WITH MULTIPLE APPLICATIONS.



STANDARDS AND GUIDELINES

So far in this chapter you've learned about selecting the software that you'll compatibility test with your program. Now, it's time to look at how you'll approach the actual testing. Your first stop should be researching the existing standards and guidelines that might apply to your software or the platform.
There are really two levels of these requirements: high-level and low-level. It may be a misnomer to refer to them as high and low, but in a sense, that's really what they are. High-level standards are the ones that guide your product's general operation, its look and feel, its supported features, and so on. Low-level standards are the nitty-gritty details, such as the file formats and the network communications protocols. Both are important and both need to be tested to assure compatibility.

HIGH-LEVEL STANDARDS AND GUIDELINES

Will your software run under Windows, Mac, or Linux operating systems? Is it a web application? If so, what browsers will it run on? Each of these is considered a platform and most have their own set of standards and guidelines that must be followed if an application is to claim that it's compatible with the platform.
An example of this is the Certified for Microsoft Windows logo (see Figure 9.5). To be awarded this logo, your software must undergo and pass compatibility testing by an independent testing laboratory. The goal is to assure that the software runs stably and reliably on the operating system.

FIGURE 9.5. THE CERTIFIED FOR MICROSOFT WINDOWS LOGO SIGNIFIES THAT THE SOFTWARE MEETS ALL THE CRITERIA DEFINED BY THE GUIDELINES.

A few examples of the logo requirements are that the software

- Supports mice with more than three buttons
- Supports installation on disk drives other than C: and D:
- Supports filenames longer than the DOS 8.3 format

- Doesn't read, write, or otherwise use the old system files win.ini, system.ini, autoexec.bat, or config.sys

These may sound like simple, matter-of-fact requirements, but they're only four items out of a 100+ page document. It's a great deal of work to assure that your software complies with all the logo requirements, but it makes for much more compatible software.
NOTE
The details of the Windows logo can be obtained at msdn.microsoft.com/certification. Details for using the Apple Mac logo are at developer.apple.com/testing.

LOW-LEVEL STANDARDS AND GUIDELINES

Low-level standards are, in a sense, more important than the high-level standards. You could create a program that would run on Windows that didn't have the look and feel of other Windows software. It wouldn't be granted the Certified for Microsoft Windows logo. Users might not be thrilled with the differences from other applications, but they could use the product.

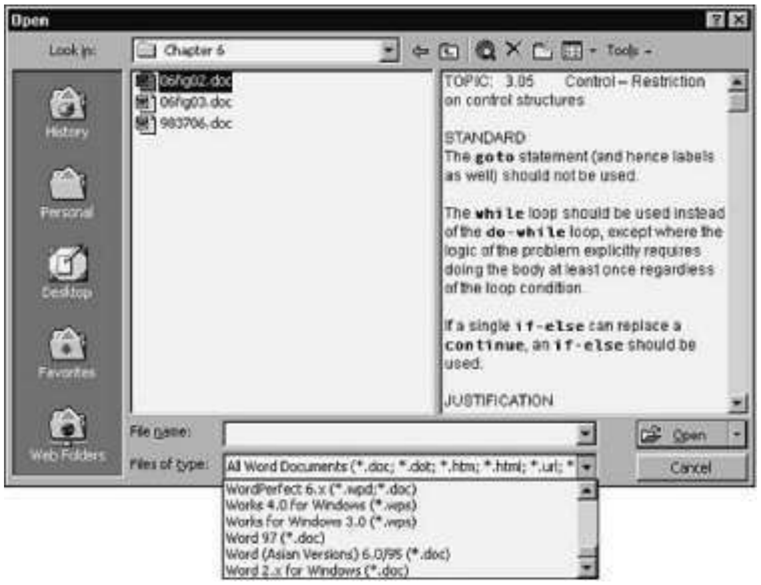
If, however, your software is a graphics program that saves its files to disk as .pict files (a standard Macintosh file format for graphics) but the program doesn't follow the standard for .pict files, your users won't be able to view the files in any other program. Your software wouldn't be compatible with the standard and would likely be a short-lived product.
Similarly, communications protocols, programming language syntax, and any means that programs use to share information must adhere to published standards and guidelines.
These low-level standards are often taken for granted, but from a tester's perspective must be tested. You should treat low-level compatibility standards as an extension of the software's specification. If the software spec states, "The software will save and load its graphics files as .bmp, .jpg, and .gif formats," you need to find the standards for these formats and design tests to confirm that the software does indeed adhere to them.
NOTE
Don't necessarily trust your team's interpretation of the standards or guidelines. Look them up yourself and develop your tests directly from the source. Remember the difference between precision and accuracy? You don't want your product's compatibility code to be perfectly precise, but totally inaccurate.

DATA SHARING COMPATIBILITY

The sharing of data among applications is what really gives software its power. A well-written program that supports and adheres to published standards and allows users to easily transfer data to and from other software is a great compatible product.
The most familiar means of transferring data from one program to another is saving and loading disk files. As discussed in the previous section, adhering to the low-level standards for the disk and file formats is what makes this sharing possible. Other means, though, are sometimes taken for granted but still need to be tested for compatibility. Here are a few examples:

- File save and file load are the data-sharing methods that everyone is aware of. You save your data to a floppy disk (or some other means of magnetic or optical storage) and then hand carry it over to another computer running different software and load it in. The data format of the files needs to meet standards for it to be compatible on both computers.
- File export and file import are the means that many programs use to be compatible with older versions of themselves and with other programs. Figure 9.6 shows the Microsoft Word File Open dialog box and some of the 23 different file formats that can be imported into the word processor.

FIGURE 9.6. MICROSOFT WORD CAN IMPORT 23 DIFFERENT FILE FORMATS.



06fig02.doc
06fig03.doc
983706.doc

TOPIC: 3.05 Control – Restriction on control structures

STANDARD
The **goto** statement (and hence labels as well) should not be used.

The **while** loop should be used instead of the **do-while** loop, except where the logic of the problem explicitly requires doing the body at least once regardless of the loop condition.

If a single **if-else** can replace a **continue**, an **if-else** should be used.

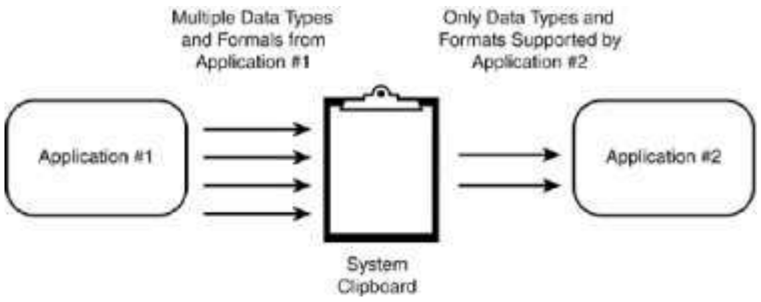
JUSTIFICATION

All Word Documents (*.doc; *.dot; *.htm; *.html; *.url; *
WordPerfect 6.x (*.wpd;*.doc)
Works 4.0 for Windows (*.wps)
Works for Windows 3.0 (*.wps)
Word 97 (*.doc)
Word (Asian Versions) 6.0/95 (*.doc)
Word 2.x for Windows (*.doc)

To test the file import feature, you would need to create test documents in each compatible file format - probably using the original software that wrote that format. Those documents would need to have equivalence partitioned samples of the possible text and formatting to check that the importing code properly converts it to the new format.

- Cut, copy, and paste are the most familiar methods for sharing data among programs without transferring the data to a disk. In this case, the transfer happens in memory through an intermediate program called the Clipboard. Figure 9.7 shows how this transfer occurs.

FIGURE 9.7. THE SYSTEM CLIPBOARD IS A TEMPORARY HOLDING PLACE FOR DIFFERENT TYPES OF DATA THAT'S BEING COPIED FROM ONE APPLICATION TO ANOTHER.


Multiple Data Types and Formats from Application #1
Only Data Types and Formats Supported by Application #2
Application #1
Application #2
System Clipboard

The Clipboard is designed to hold several different data types. Common ones in Windows are text, pictures, and sounds. These data types can also be different formats - for example, the text can be plain old text, HTML, or rich text. Pictures can be bitmaps, metafiles, or .tifs.

Whenever a user performs a cut or copy, the data that's chosen is placed in the Clipboard. When he does a paste, it's copied from the Clipboard to the destination software. Some applications may only accept certain data types or formats being pasted into them - for example, a painting program may accept pictures, but not text.

If you're compatibility testing a program, you need to make sure that its data can be properly copied in and out of the Clipboard to other programs. This feature is so transparent and so frequently used, people forget that there's a lot of code behind making sure that it works and is compatible across lots of different software.

- DDE (pronounced D-D-E), COM (for Component Object Model), and OLE (pronounced oh-lay) are the methods in Windows of transferring data between two applications. DDE stands for Dynamic Data Exchange and OLE stands for Object Linking and Embedding. Other platforms support similar methods.

  There's no need to get into the gory details of these technologies in this book, but the primary difference between these two methods and the Clipboard is that with DDE and OLE data can flow from one application to the other in real time. Cutting and copying is a manual operation. With DDE and OLE, the transfer can happen automatically.

  An example of how these might be used could be a written report done in a word processor that has a pie-chart created by a spreadsheet program. If the report's author copied and pasted the chart into the report, it would be a snapshot in time of the data. If, however, the author linked the pie chart into the report as an object, when the underlying numbers for the chart change, the new graphics will automatically appear in the report.

  This is all pretty fancy, yes, but it's also a testing challenge to make sure that all the object linking, embedding, and data exchanging happens correctly.

SUMMARY

This chapter introduced you to the basics of compatibility testing. In reality, an entire book could be written on the subject, and a single chapter doesn't do the topic justice. Every platform and every application is unique, and the compatibility issues on one system can be totally different than on another. As a new software tester, you may be assigned a task of compatibility testing your software. That may seem strange, given that it's potentially such a large and complex task, but you'll likely be assigned just a piece of the entire job. If your project is a new operating system, you may be asked to compatibility test just word processors or graphics programs. If your project is an applications program, you may be asked to compatibility test it on several different platforms.
Each is a manageable task that you can easily handle if you approach your testing with these three things in mind:

- Equivalence partition all the possible choices of compatible software into a manageable set. Of course, your project manager should agree with your list and understand the risk involved in not testing everything.
- Research the high-level and low-level standards and guidelines that might apply to your software. Use these as extensions of your product's specification.
- Test the different ways that data can flow between the software programs you're testing. This data exchange is what makes one program compatible with another.

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. True or False: All software must undergo some level of compatibility testing.
2. True or False: Compatibility is a product feature and can have different levels of compliance.
3. If you're assigned to test compatibility of your product's data file formats, how would you approach the task?
4. How can you test forward compatibility?

CHAPTER 10. FOREIGN-LANGUAGE TESTING

IN THIS CHAPTER

- Making the Words and Pictures Make Sense
- Translation Issues
- Localization Issues
- Configuration and Compatibility Issues
- How Much Should You Test?

Si eres fluente en más de un idioma y competente probando programas de computadora, usted tiene una habilidad muy deseada en el mercado.
Wenn Sie eine zuverläßig Software Prüferin sind, und fließend eine fremd sprache, ausser English, sprechen können, dann können Sie gut verdienen.
Translated roughly from Spanish and German, the preceding two sentences read: If you are a competent software tester and are fluent in a language other than English, you have a very marketable skill set.
Most software today is released to the entire world, not just to a certain country or in a specific language. Microsoft shipped Windows XP with support for 106 different languages and dialects, from Afrikaans to Hungarian to Zulu. Most other software companies do the same, realizing that the U.S. English market is less than half of their potential customers. It makes business sense to design and test your software for worldwide distribution.
This chapter covers what's involved in testing software written for other countries and languages. It might seem like a straightforward process, but it's not, and you'll learn why.
Highlights of this chapter include

- Why just translating is not enough
- How words and text are affected
- Why footballs and telephones are important
- The configuration and compatibility issues
- How large of a job testing another language is

MAKING THE WORDS AND PICTURES MAKE SENSE

Have you ever read a user's manual for an appliance or a toy that was poorly converted word for word from another language? "Put in a bolt number five past through green bar and tighten no loose to nut." Got it?
That's a poor translation, and it's what software can look like to a non-English speaker if little effort is put into building the software for foreign languages. It's easy to individually convert all the words, but to make the overall instructions meaningful and useful requires much more work and attention.
Good translators can do that. If they're fluent in both languages, they can make the foreign text read as well as the original. Unfortunately, what you'll find in the software industry is that even a good translation isn't sufficient.

Take Spanish, for example. It should be a simple matter to convert English text to Spanish, right? Well, which Spanish are you referring to? Spanish from Spain? What about Spanish from Costa Rica, Peru, or the Dominican Republic? They're all Spanish, but they're different enough that software written for one might not be received well by the others. Even English has this problem. There's not just American English, there's also Canadian, Australian, and British English. It would probably seem strange to you to see the words colour, neighbour, and rumour in your word processor.
What needs to be accounted for, besides the language, is the region or locale - the user's country or geographic area. The process of adapting software to a specific locale, taking into account its language, dialect, local conventions, and culture, is called localization or sometimes internationalization. Testing the software is called localization testing.

TRANSLATION ISSUES

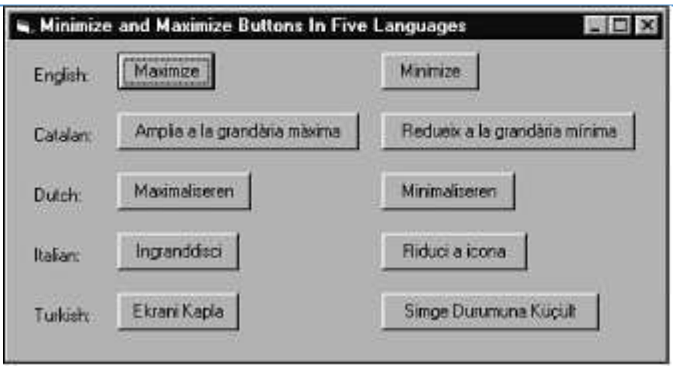
Although translation is just a part of the overall localization effort, it's an important one from a test standpoint. The most obvious problem is how to test something that's in another language. Well, you or someone on your test team will need to be at least semi-fluent in the language you're testing, being able to navigate the software, read any text it displays, and type the necessary commands to run your tests. It might be time to sign up for the community college course in Slovenian you always wanted to take.
NOTE
It's important that you or someone on your test team be at least a little familiar with the language you're testing. Of course, if you're shipping your program in 32 different languages, they may be difficult. The solution is to contract out this work to a localization testing company. Numerous such companies worldwide can perform testing in nearly any language. For more information, search the Internet for "localization testing."

It's not a requirement that everyone on the test team speak the language that the software is being localized into; you probably need just one person. Many things can be checked without knowing what the words say. It would be helpful, sure, to know a bit of the language, but you'll see that you might be able to do a fair amount of the testing without being completely fluent.

TEXT EXPANSION

The most straightforward example of a translation problem that can occur is due to something called text expansion. Although English may appear at times to be wordy, it turns out that when English is translated into other languages, often more characters are necessary to say the same thing. Figure 10.1 shows how the size of a button needs to expand to hold the translated text of two common computer words. A good rule of thumb is to expect up to 100 percent increase in size for individual words - on a button, for example. Expect a 50 percent increase in size for sentences and short paragraphs - typical phrases you would see in dialog boxes and error messages.

FIGURE 10.1. WHEN TRANSLATED INTO OTHER LANGUAGES, THE WORDS MINIMIZE AND MAXIMIZE CAN VARY GREATLY IN SIZE OFTEN FORCING THE UI TO BE REDESIGNED TO ACCOMMODATE THEM.


Minimize and Maximize Buttons In Five Languages
English: Maximize Minimize
Catalan: Amplia a la grandària màxima Redueix a la grandària mínima
Dutch: Maximaliseren Minimaliseren
Italian: Ingranddisci Riduci a icona
Turkish: Ekrani Kapla Simge Durumuna Küçült

Because of this expansion, you need to carefully test areas of the software that could be affected by longer text. Look for text that doesn't wrap correctly, is truncated, or is hyphenated incorrectly. This could occur anywhere - onscreen, in windows, boxes, buttons, and so on. Also look for cases where the text had enough room to expand, but did so by pushing something else out of the way.
Another possibility is that this longer text can cause a major program failure or even a system crash. A programmer could have allocated enough internal memory for the English text messages, but not enough for the translated strings. The English version of the software will work fine but the German version will crash when the message is displayed. A white-box tester could catch this problem without knowing a single word of the language.

ASCII, DBCS, AND UNICODE

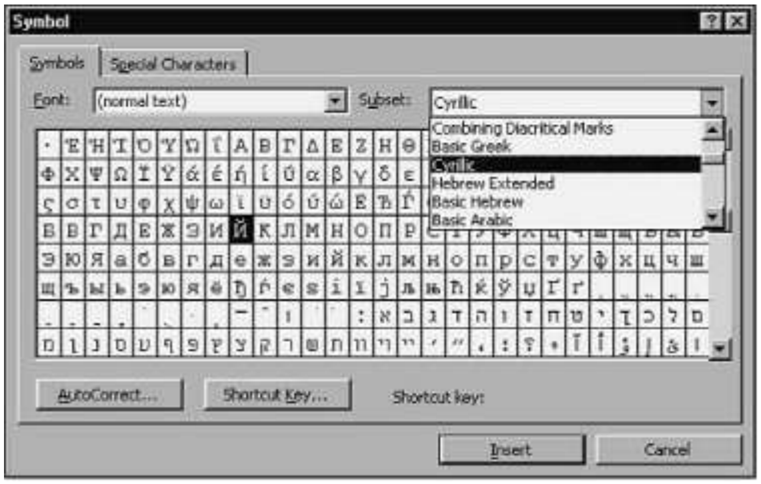
Chapter 5, "Testing the Software with Blinders On," briefly discussed the ASCII character set. ASCII can represent only 256 different characters - not nearly enough to represent all the possible characters in all languages. When software started being developed for different languages, solutions needed to be found to overcome this limitation. An approach common in the days of MS-DOS, but still in use today, is to use a technique called code pages. Essentially, a code page is a replacement ASCII table, with a different code page for each language. If your software runs in Quebec on a French PC, it could load and use a code page that supports French characters. Russian uses a different code page for its Cyrillic characters, and so on.
This solution is fine, although a bit clunky, for languages with less than 256 characters, but Japanese, Chinese, and other languages with thousands of symbols cause problems. A system called DBCS (for Double-Byte Character Set) is used by some software to provide more than 256 characters. Using 2 bytes instead of 1 byte allows for up to 65,536 different characters.
Code pages and DBCS are sufficient in many situations but suffer from a few problems. Most important is the issue of compatibility. If a Hebrew document is loaded onto a German computer running a British word processor, the result can be gibberish. Without the proper code pages or the proper conversion from one to the other, the characters can't be interpreted correctly, or even at all.
The solution to this mess is the Unicode standard.

Unicode provides a unique number for every character,
no matter what the platform,
no matter what the program,
no matter what the language.

"What is Unicode?" from the Unicode Consortium website, www.unicode.org

Because Unicode is a worldwide standard supported by the major software companies, hardware manufacturers, and other standards groups, it's becoming more commonplace. Most major software applications support it. Figure 10.2 shows many of the different characters supported. If it's at all possible that your software will ever be localized, you and the programmers on your project should cut your ties to "ol' ASCII" and switch to Unicode to save yourself time, aggravation, and bugs.

FIGURE 10.2. THIS MICROSOFT WORD DIALOG SHOWS SUPPORT FOR THE UNICODE STANDARD.



HOT KEYS AND SHORTCUTS

In English, it's Search. In French, it's Réchercher. If the hotkey for selecting Search in the English version of your software is Alt+S, that will need to change in the French version.

In localized versions of your software, you'll need to test that all the hotkeys and shortcuts work properly and aren't too difficult to use - for example, requiring a third keypress. And, don't forget to check that the English hotkeys and shortcuts are disabled.

EXTENDED CHARACTERS

A common problem with localized software, and even non-localized software, is in its handling of extended characters. Referring back to that ancient ASCII table, extended characters are the ones that fall outside the normal English alphabet of AZ and az. Examples of these would be the accented characters such as the é in José or the ñ in El Niño. They also include the many symbol characters such as

that aren't on your typical keyboard. If your software is properly written to use Unicode or even if it correctly manages code pages or DBCS, this shouldn't be an issue, but a tester should never assume anything, so it's worthwhile to check.

The way to test this is to look for all the places that your software can accept character input or send output. In each place, try to use extended characters to see if they work just as regular characters would. Dialog boxes, logins, and any text fields are fair game. Can you send and receive extended characters through a modem? Can you name your files with them or even have the characters in the files? Will they print out properly? What happens if you cut, copy, and paste them between your program and another one?

TIP

The simplest way to ensure that you test for proper handling of extended characters is to add them to your equivalence partition of the standard characters that you test. Along with those bug-prone characters sitting on the ASCII table boundaries, throw in an Æ, an Ø and a ß.

COMPUTATIONS ON CHARACTERS

Related to extended characters are problems with how they're interpreted by software that performs calculations on them. Two examples of this are word sorting and upper- and lowercase conversion.

Does your software sort or alphabetize word lists? Maybe in a list box of selectable items such as filenames or website addresses? If so, how would you sort the following words?

Kopiëren	Reiste
Ärmlich	Arg
Reiskorn	résumé

Kopiëren	Reiste
Reißaus	kopieën
reiten	Reisschnaps
reißen	resume

If you're testing software to be sold to one of the many Asian cultures, are you aware that the sort order is based on the order of the brush strokes used to paint the character? The preceding list would likely have a completely different sort order if written in Mandarin Chinese. Find out what the sorting rules are for the language you're testing and develop tests to specifically check that the proper sort order occurs.
The other area where calculation on extended characters breaks down is with upper- and lowercase conversion. It's a problem because the "trick" solution that many programmers learn in school is to simply add or subtract 32 to the ASCII value of the letter to convert it between cases. Add 32 to the ASCII value of A and you get the ASCII value of a. Unfortunately, that doesn't work for extended characters. If you tried this technique using the Apple Mac extended character set, you'd convert Ñ (ASCII 132) to § (ASCII 164) instead of ñ (ASCII 150)not exactly what you'd expect.
Sorting and alphabetizing are just two examples. Carefully look at your software to determine if there are other situations where calculations are performed on letters or words. Spell-checking perhaps?

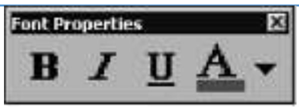
READING LEFT TO RIGHT AND RIGHT TO LEFT

A huge issue for translation is that some languages, such as Hebrew and Arabic, read from right to left, not left to right. Imagine flipping your entire user interface into a mirror image of itself.
Thankfully, most major operating systems provide built-in support for handling these languages. Without this, it would be a nearly impossible task. Even so, it's still not a simple matter of translating the text. It requires a great deal of programming to make use of the OS's features to do the job. From a testing standpoint, it's probably safe to consider it a completely new product, not just a localization.

TEXT IN GRAPHICS

Another translation problem occurs when text is used in graphics. See Figure 10.3 for several examples.

FIGURE 10.3. WORD 2000 HAS EXAMPLES OF TEXT IN BITMAPS THAT WOULD BE DIFFICULT TO TRANSLATE.



The icons in Figure 10.3 are the standard ones for selecting Bold, Italic, Underline, and Font Color. Since they use the English letters B, I, U, and A, they'll mean nothing to someone from Japan who doesn't read English. They might pick up on the meaning based on their look - the B is a bit dark, the I is leaning, and the U has a line under it - but software isn't supposed to be a puzzle.
The impact of this is that when the software is localized, each icon will have to be changed to reflect the new languages. If there were many of these icons, it could get prohibitively expensive to localize the program. Look for text-in-graphic bugs early in the development cycle so they don't make it through to the end.

KEEP THE TEXT OUT OF THE CODE

The final translation problem to cover is a white-box testing issue - keep the text out of the code. What this means is that all text strings, error messages, and really anything that could possibly be translated should be stored in a separate file independent of the source code. You should never see a line of code such as:

```
Print "Hello World"
```

Most localizers are not programmers, nor do they need to be. It's risky and inefficient to have them modifying the source code to translate it from one language to another. What they should modify is a simple text file, called a resource file, that contains all the messages the software can display. When the software runs, it references the messages by looking them up, not knowing or caring what they say. If the message is in English or Dutch, it gets displayed just the same.
That said, it's important for white-box testers to search the code to make sure there are no embedded strings that weren't placed in the external file. It would be pretty embarrassing to have an important error message in a Spanish program appear in English.
Another variation of this problem is when the code dynamically generates a text message. For example, it might piece together snippets of text to create a larger message. The code could take three strings:

1. "You pressed the"
2. a variable string containing the name of the key just pressed
3. "key just in time!"

and put them together to create a message. If the variable string had the value "stop nuclear reaction," the total message would read:

> You pressed the stop nuclear reaction key just in time!

The problem is that the word order is not the same in all languages. Although it pieces together nicely in English, with each phrase translated separately, it could be gibberish when stuck together in Mandarin Chinese or even German. Don't let strings crop into the code and don't let them be built up into larger strings by the code.

LOCALIZATION ISSUES

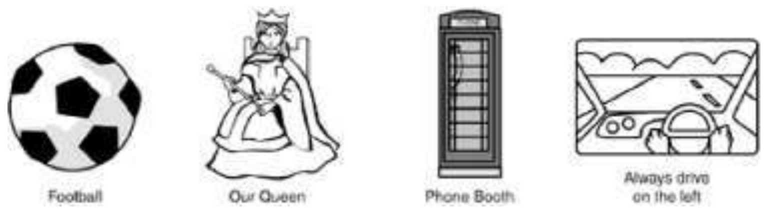
As mentioned previously, translation issues are only half the problem. Text can easily be translated and allowances made for different characters and lengths of strings. The difficulty occurs in changing the software so that it's appropriate for the foreign market.
REMINDER
Remember those terms from Chapter 3, "The Realitites of Software Testing": precision, accuracy, and reliability and quality?

Well translated and tested software is precise and reliable, but, if the programmers don't consider localization issues, it's probably not accurate or of high quality. It might look and feel great, read perfectly, and never crash, but to someone from another locale, it might just seem plain-old wrong. Assuring that the product is correctly localized gets you to this next step.

CONTENT

What would you think of a new software encyclopedia for the U.S. English market if it had the content shown in Figure 10.4?

FIGURE 10.4. THESE CONTENT SAMPLES WOULD SEEM STRANGE IN AN AMERICAN ENGLISH ENCYCLOPEDIA.



In the United States, a soccer ball isn't the same thing as a football! You don't drive on the left! These may not seem right to you, but in other countries they would be perfectly accurate. If you're testing a product that will be localized, you need to carefully examine the content to make sure it's appropriate to the area where it will be used.
Content is all the other "stuff" besides the code that goes into the product (see Chapter 2, "The Software Development Process"). The following list shows various types of content that you should carefully review for localization issues. Don't consider it a complete list; there can be many more examples depending on the product. Think about what other items in your software might be problematic if it was sent to another country.

Sample documents	Icons
Pictures	Sounds
Video	Help files
Maps with disputed boundaries	Marketing material
Packaging	Web links

A NOSE TOO LONG
In 1993, Microsoft released two products for kids called Creative Writer and Fine Artist. These products used a helper character named McZee to guide the kids through the software. A great deal of research went into the design of McZee to select his look, color, mannerisms, personality, and so on. He turned out to be a rather strange looking fellow with buck teeth, dark purple skin, and a big nose.
Unfortunately, after a great deal of work was done drawing the animations that would appear on the screen, a call came in from one of Microsoft's foreign offices. They had received a preliminary version of the software and after reviewing it said that it was unacceptable. The reason: McZee's nose was too long. In their culture, people with large noses weren't common and, right or wrong, they associated having a large nose with lots of negative stereotypes. They said that the product wouldn't sell if it was localized for their locale.
It would have been way too costly to create two different McZees, one for each market, so the artwork completely to that point was thrown out, and McZee had his first nose job.

The bottom line is that the content that goes with the software, whether it's text, graphics, sounds, or whatever, is especially prone to having localization issues. Test the content with an eye for these types of problems and, if you're not experienced with the culture of the locale that the software is destined for, be sure to call in someone who is.

DATA FORMATS
Different locales use different formats for data units such as currency, time, and measurement. Just as with content, these are localization, not translation, issues. An American English publishing program that works with inches couldn't simply undergo a text translation to use centimeters. It would require code changes to alter the underlying formulas, gridlines, and so on.

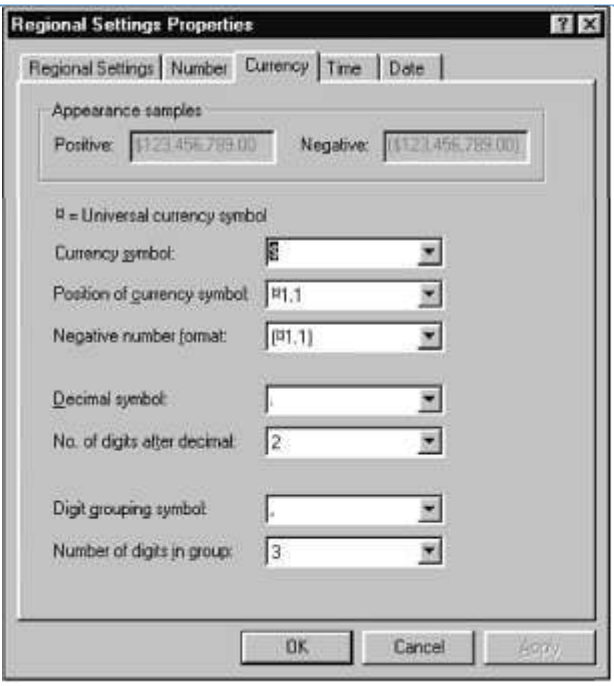
Table 10.1 shows many of the different categories of units that you'll need to become familiar with if you're testing localized software.

TABLE 10.1. DATA FORMAT CONSIDERATIONS FOR LOCALIZED SOFTWARE

Unit	Considerations
Measurements	Metric or English: meters vs. yards
Numbers	Comma, decimal, or space separators; how negatives are shown; # symbol for number; 1.200,00 vs. 1200.00 or 100 vs. (100)
Currency	Different symbols and where they're placed: 30? vs. ?30
Dates	Order of month, day, year; separators; leading zeros; long and short formats: dd/mm/yy vs. mm/dd/yy or May 5, 2005 vs. 15 de mayo 2005
Times	12-hour or 24-hour, separators 3:30pm vs. 15:30
Calendars	Different calendars and starting days: In some countries Sunday is not the first day of the week
Addresses	Order of lines; postal code used: 98072 vs. T2N 0E6
Telephone numbers	Parenthesis or dash separators: (425) 555-1212 vs. 425-555-1212 vs. 425.555.1212
Paper sizes	Different paper and envelope sizes: US Letter vs. A4

Fortunately, most operating systems designed for use in multiple locales support these different units and their formats. Figure 10.5 shows an example from Windows. Having this built-in support makes it easier, but by no means foolproof, for programmers to write localized software.

FIGURE 10.5. THE WINDOWS REGIONAL SETTINGS OPTIONS ALLOW A USER TO SELECT HOW NUMBERS, CURRENCY, TIMES, AND DATES WILL BE DISPLAYED.



Appearance samples

Positive: $123,456,789.00 Negative: ($123,456,789.00)

¤ = Universal currency symbol

Currency symbol: $

Position of currency symbol: ¤1.1

Negative number format: (¤1.1)

Decimal symbol: .

No. of digits after decimal: 2

Digit grouping symbol: ,

Number of digits in group: 3

NOTE
How a unit is displayed isn't necessarily how it's treated internally by the software. For example, the Date tab on the Regional Settings program shows a short date style of m/d/yy. That doesn't imply that the operating system handles only a 2-digit year (and hence is a Y2K bug). In this case, the setting means only a 2-digit year is displayed. The operating system still supports a 4-digit year for computations - more things to consider when testing.

If you're testing localized software, you'll need to become very familiar with the units of measure used by the target locale. To properly test the software, you'll need to create different equivalence partitions of test data from the ones you create for testing the original version of the software.

CONFIGURATION AND COMPATIBILITY ISSUES

The information covered in Chapters 8, "Configuration Testing," and 9, "Compatibility Testing," on configuration and compatibility testing is very important when testing localized versions of software. The problems that can crop up when software interacts with different hardware and software are amplified by all the new and different combinations. Performing this testing isn't necessarily more difficult, just a bit larger of a task. It can also tax your logistical skills to locate and acquire the foreign version of hardware and software to test with.

FOREIGN PLATFORM CONFIGURATIONS

Windows XP supports 106 different languages and 66 different keyboard layouts. It does this, as shown in Figure 10.6, through the Keyboard Properties dialog via Control Panel. The drop-down list for languages runs from Afrikaans to Ukrainian and includes eight different versions of English other than American English (Australian, British, Canadian, Caribbean, Irish, Jamaican, New Zealand, and South African), five different German dialects, and 20 different Spanish dialects.

FIGURE 10.6. WINDOWS SUPPORTS THE USE OF DIFFERENT KEYBOARDS AND LANGUAGES THROUGH THE KEYBOARD PROPERTIES DIALOG.

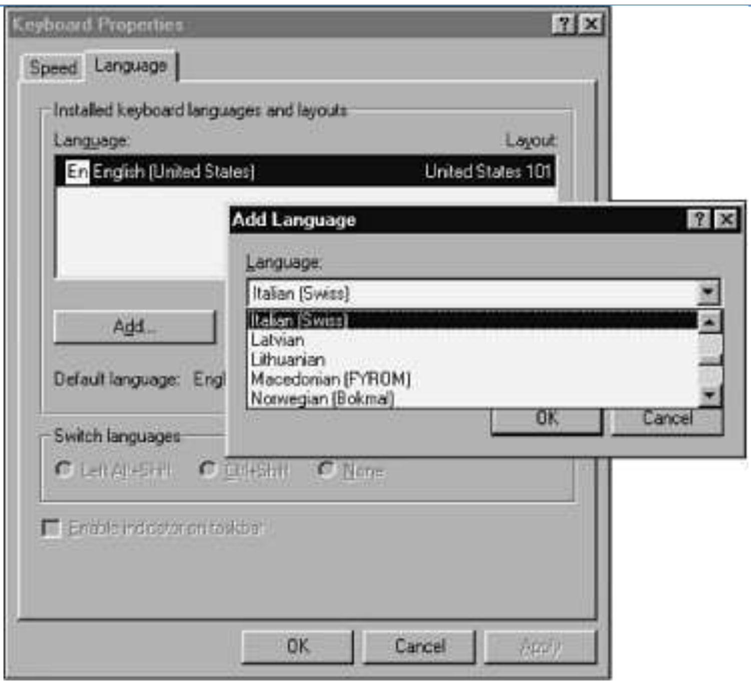
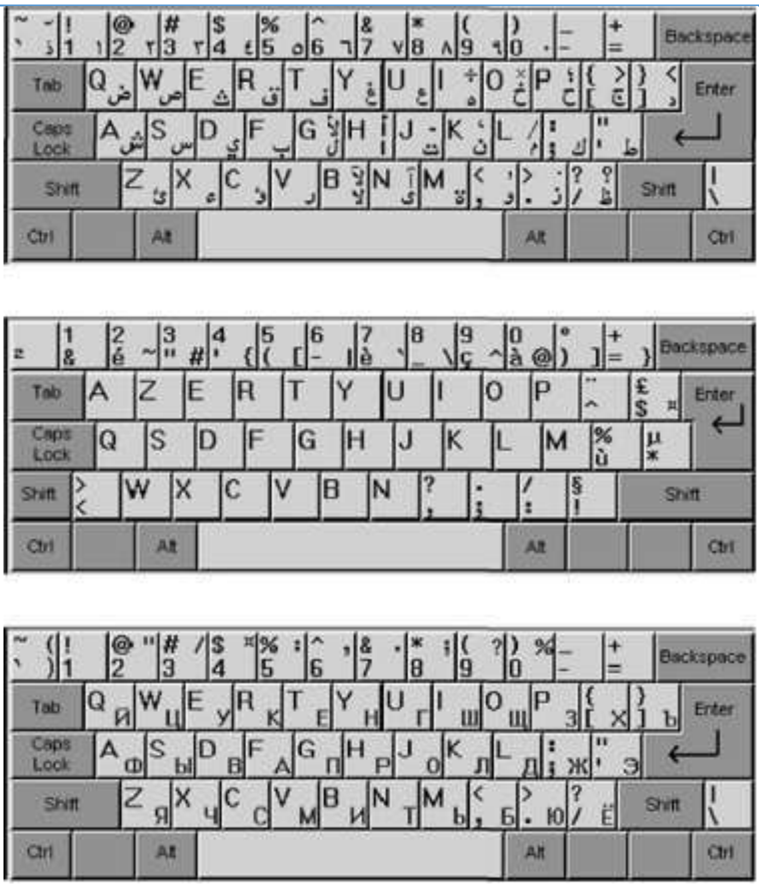


Figure 10.7 shows examples of three different keyboard layouts designed for different countries. You'll notice that each has keys specific to its own language, but also has English characters. This is fairly common, since English is often spoken as a second language in many countries, and allows the keyboard to be used with both native and English language software.

FIGURE 10.7. THE ARABIC, FRENCH, AND RUSSIAN KEYBOARDS SUPPORT CHARACTERS SPECIFIC TO THOSE LANGUAGES. (WWW.FINGERTIPSOFT.COM)



Keyboards are probably the piece of hardware with the largest language dependencies, but depending on what you're testing, there can be many others. Printers, for example, would need to print all the characters your software sends to them and properly format the output on the various paper sizes used in different countries. If your software uses a modem, there might be issues related to the phone lines or communication protocol differences. Basically, any peripheral that your software could potentially work with needs to be considered for a place in your equivalence partitions for platform configuration and compatibility testing.

NOTE

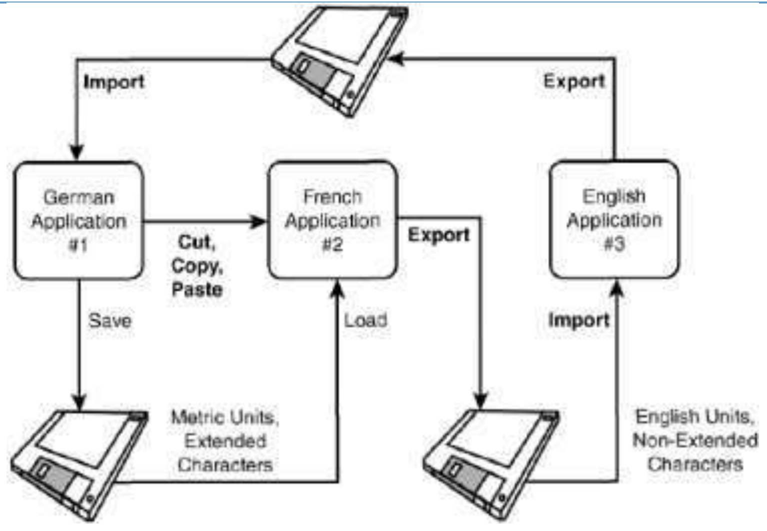
When designing your equivalence partitions, don't forget that you should consider all the hardware and software that can make up the platform. This includes the hardware, device drivers for the hardware, and the operating system. Running a French printer on a Mac, with a British operating system, and a German version of your software might be a perfectly legitimate configuration for your users.

DATA COMPATIBILITY

Just as with platform configuration testing, compatibility testing of data takes on a whole new meaning when you add localization to the equation. Figure 10.8 shows how complex it can get moving data from one application to another. In this example, a German application that uses metric units and extended characters can move data to a different French program by saving and loading to disk or using cut and paste. That French application can then export the data for import to yet another English application.

That English program, which uses English units and non-extended characters, can then move it all back to original German program.

FIGURE 10.8. DATA COMPATIBILITY TESTING OF LOCALIZED SOFTWARE CAN GET FAIRLY COMPLEX.



During this round and round of data transfers, with all the conversions and handling of measurement units and extended characters, there are numerous places for bugs. Some of these bugs might be due to design decisions. For example, what should happen to data moved from one application to another if it needs to change formats? Should it be automatically converted, or should the user be prompted for a decision? Should it show an error or should the data just move and the units change?
These important questions need to be answered before you can start testing the compatibility of your localized software. As soon as you have those specifications, your compatibility testing should proceed as it normally would - just with more test cases in your equivalence partitions.

HOW MUCH SHOULD YOU TEST?

The big uncertainty that looms over localization testing is in determining how much of the software you should test. If you spent six months testing the American English version, should you spend six months testing a version localized into French? Should you spend even more because of additional configuration and compatibility issues?
This complex issue comes down to two questions:

- Was the project intended to be localized from the very beginning?
- Was programming code changed to make the localized version?

If the software was designed from the very beginning to account for all the things discussed in this chapter, the risk is much smaller that a localized version will be very buggy and require lots of testing. If, on the other hand, the software was written specifically for the U.S. English market and then it was decided to localize it into another language, it would probably be wise to treat the software as a completely new release requiring full testing.
NOTE
The amount of localization testing required is a risk-based decision, just as all testing is. As you gain experience in testing, you'll learn what variables go into the decision-making process.

The other question deals with what needs to change in the overall software product. If the localization effort involves changing only content such as graphics and text - not code - the test effort can sometimes be just a validation of the changes. If, however, because of poor design or other problems, the underlying code must change, the testing needs take that into account and check functionality as well as content.

IS IT LOCALIZABLE?

One method used by teams who know they are going to localize their product is to test for localizability. That is, they test the first version of the product, assuming that it will eventually be localized. The white-box testers examine the code for text strings, proper handling of units of measure, extended characters, and other code-level issues. They may even create their own "fake" localized version. The black-box testers carefully review the spec and the product itself for localizing problems such as text in graphics and configuration issues. They can use the "fake" version to test for compatibility.

Eventually, when the product is localized, many of the problems that would have shown up later have already been found and fixed, making the localization effort much less painful and costly.

SUMMARY

Ha Ön egy rátermett és képzett softver ismer?, és folyékonyan beszél egy nyelvet az Angolon kívül, Ön egy nagyon piacképes szakképzett személy.
That's the same first sentence of this chapter - only written in Hungarian this time. Don't worry if you can't read it. You've learned in this chapter that knowing the language is only part of the overall testing required for a localized product. Much work can be done by checking the product for localizability and for testing language-independent areas.
If you are fluent in a language other than English, keep reading this book, and learn all you can about software testing. With the global economy and the worldwide adoption of technology and computers you will, as the Hungarian phrase roughly says, "have a very marketable skill set."
For more information on localization programming and testing for Windows, visit www.microsoft.com/globaldev. For the Mac, consult the Apple website, developer.apple.com/intl/localization/tools.html. Linux programmers and testers can find localization information at www.linux.com/howtos/HOWTO-INDEX/other-lang.shtml.

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. What's the difference between translation and localization?
2. Do you need to know the language to be able to test a localized product?
3. What is text expansion and what common bugs can occur because of it?
4. Identify several areas where extended characters can cause problems.
5. Why is it important to keep text strings out of the code?
6. Name a few types of data formats that could vary from one localized program to another.

CHAPTER 11. USABILITY TESTING

IN THIS CHAPTER

- User Interface Testing
- What Makes a Good UI?
- Testing for the Disabled: Accessibility Testing

Software is written to be used. That sounds pretty obvious, but it's sometimes forgotten in the rush to design, develop, and test a complex product. So much time and effort is spent on the technology aspects of writing the code that the development team ignores the most important aspect of software - that someone will eventually use the stuff. It really doesn't matter whether the software is embedded in a microwave oven, a telephone switching station, or an Internet stock trading website. Eventually the bits and bytes bubble up to where a live person will interact with it. Usability is how appropriate, functional, and effective that interaction is.

You may have heard the term ergonomics, the science of designing everyday things so that they're easy and functional to use. An ergonomist's main concern is in achieving usability.

Now, you're not going to get the knowledge of a four-year ergonomics degree in the 15 or so pages of this chapter, nor do you need to. Remember from Chapter 1, "Software Testing Background," the fifth rule of what constitutes a bug: 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. That's your blank check for usability testing.

You're likely the first person, other than the programmers, to use the software. You've become familiar with the specification and investigated who the customers will be. If you have problems using the software while you're testing it, odds are the customers will, too.

Because there are so many different types of software, it's impossible to go into detail about usability issues for all of them. Usability of a nuclear reactor shutdown sequence is pretty different from usability of a voicemail menu system. What you'll learn in this chapter are the basics of what to look for - with a bias toward software that you use on your PC every day. You can then take those ideas and apply them to whatever software you have to test.

Highlights of this chapter include

- What usability testing involves
- What to look for when testing a user interface
- What special usability features are needed by the disabled

USER INTERFACE TESTING

The means that you use to interact with a software program is called its user interface or UI. All software has some sort of UI. Purists might argue that this isn't true, that software such as what's in your car to control the fuel/air ratio in the engine doesn't have a user interface. In truth, it doesn't have a conventional UI, but the extra pressure you need to apply to the gas pedal and the audible sputtering you hear from the tailpipe is indeed a user interface.

The computer UI we're all familiar with has changed over time. The original computers had toggle switches and light bulbs. Paper tape, punch cards, and teletypes were popular user interfaces in the '60s and '70s. Video monitors and simple line editors such as MS-DOS came next. Now we're using personal computers with sophisticated graphical user interfaces (GUIs). Soon we'll be speaking and listening to our PCs, carrying on verbal conversations as we do with people!

Although these UIs were very different, technically they all provided the same interaction with the computer - the means to give it input and receive output.

WHAT MAKES A GOOD UI?

Many software companies spend large amounts of time and money researching the best way to design the user interfaces for their software. They use special usability labs run by ergonomic specialists. The labs are equipped with one-way mirrors and video cameras to record exactly how people use their software. Everything the users (subjects) do from what keys they press, how they use the mouse, what mistakes they make, and what confuses them is analyzed to make improvements to the UI.

You may be wondering what a software tester could possibly contribute with such a detailed and scientific process. By the time the software is specified and written, it should have the perfect UI. But, if that's the case, why are there so many VCRs blinking 12:00?

First, not every software development team designs their interface so scientifically. Many UIs are just thrown together by the programmers - who may be good at writing code, but aren't necessarily ergonomics experts. Other reasons might be that technological limitations or time constraints caused the UI to be sacrificed. As you learned in Chapter 10, "Foreign-Language Testing," the reason might be that the software wasn't properly localized. In the end, the software tester needs to assume the responsibility of testing the product's usability, and that includes its user interface.
You might not feel that you're properly trained to test a UI, but you are. Remember, you don't have to design it. You just have to pretend you're the user and find problems with it.
Here's a list of seven important traits common to a good UI. It doesn't matter if the UI is on a digital watch or is the Mac OS X interface, they all still apply.

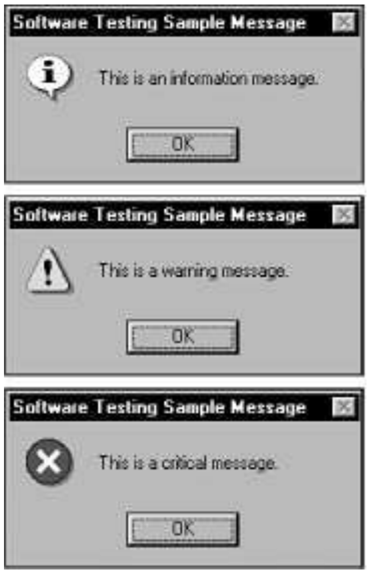
- Follows standards and guidelines
- Intuitive
- Consistent
- Flexible
- Comfortable
- Correct
- Useful

If you read a UI design book, you may also see other traits being listed as important. Most of them are inherent or follow from these seven. For example, "easy to learn" isn't listed above, but if something is intuitive and consistent, it's probably easy to learn. As a tester, if you concentrate on making sure your software's UI meets these criteria, you'll have a darn good interface. Each trait is discussed in detail in the following sections.

FOLLOWS STANDARDS AND GUIDELINES

The single most important user interface trait is that your software follows existing standards and guidelines - or has a really good reason not to. If your software is running on an existing platform such as Mac or Windows, the standards are set. Apple's are defined in the book Macintosh Human Interface Guidelines, published by Addison-Wesley, also available online at developer.apple.com/documentation/mac/ HIGuidelines/HIGuidelines-2.html. Microsoft's are in the book Microsoft Windows User Experience, published by Microsoft Press, with the online version at msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwue/html/welcome.asp.
Each book goes into meticulous detail about how software that runs on each platform should look and feel to the user. Everything is defined from when to use check boxes instead of an option button (when both states of the choice are clearly opposite and unambiguous) to when it's proper to use the information, warning, and critical messages as shown in Figure 11.1.

FIGURE 11.1. DID YOU EVER NOTICE THAT THERE ARE THREE DIFFERENT LEVELS OF MESSAGES IN WINDOWS? WHEN AND HOW TO USE EACH ONE IS DEFINED IN THE USER INTERFACE STANDARDS FOR WINDOWS.



Software Testing Sample Message

This is an information message.

OK

Software Testing Sample Message

This is a warning message.

OK

Software Testing Sample Message

This is a critical message.

OK

NOTE
If you're testing software that runs on a specific platform, you need to treat the standards and guidelines for that platform as an addendum to your product's specification. Create test cases based on it just as you would from the product's spec.

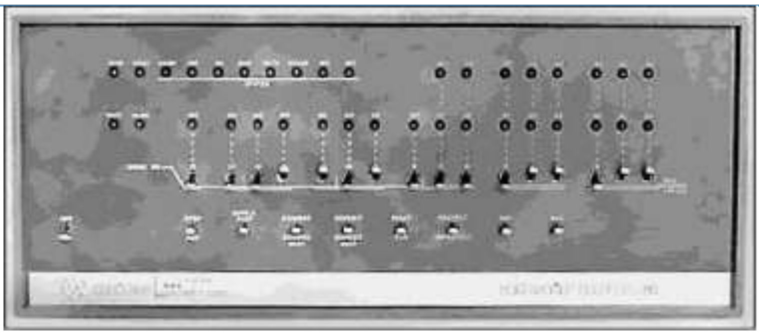
These standards and guidelines were developed (hopefully) by experts in software usability. They have accounted for a great deal of formal testing, experience, and trial and error to devise rules that work well for their users. If your software strictly follows the rules, most of the other traits of a good UI will happen automatically. Not all of them will because your team may want to improvise on them a bit, or the rules may not perfectly fit with your software. In those cases, you need to really pay attention to usability issues.

It's also possible that your platform doesn't have a standard, or maybe your software is the platform. In those situations, your design team will be the ones creating the usability standards for your software. You won't be able to take for granted the rules that someone else has already figured out, and the remaining traits of a good user interface will be even more important for you to follow.

INTUITIVE

In 1975 the MITS (Micro Instrumentation Telemetry Systems) Altair 8800 was released as one of the first personal computers. Its user interface (see Figure 11.2) was nothing but switches and lights - not exactly intuitive to use.

FIGURE 11.2. THE MITS ALTAIR 8800 AND ITS LESS-THAN-INTUITIVE USER INTERFACE. (PHOTO COURTESY OF THE COMPUTER MUSEUM OF AMERICA, WWW.COMPUTER-MUSEUM.ORG.)



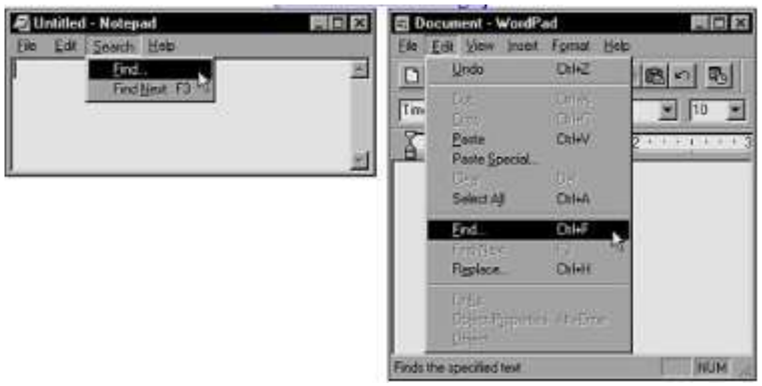
The Altair was designed for computer hobbyists, people who are a lot more forgiving of user interface issues. Today, users want much more out of their software than what the Altair 8800 provided. Everyone from grandmothers to little kids to Ph.D.s are using computers in their daily lives. The computers with the most intuitive UIs are the ones that people don't even realize they're using.
When you're testing a user interface, consider the following things and how they might apply to gauging how intuitive your software is:

- Is the user interface clean, unobtrusive, not busy? The UI shouldn't get in the way of what you want to do. The functions you need or the response you're looking for should be obvious and be there when you expect them.
- Is the UI organized and laid out well? Does it allow you to easily get from one function to another? Is what to do next obvious? At any point can you decide to do nothing or even back up or back out? Are your inputs acknowledged? Do the menus or windows go too deep?
- Is there excessive functionality? Does the software attempt to do too much, either as a whole or in part? Do too many features complicate your work? Do you feel like you're getting information overload?
- If all else fails, does the help system really help you?

CONSISTENT

Consistency within your software and with other software is a key attribute. Users develop habits and expect that if they do something a certain way in one program, another will do the same operation the same way. Figure 11.3 shows an example of how two Windows applications, which should be following a standard, aren't consistent. In Notepad, Find is accessed through the Search menu or by pressing F3. In WordPad, a very similar program, it's accessed through the Edit menu or by pressing Ctrl+F.

FIGURE 11.3. WINDOWS NOTEPAD AND WORDPAD ARE INCONSISTENT IN HOW THE FIND FEATURE IS ACCESSED.



Inconsistencies such as this frustrate users as they move from one program to another. It's even worse if the inconsistency is within the same program. If there's a standard for your software or your platform, follow it. If not, pay particular attention to your software's features to make sure that similar operations are performed similarly. Think about a few basic items as you review your product:

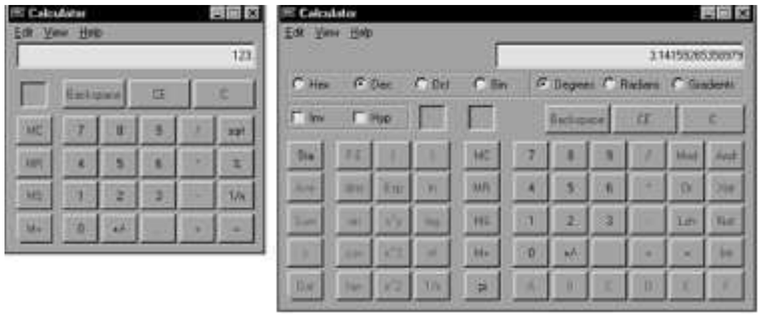
- Shortcut keys and menu selections. In a voicemail system, pressing 0, not other numbers, is almost always the "get-out" button that connects you to a real person. In Windows, pressing F1 should always get you help.
- Terminology and naming. Are the same terms used throughout the software? Are features named consistently? For example, is Find always called Find, or is it sometimes called Search?

- Audience. Does the software consistently talk to the same audience level? A fun greeting card program with a colorful user interface shouldn't display error messages of arcane technobabble.
- Placement for buttons such as OK and Cancel. Did you ever notice that in Windows, OK is always on the top or left and Cancel on the right or bottom? The Mac OS places OK on the right. Keyboard equivalents to onscreen buttons should also be consistent. For example, the Esc key always does a cancel and Enter does an OK.

FLEXIBLE

Users like choices - not too many, but enough to allow them to select what they want to do and how they want to do it. The Windows Calculator (see Figure 11.4) has two views: Standard and Scientific. Users can decide which one they need for their task or the one they're most comfortable using.

FIGURE 11.4. THE WINDOWS CALCULATOR SHOWS ITS FLEXIBILITY BY HAVING TWO DIFFERENT VIEWS.



Of course, with flexibility comes complexity. In the Calculator example you'll have a much larger test effort than if there's just one view. The test impact of flexibility is felt most in the areas covered in Chapter 5, "Testing the Software with Blinders On," with states and with data:

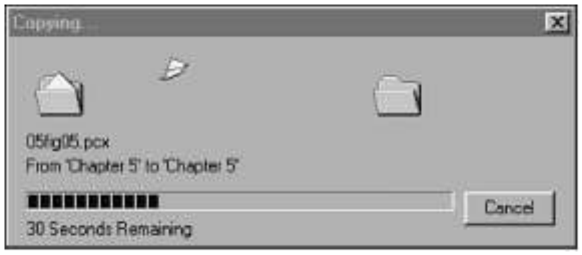
- State jumping. Flexible software provides more options and more ways to accomplish the same task. The result is additional paths among the different states of the software. Your state transition diagrams can become much more complex and you'll need to spend more time deciding which interconnecting paths should be tested.
- State termination and skipping. This is most evident when software has power-user modes where a user who's very familiar with the software can skip numerous prompts or windows and go directly to where they want to go. A voicemail system that allows you to directly punch in your party's extension is an example. If you're testing software that allows this, you'll need to make sure that all the state variables are correctly set if all the intermediate states are skipped or terminated early.
- Data input and output. Users want different ways to enter their data and see their results. To put text into a WordPad document, you can type it, paste it, load it from six different file formats, insert it as an object, or drag it with the mouse from another program. The Microsoft Excel spreadsheet program allows you to view your data in 14 different standard and 20 different custom graphs. Who even knew there were that many possibilities? Testing all the different ways to get data in and out of your software can very quickly increase the effort necessary and make for tough choices when creating your equivalence partitions.

COMFORTABLE

Software should be comfortable to use. It shouldn't get in the way or make it difficult for a user to do his work. Software comfort is a pretty touchy-feely concept. Researchers have spent their careers trying to find the right formula to make software comfortable. It can be a difficult concept to quantify, but you can look for a few things that will give you a better idea of how to identify good and bad software comfort:

- Appropriateness. Software should look and feel proper for what it's doing and who it's for. A financial business application should probably not go crazy with loud colors and sound effects. A space game, on the other hand, will have much more leeway with the rules. Software should neither be too garish nor too plain for the task it's intended to perform.
- Error handling. A program should warn users before a critical operation and allow users to restore data lost because of a mistake. People take the Undo/Redo feature for granted today, but it wasn't long ago that these features didn't exist.
- Performance. Being fast isn't always a good thing. More than one program has flashed error messages too quickly to read. If an operation is slow, it should at least give the user feedback on how much longer it will take and show that it's still working and hasn't frozen. Status bars, as shown in Figure 11.5, are a popular way to accomplish this.

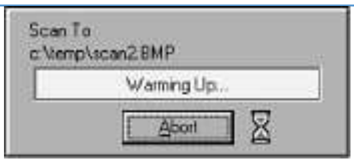
FIGURE 11.5. STATUS BARS SHOW HOW MUCH OF THE WORK HAS BEEN COMPLETED AND HOW MUCH IS LEFT TO GO.



CORRECT

The comfort trait is admittedly a bit fuzzy and often can be left to interpretation. Correctness, though, isn't. When you're testing for correctness, you're testing whether the UI does what it's supposed to do. Figure 11.6 is an example of a UI that isn't correct.

FIGURE 11.6. THIS SOFTWARE HAS A COMPLETELY USELESS ABORT BUTTON.



This figure shows a message box from a popular page-scanning program for Windows. The box appears when a scan is started and is supposed to provide a way for the user to stop the scan mid-process. Unfortunately, it doesn't work. Note that the cursor is an hourglass. An hourglass means (according to the Windows standard) that the software is busy and can't accept any input. Then why is the Abort button there? You can repeatedly click the Abort button during the entire scan, which can take a minute or more, and nothing happens. The scan continues uninterrupted until it completes. If clicking the Abort button with the hourglass cursor did stop the scan, would that be a bug? You bet it would!
Correctness problems such as this are usually obvious and will be found in your normal course of testing against the product specification. You should pay attention to some areas in particular, however:

- Marketing differences. Are there extra or missing functions, or functions that perform operations different from what the marketing material says? Notice that you're not comparing the software to the specification - you're comparing it to the sales information. They're usually different.
- Language and spelling. Some programmers are poor spellers and writers and often create very interesting user messages. The following is an order confirmation message from a popular e-commerce website - hopefully fixed by the time you read this:

  If there are any discrepancies with the information below, please contact us immediately to ensure timely delivery of the products that you ordered.

- Bad media. Media is any supporting icons, images, sounds, or videos that go with your software's UI. Icons should be the same size and have the same palette. Sounds should all be of the same format and sampling rate. The correct ones should be displayed when chosen from the UI.
- WYSIWYG (what you see is what you get). Make sure that what the UI displays is really what you have. When you click the Save button, is the document onscreen exactly what's saved to disk? When you load it back, does it perfectly compare with the original? When you print it, does the output perfectly match what's previewed on the screen?

USEFUL

The final trait of a good user interface is whether it's useful. Remember, here you're not concerned with whether the software itself is useful, just whether the particular feature is. A popular term used in the software industry to describe unnecessary or gratuitous features is dancing bologna. Think of a sausage bouncing around on the screen - completely unnecessary!
When you're reviewing the product specification, preparing to test, or actually performing your testing, ask yourself if the features you see actually contribute to the software's value. Do they help users do what the software is intended to do? If you don't think they're necessary, do some research to find out why they're in the software. It's possible that there are reasons you're not aware of, or it could just be dancing bologna. Those superfluous features, whether they be in a solitaire program or a heart monitor are bad for the user and mean extra testing for you.

TESTING FOR THE DISABLED: ACCESSIBILITY TESTING

A serious topic that falls under the area of usability testing is that of accessibility testing or testing for the disabled. A 1997 government Survey of Income and Program Participation (SIPP) used by the U.S. Census Bureau found that about 53 million people (nearly 20% of the population) in the country had some sort of disability. Table 11.1 shows a more detailed breakdown.

TABLE 11.1. PEOPLE WITH DISABILITIES

Age	Percentage of People with Disabilities
024	18%
2544	13%
4554	23%
5564	36%
6569	45%
7074	47%

TABLE 11.1. PEOPLE WITH DISABILITIES

Age	Percentage of People with Disabilities
7579	58%
80+	74%

Cutting the data another way, reveals that 7.7 million people have difficulty seeing the words and letters in a newspaper. 1.8 million people are legally blind and 8 million people have difficulty hearing.
With our aging population and the penetration of technology into nearly every aspect of our lives, the usability of software becomes more important every day.
Although there are many types of disabilities, the following ones make using computers and software especially difficult:

- Visual impairments. Color blindness, extreme near and far sightedness, tunnel vision, dim vision, blurry vision, and cataracts are examples of visual limitations. People with one or more of these would have their own unique difficulty in using software. Think about trying to see where the mouse pointer is located or where text or small graphics appear onscreen. What if you couldn't see the screen at all?
- Hearing impairments. Someone may be partially or completely deaf, have problems hearing certain frequencies, or picking a specific sound out of background noise. Such a person may not be able to hear the sounds or voices that accompany an onscreen video, audible help, or system alerts.
- Motion impairments. Disease or injury can cause a person to lose fine, gross, or total motor control of his hands or arms. It may be difficult or impossible for some people to properly use a keyboard or a mouse. For example, they may not be able to press more than one key at a time or may find it impossible to press a key only once. Accurately moving a mouse may not be possible.
- Cognitive and language. Dyslexia and memory problems may make it difficult for someone to use complex user interfaces. Think of the issues outlined previously in this chapter and how they might impact a person with cognitive and language difficulties.

LEGAL REQUIREMENTS

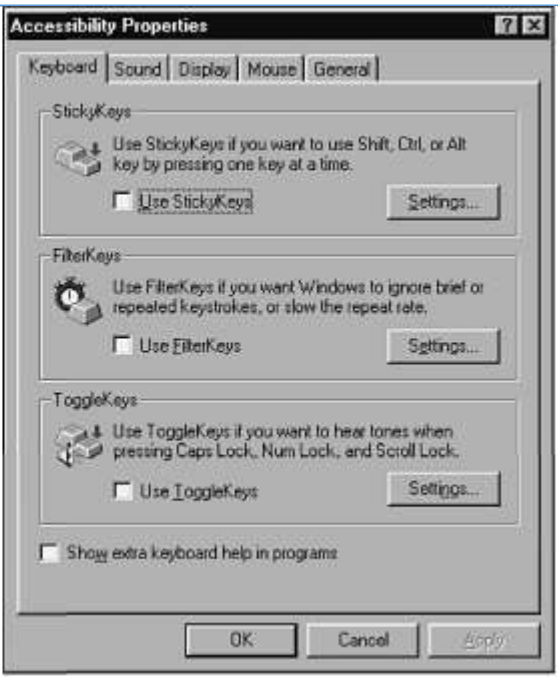
Fortunately, developing software with a user interface that can be used by the disabled isn't just a good idea, a guideline, or a standard - it's often the law. In the United States, three laws apply to this area and other countries are considering and adopting similar laws:

- The Americans with Disability Act states that businesses with 15 or mores employees must make reasonable accommodations for employees, or potential employees, with disabilities. The ADA has recently been applied to commercial Internet websites, mandating that they be made accessible to the public who uses them.
- Section 508 of the Rehabilitation Act is very similar to the ADA and applies to any organization that receives federal funding.
- Section 255 of the Telecommunications Act requires that all hardware and software that transfers information over the Internet, a network, or the phone lines be made so that it can be used by people with disabilities. If it's not directly usable, it must be compatible (see Chapter 8, "Configuration Testing," and Chapter 9, "Compatibility Testing") with existing hardware and software accessibility aids.

ACCESSIBILITY FEATURES IN SOFTWARE

Software can be made accessible in one of two ways. The easiest is to take advantage of support built into its platform or operating system. Windows, Mac OS, Java, and Linux all support accessibility to some degree. Your software only needs to adhere to the platform's standards for communicating with the keyboard, mouse, sound card, and monitor to be accessibility enabled. Figure 11.7 shows an example of the Windows accessibility settings control panel.

FIGURE 11.7. THE WINDOWS ACCESSIBILITY FEATURES ARE SET FROM THIS CONTROL PANEL.



Use StickyKeys if you want to use Shift, Ctrl, or Alt key by pressing one key at a time.

☐ Use StickyKeys

Use FilterKeys if you want Windows to ignore brief or repeated keystrokes, or slow the repeat rate.

☐ Use FilterKeys

Use ToggleKeys if you want to hear tones when pressing Caps Lock, Num Lock, and Scroll Lock.

☐ Use ToggleKeys

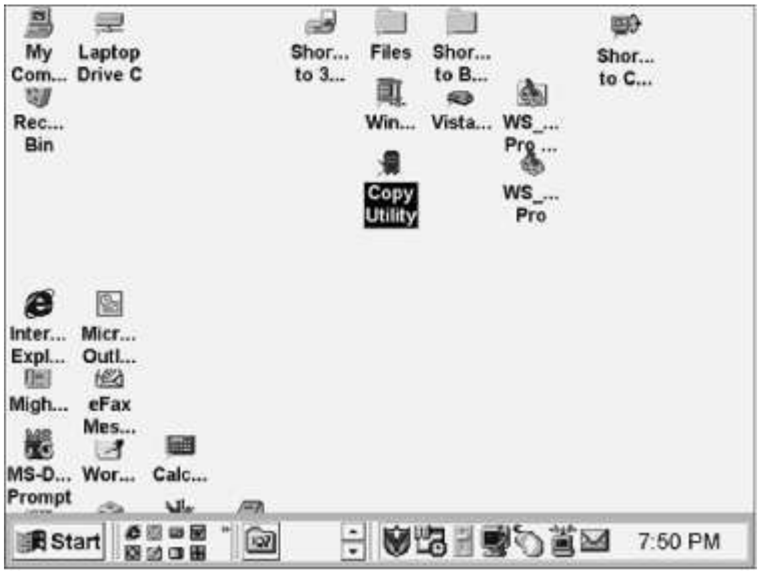
☐ Show extra keyboard help in programs

If the software you're testing doesn't run on these platforms or is its own platform, it will need to have its own accessibility features specified, programmed, and tested.
The latter case is obviously a much larger test effort than the first, but don't take built-in support for granted, either. You'll need to test accessibility features in both situations to make sure that they comply.
NOTE
If you're testing usability for your product, be sure to create test cases specifically for accessibility. You'll feel good knowing that this area is thoroughly tested.

Each platform is slightly different in the features that it offers, but they all strive to make it easier for applications to be accessibility enabled. Windows provides the following capabilities:

- StickyKeys allows the Shift, Ctrl, or Alt keys to stay in effect until the next key is pressed.
- FilterKeys prevents brief, repeated (accidental) keystrokes from being recognized.
- ToggleKeys plays tones when the Caps Lock, Scroll Lock, or NumLock keyboard modes are enabled.
- SoundSentry creates a visual warning whenever the system generates a sound.
- ShowSounds tells programs to display captions for any sounds or speech they make. These captions need to be programmed into your software.
- High Contrast sets up the screen with colors and fonts designed to be read by the visually impaired. Figure 11.8 shows an example of this.

FIGURE 11.8. THE WINDOWS DESKTOP CAN BE SWITCHED TO THIS HIGH CONTRAST MODE FOR EASIER VIEWING BY THE VISUALLY IMPAIRED.



- MouseKeys allows use of keyboard keys instead of the mouse to navigate.
- SerialKeys sets up a communications port to read in keystrokes from an external non-keyboard device. Although the OS should make these devices look like a standard keyboard, it would be a good idea to add them to your configuration testing equivalence partitions.

For more information about the accessibility features built into the popular OS platforms, consult the following websites:

- www.microsoft.com/enable
- www.apple.com/accessibility
- www-3.ibm.com/able
- www.linux.org/docs/ldp/howto/Accessibility-HOWTO

SUMMARY

Remember this from our definition of a bug in Chapter 1? 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.
As a software tester checking the usability of a software product, you're likely the first person to use the product in a meaningful way, the first person to see it all come together in its proposed final form. If it's hard to use or doesn't make sense to you, customers will have the same issues.
Above all, don't let the vagueness or subjectivity of usability testing hinder your test effort. It's vague and subjective by nature. Even the experts who design the user interfaces will admit to that - well, some of them will. If you're testing a new product's UI, refer to the lists in this chapter that define what makes for a good one. If it doesn't meet these criteria, it's a bug, and if it's a usability bug, it might just be the law.

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. True or False: All software has a user interface and therefore must be tested for usability.
2. Is user interface design a science or an art?
3. If there's no definitive right or wrong user interface, how can it be tested?
4. List some examples of poorly designed or inconsistent UIs in products you're familiar with.

5. What four types of disabilities could affect software usability?
6. If you're testing software that will be accessibility enabled, what areas do you need to pay close attention to?

CHAPTER 12. TESTING THE DOCUMENTATION

IN THIS CHAPTER

- Types of Software Documentation
- The Importance of Documentation Testing
- What to Look for When Reviewing Documentation
- The Realities of Documentation Testing

In Chapter 2, "The Software Development Process," you learned that there's a great deal of work and a great number of non-software pieces that make up a software product. Much of that non-software is its documentation.

In simpler days, software documentation was at most a readme file copied onto the software's floppy disk or a short 1-page insert put into the box. Now it's much, much more, sometimes requiring more time and effort to produce than the software itself.

As a software tester, you typically aren't constrained to just testing the software. Your responsibility will likely cover all the parts that make up the entire software product. Assuring that the documentation is correct is your job, too.

In this chapter you'll learn about testing software documentation and how to include it in your overall software test effort. Highlights of this chapter include

- The different types of software documentation
- Why documentation testing is important
- What to look for when testing documentation

TYPES OF SOFTWARE DOCUMENTATION

If your software's documentation consisted of nothing but a simple readme file, testing it wouldn't be a big deal. You'd make sure that it included all the material that it was supposed to, that everything was technically accurate, and (for good measure) you might run a spell check and a virus scan on the disk.

That would be it. But, the days of documentation consisting of just a readme file are gone.

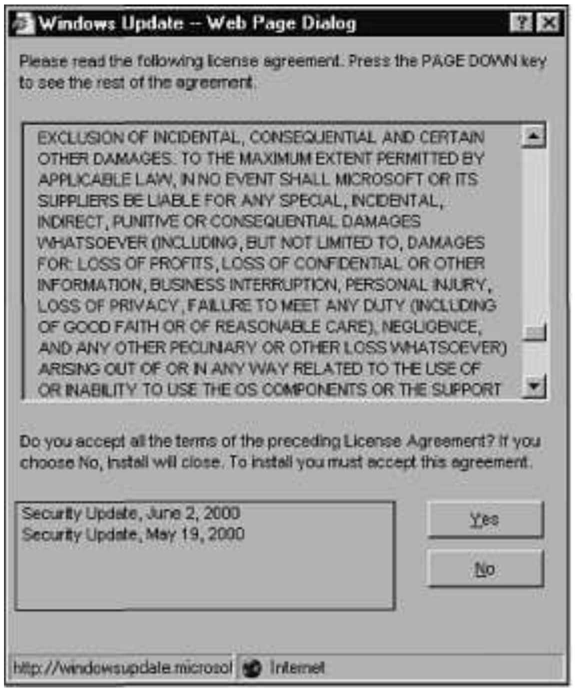
Today, software documentation can make up a huge portion of the overall product. Sometimes, it can seem as if the product is nothing but documentation with a little bit of software thrown in.

Here's a list of software components that can be classified as documentation. Obviously, not all software will have all the components, but it's possible:

- Packaging text and graphics. This includes the box, carton, wrapping, and so on. The documentation might contain screen shots from the software, lists of features, system requirements, and copyright information.
- Marketing material, ads, and other inserts. These are all the pieces of paper you usually throw away, but they are important tools used to promote the sale of related software, add-on content, service contracts, and so on. The information for them must be correct for a customer to take them seriously.

- Warranty/registration. This is the card that the customer fills out and sends in to register the software. It can also be part of the software, being displayed onscreen for the user to read, acknowledge, and complete online.
- EULA. Pronounced "you-la," it stands for End User License Agreement. This is the legal document that the customer agrees to that says, among other things, that he won't copy the software nor sue the manufacturer if he's harmed by a bug. The EULA is sometimes printed on the envelope containing the media - the floppy or CD. It also may pop up onscreen during the software's installation. An example is shown in Figure 12.1.

FIGURE 12.1. THE EULA IS PART OF THE SOFTWARE'S DOCUMENTATION AND EXPLAINS THE LEGAL TERMS OF USE FOR THE SOFTWARE.



Please read the following license agreement. Press the PAGE DOWN key to see the rest of the agreement.

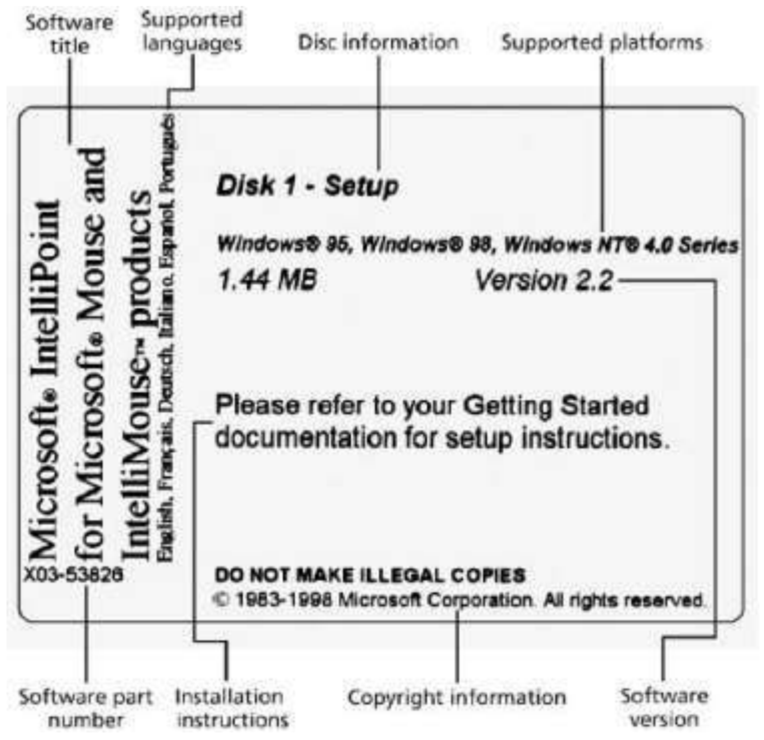
EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, PUNITIVE OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR: LOSS OF PROFITS, LOSS OF CONFIDENTIAL OR OTHER INFORMATION, BUSINESS INTERRUPTION, PERSONAL INJURY, LOSS OF PRIVACY, FAILURE TO MEET ANY DUTY (INCLUDING OF GOOD FAITH OR OF REASONABLE CARE), NEGLIGENCE, AND ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE OS COMPONENTS OR THE SUPPORT

Do you accept all the terms of the preceding License Agreement? If you choose No, Install will close. To install you must accept this agreement.

Security Update, June 2, 2000
Security Update, May 19, 2000

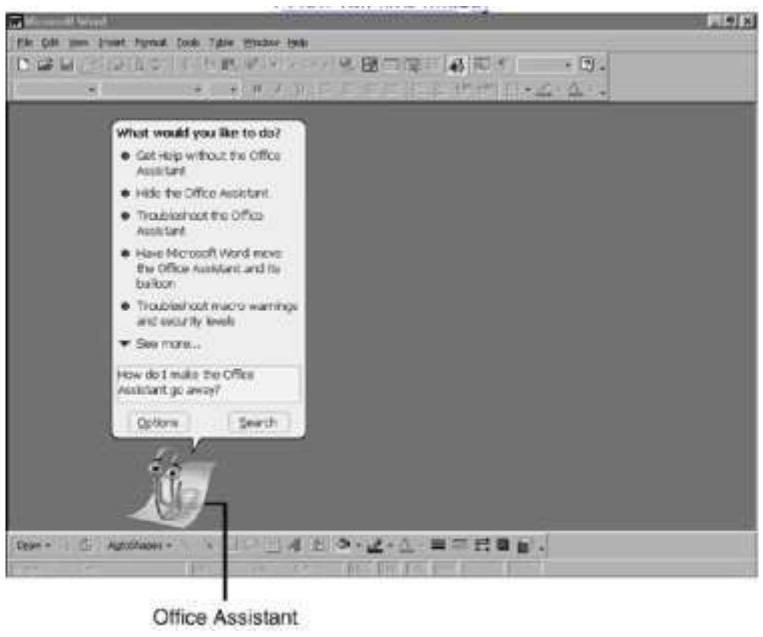
- Labels and stickers. These may appear on the media, on the box, or on the printed material. There may also be serial number stickers and labels that seal the EULA envelope. Figure 12.2 shows an example of a disk label and all the information that needs to be checked.

FIGURE 12.2. THERE'S LOTS OF DOCUMENTATION ON THIS DISK LABEL FOR THE SOFTWARE TESTER TO CHECK.


Software title
Supported languages
Disc information
Supported platforms
Microsoft® IntelliPoint for Microsoft® Mouse and IntelliMouse™ products
English, Français, Deutsch, Italiano, Español, Português
Disk 1 - Setup
Windows® 95, Windows® 98, Windows NT® 4.0 Series
1.44 MB
Version 2.2
Please refer to your Getting Started documentation for setup instructions.
X03-53826
DO NOT MAKE ILLEGAL COPIES
© 1983-1998 Microsoft Corporation. All rights reserved.
Software part number
Installation instructions
Copyright information
Software version

- Installation and setup instructions. Sometimes this information is printed directly on the discs, but it also can be included on the CD sleeve or as a CD jewel box insert. If it's complex software, there could be an entire installation manual.
- User's manual. The usefulness and flexibility of online manuals has made printed manuals much less common than they once were. Most software now comes with a small, concise "getting started"-type manual with the detailed information moved to online format. The online manuals can be distributed on the software's media, on a website, or a combination of both.
- Online help. Online help often gets intertwined with the user's manual, sometimes even replacing it. Online help is indexed and searchable, making it much easier for users to find the information they're looking for. Many online help systems allow natural language queries so users can type Tell me how to copy text from one program to another and receive an appropriate response.
- Tutorials, wizards, and CBT (Computer Based Training). These tools blend programming code and written documentation. They're often a mixture of both content and high-level, macro-like programming and are often tied in with the online help system. A user can ask a question and the software then guides him through the steps to complete the task. Microsoft's Office Assistant, sometimes referred to as the "paper clip guy" (see Figure 12.3), is an example of such a system.

FIGURE 12.3. THE MICROSOFT OFFICE ASSISTANT IS AN EXAMPLE OF A VERY ELABORATE HELP AND TUTORIAL SYSTEM.


What would you like to do?
Get Help without the Office Assistant
Hide the Office Assistant
Troubleshoot the Office Assistant
Have Microsoft Word move the Office Assistant and its balloon
Troubleshoot macro warnings and security levels
See more...
How do I make the Office Assistant go away?
Options
Search
Office Assistant

- Samples, examples, and templates. An example of these would be a word processor with forms or samples that a user can simply fill in to quickly create professional-looking results. A compiler could have snippets of code that demonstrate how to use certain aspects of the language.
- Error messages. These have already been discussed a couple times in this book as an often neglected area, but they ultimately fall under the category of documentation.

THE IMPORTANCE OF DOCUMENTATION TESTING

Software users consider all these individual non-software components parts of the overall software product. They don't care whether the pieces were created by a programmer, a writer, or a graphic artist. What they care about is the quality of the entire package.

NOTE

If the installation instructions are wrong or if an incorrect error message leads them astray, users will view those as bugs with the software - ones that should have been found by a software tester.

Good software documentation contributes to the product's overall quality in three ways:

- It improves usability. Remember from Chapter 11, "Usability Testing," all the issues related to a product's usability? Much of that usability is related to the software documentation.
- It improves reliability. Reliability is how stable and consistent the software is. Does it do what the user expects and when he expects it? If the user reads the documentation, uses the software, and gets unexpected results, that's poor reliability. As you'll see in the rest of this chapter, testing the software and the documentation against each other is a good way to find bugs in both of them.
- It lowers support costs. In Chapter 2 you learned that problems found by a customer can cost 10 to 100 times as much as if they were found and fixed early in the product's development. The reason is that users who are confused or run into unexpected problems will call the company for help, which is expensive. Good documentation can prevent these calls by adequately explaining and leading users through difficult areas.

NOTE

As a software tester, you should treat the software's documentation with the same level of attention and give it the same level of effort that you do the code. They are one in the same to the user. If you're not asked to test the documentation, be sure to raise this as an issue and work to have it included in your overall testing plan.

WHAT TO LOOK FOR WHEN REVIEWING DOCUMENTATION

Testing the documentation can occur on two different levels. If it's non-code, such as a printed user's manual or the packaging, testing is a static process much like what's described in Chapters 4, "Examining the Specification," and 6, "Examining the Code." Think of it as technical editing or technical proofreading. If the documentation and code are more closely tied, such as with a hyperlinked online manual or with a helpful paper clip guy, it becomes a dynamic test effort that should be checked with the techniques you learned in Chapters 5, "Testing the Software with Blinders On," and 7, "Testing the Software with X-Ray Glasses." In this situation, you really are testing software.

NOTE

Whether or not the documentation is code, a very effective approach to testing it is to treat it just like a user would. Read it carefully, follow every step, examine every figure, and try every example. If there is sample code, type it in and make sure it works as described. With this simple real-world approach, you'll find bugs both in the software and the documentation.

Table 12.1 is a simple checklist to use as a basis for building your documentation test cases.

TABLE 12.1. A DOCUMENTATION TESTING CHECKLIST

What to Check	What to Consider
General Areas	
Audience	Does the documentation speak to the correct level of audience, not too novice, not too advanced?
Terminology	Is the terminology proper for the audience? Are the terms used consistently? If acronyms or abbreviations are used, are they standard ones or do they need to be defined? Make sure that your company's acronyms don't accidentally make it through. Are all the terms indexed and cross-referenced correctly?
Content and subject matter	Are the appropriate topics covered? Are any topics missing? How about topics that shouldn't be included, such as a feature that was cut from the product and no one told the manual writer. Is the material covered in the proper depth?
Correctness	
Just the facts	Is all the information factually and technically correct? Look for mistakes caused by the writers working from outdated specs or sales people inflating the truth. Check the table of contents, the index, and chapter references. Try the website URLs. Is the product support phone number correct? Try it.
Step by step	Read all the text carefully and slowly. Follow the instructions exactly. Assume nothing! Resist the temptation to fill in missing steps; your customers won't know what's missing. Compare your results to the ones shown in the documentation.
Correctness	

TABLE 12.1. A DOCUMENTATION TESTING CHECKLIST

What to Check	What to Consider
Figures and screen captures	Check figures for accuracy and precision. Do they represent the correct image and is the image correct? Make sure that any screen captures aren't from prerelease software that has since changed. Are the figure captions correct?
Samples and examples	Load and use every sample just as a customer would. If it's code, type or copy it in and run it. There's nothing more embarrassing than samples that don't work - and it happens all the time!
Spelling and grammar	In an ideal world, these types of bugs wouldn't make it through to you. Spelling and grammar checkers are too commonplace not to be used. It's possible, though, that someone forgot to perform the check or that a specialized or technical term slipped through. It's also possible that the checking had to be done manually, such as in a screen capture or a drawn figure. Don't take it for granted.

Finally, if the documentation is software driven, test it as you would the rest of the software. Check that the index list is complete, that searching finds the correct results, and that the hyperlinks and hotspots jump to the correct pages. Use equivalence partition techniques to decide what test cases to try.

THE REALITIES OF DOCUMENTATION TESTING

To close this chapter, it's important for you to learn a few things that make documentation development and testing a bit different from software development. Chapter 3 was titled "The Realities of Software Testing." You might call these issues the realities of documentation testing:

- Documentation often gets the least attention, budget, and resources. There seems to be the mentality that it's a software project first and foremost and all the other stuff is less important. In reality, it's a software product that people are buying and all that other stuff is at least as important as the bits and bytes. If you're responsible for testing an area of the software, make sure that you budget time to test the documentation that goes along with that code. Give it the same attention that you do the software and if it has bugs, report them.
- It's possible that the people writing the documentation aren't experts in what the software does. Just as you don't have to be an accounting expert to test a spreadsheet program, the writer doesn't have to be an expert in the software's features to write its documentation. As a result, you can't rely on the person creating the content to make sense out of poorly written specs or complex or unclear product features. Work closely with writers to make sure they have the information they need and that they're up-to-date with the product's design. Most importantly, tell them about difficult-to-use or difficult-to-understand areas of the code that you discover so they can better explain those areas in the documentation.
- Printed documentation takes time to produce, sometimes weeks or even months. Because of this time difference, a software product's documentation may need to be finalized - locked down - before the software is completed. If the software functionality changes or bugs are discovered during this critical period, the documentation can't be changed to reflect them. That's why the readme file was invented. It's how those last-minute changes are often communicated to users.

  The solution to this problem is to have a good development model, follow it, hold your documentation release to the last possible minute, and release as much documentation as possible, in electronic format, with the software.

SUMMARY

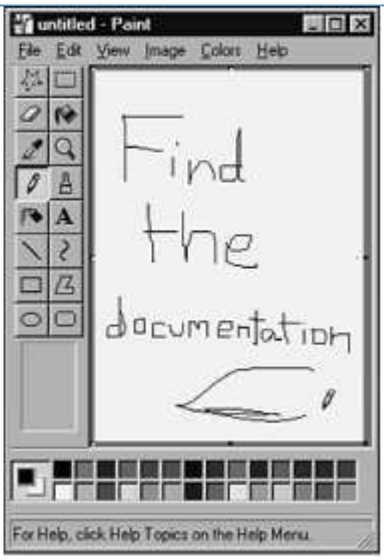
Hopefully this chapter opened your eyes to how much more there can be to a software product than the code the programmers write. The software's documentation, in all its forms, created by writers, illustrators, indexers, and so on, can easily take more effort to develop and test than the actual software. From the user's standpoint, it's all the same product. An online help index that's missing an important term, an incorrect step in the installation instructions, or a blatant misspelling are bugs just like any other software failure. If you properly test the documentation, you'll find the bugs before your users do.
In the next chapter you'll learn about applying your testing skills to an area that's in the news almost daily - software security. It's an area that will require particular attention in every task you perform as a software tester from early code and specification reviews to testing the documentation.

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. Start up Windows Paint (see Figure 12.4) and look for several examples of documentation that should be tested. What did you find?

FIGURE 12.4. WHAT EXAMPLES OF DOCUMENTATION CAN YOU FIND IN WINDOWS PAINT?


untitled - Paint
File Edit View Image Colors Help
Find the documentation
For Help, click Help Topics on the Help Menu.

2. The Windows Paint Help Index contains more than 200 terms from airbrush tool to zooming in or out. Would you test that each of these takes you to the correct help topics? What if there were 10,000 indexed terms?
3. True or False: Testing error messages falls under documentation testing.
4. In what three ways does good documentation contribute to the product's overall quality?

CHAPTER 13. TESTING FOR SOFTWARE SECURITY

IN THIS CHAPTER

- WarGames - the Movie
- Understanding the Motivation
- Threat Modeling
- Is Software Security a Feature? Is Security Vulnerability a Bug?
- Understanding the Buffer Overrun

- Using Safe String Functions
- Computer Forensics

It seems as though a day doesn't go by without a news story about yet another computer security issue. Hackers, viruses, worms, spyware, backdoors, Trojan horses, and denial-of-service attacks have become common terms. Even average computer users have been impacted beyond the nuisance level, losing important data and valuable time restoring their systems after an attack. A January 14, 2005, story in the LA Times by Joseph Menn, titled, "No More Internet for Them," reveals that many people have had enough. They're frustrated, angry, and they are "plugging out" - disconnecting their computers from the Internet in an effort to regain control of their PCs. That's not a good sign for the health and growth of the computer industry.
For these reasons, software security is on every programmer's mind (or at least it should be if she wants to stay employed) and is touching all aspects of the software development process. Software testing has yet another area to be concerned with, and this chapter will give you an introduction to this important and timely topic.
Highlights of this chapter include

- Why someone would want to break into a computer
- What types of break-ins are common
- How to work with your design team to identify security issues
- Why software security problems are nothing more than software bugs
- What you, as a software tester, can do to find security vulnerabilities
- How the new field of computer forensics is related to software security testing

WARGAMESTHE MOVIE

One of the first mainstream examples of computer hacking that brought it into the public eye was the 1983 movie WarGames. In this movie, Mathew Broderick's teenage character, David Lightman, uses his IMSAI 8080 home computer and a 300 baud acoustic modem to hack into the US government's NORAD computer system. Once in, he stumbles across a war game scenario called "Global Thermonuclear War" that he thinks is just a game but turns out to be a real-life simulation that almost starts World War III.
How did he gain access? Pretty simply. He programmed his computer to sequentially dial phone numbers from, say 555-0000 to 555-9999, and listen for another computer modem to answer. If a person answered, the computer just hung up. His program ran unattended while he was at school and created a list of "hot" numbers. After school, with that narrowed down list, David called the numbers individually to see which ones would allow him to log-in.
In the case of the NORAD computer, he did some sleuthing to uncover the name of one of the government programmers and found the name of his deceased son. Typing the son's name, JOSHUA, into the login password field gained him full access to the military's "high security" computer system.
More than 20 years later, the technology has changed, but the processes and techniques remain the same. Software and computer systems are now much more interconnected and, unfortunately, there are many more hackers. You might say that WarGames has taken on a new meaning from what it had in the movie - the hackers are playing games and the software industry is fighting a war against them. Software security, or more correctly, the lack of it, has become a huge issue.

WAR DRIVING

With the popularity of wireless "WiFi" networks in metropolitan areas, resourceful hackers have discovered that they can search for open computer networks much like Mathew Broderick's character in WarGames. They simply drive around city streets, sit outside corporate buildings, and walk through coffee shops and restaurants with laptops or inexpensive "sniffers" looking for unprotected wireless networks. The technique, named after the movie, is known as "war driving."

UNDERSTANDING THE MOTIVATION

In the WarGames example, the hacker was motivated by curiosity and the desire to use a secure computer with more capability than his own. Although he felt that his actions weren't malicious because he didn't plan to harm the system, his actions resulted in a cascade effect with serious consequences. As a software tester it's important to understand why someone may want to break into your software. Understanding their intent will aid you in thinking about where the security vulnerabilities might be in the software you're testing.

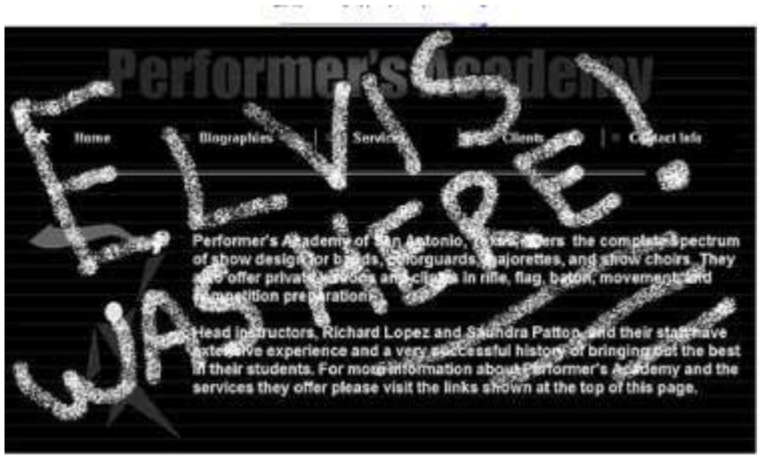
> A secure product is a product that protects the confidentiality, integrity, and availability of the customers' information, and the integrity and availability of processing resources, under control of the system's owner or administrator.
>
> www.microsoft.com/technet/community/chats/trans/security/sec0612.mspx
>
> A security vulnerability is a flaw in a product that makes it infeasible - even when using the product properly - to prevent an attacker from usurping privileges on the user's system, regulating its operation, compromising data on it, or assuming un-granted trust.
>
> www.microsoft.com/technet/archive/community/columns/security/essays/vulnrbl.mspx
>
> Hacker:
>
> One who is proficient at using or programming a computer; a computer buff.
>
> One who uses programming skills to gain illegal access to a computer network or file.
>
> www.dictionary.com

The five motives that a hacker might have to gain access to a system are

- Challenge/Prestige. The simplest and most benign form of hacking is when someone breaks into a system purely for the challenge of the task and the prestige (among his fellow hackers) of succeeding. There's no intent of anything more sinister. War driving is such an activity. Although this may not sound like much of a problem, imagine if a lock picker practiced his craft by unlocking random doors throughout your neighborhood each night and then bragging to his friends which homes had locks that were the easiest to pick. No one would feel secure - and rightly so.
- Curiosity. The next level up the scale is curiosity. Here, the hacker doesn't stop at just gaining access. Once inside, he wants to look around to see what's there. Curiosity is the motive. The hacker will peruse the system looking for something interesting. A software system could have a security vulnerability that allows a hacker to gain access (challenge/prestige) but still be secure enough to prevent the hacker from looking at any of its valuable data.
- Use/Leverage. This is the level where the hacker does more than just breaking and entering. Here the hacker will actually attempt to use the system for his own purpose. The WarGames example we've discussed is a Use/Leverage attack. A present-day example is when a home PC is attacked with an email virus, which then uses the email addresses stored on that PC and its computing power to re-send thousands more instances of the virus. The hacker is able to accomplish much more using the distributed power of many computers than if using just his own. Additionally, the hacker may be able to better cover his tracks by using these "hacked" computers to do the damage.
- Vandalize. When you think of vandalizing, remember the three D's: Defacing, Destruction, and Denial of Service. Defacing is changing the appearance of a website to promote the opinion or

thoughts of the hacker - see Figure 13.1. Destruction takes the form of deleting or altering of data stored on the system. An example might be a college student changing his grades or deleting the final exam. Denial of service is preventing or hindering the hacked system from performing its intended operation. An example of this would be flooding an ecommerce website with so much traffic that it's incapable of handling its standard transactions, locking out paying customers from making purchases. Even worse, the hacker could crash the system resulting in data loss and days of downtime.

FIGURE 13.1. THE DEFACING OF A WEBSITE IS JUST ONE TYPE OF DAMAGE THAT A HACKER CAN INFLICT.



Performer's Academy of San Antonio, Texas, offers the complete spectrum of show design for bands, colorguards, majorettes, and show choirs. They also offer private lessons and clinics in rifle, flag, baton, movement and competition preparation.

Head instructors, Richard Lopez and Saundra Patton, and their staff have extensive experience and a very successful history of bringing out the best in their students. For more information about Performer's Academy and the services they offer please visit the links shown at the top of this page.

- Steal. Probably the most severe form of hacking is stealing, outright theft. The intent is to find something of value that can be used or sold. Credit card numbers, personal information, goods and services, even login IDs and email addresses, all have value to the hacker. In 2003, a 24-year-old computer hacker gained access to and stole 92 million AOL screen names (login IDs). The list of names was later sold several times to various spammers for amounts up to $100,000! Not bad for a few days' work.

Are you afraid yet? You should be. Every day there are countless hackers out there attempting to break into theoretically secure systems. And, many of them succeed. Now that you know why a hacker may want to break into a system, we'll continue our discussion of software security testing by outlining what you can do to assist your development team in creating a secure software product.

THREAT MODELING

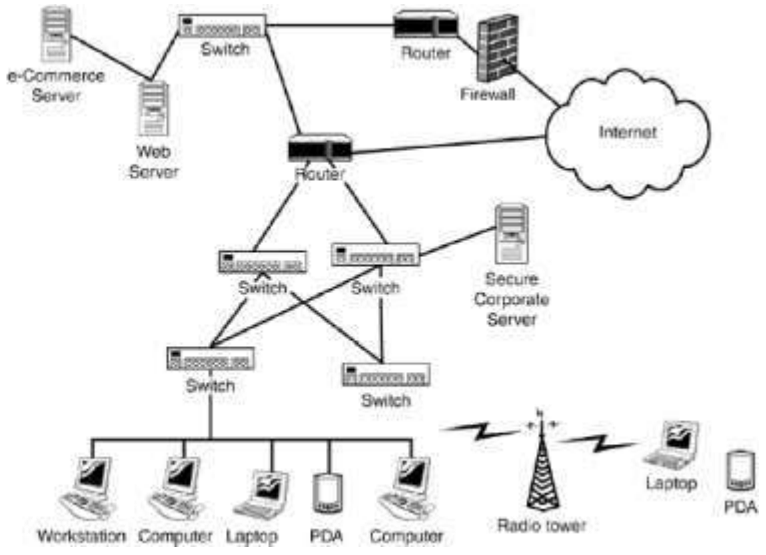
In Michael Howard and David LeBlanc's excellent book, Writing Secure Code (Microsoft Press, 2003, second edition), they describe a process known as threat modeling for evaluating a software system for security issues. Think of threat modeling as a variation of the formal reviews described in Chapter 6, "Examining the Code." The goal in this case, though, is for the review team to look for areas of the product's feature set that are susceptible to security vulnerabilities. With that information, the team can then choose to make changes to the product, spend more effort designing certain features, or concentrate testing on potential trouble spots. Ultimately, such an understanding will result in a more secure product.

NOTE

Unless everyone on the product development team - and that means everyone: project managers, programmers, testers, technical writers, marketing, and product support - understands and agrees to the possible threats, your team will not be able to create a secure product.

Performing the threat modeling is not the software tester's responsibility. This responsibility should fall under the project manager's task list and involve all of the project team members for input. For this reason, we're only going to discuss the process in generalities. For more detailed information on holding a threat modeling review, visit msdn.microsoft.com/library and search for "threat modeling."
Figure 13.2 shows a physical diagram of a fictional complex ecommerce system. Note that the system has multiple entry methods for user access. Users can access the system via PDAs, wireless connections, and standard PCs. The system is connected to the Internet and has server software from different vendors. The system stores payment information in a database that's directly accessible via an internal connection. The preliminary design of such a system requires comprehensive threat modeling to assure that it remains secure when put into operation.

FIGURE 13.2. A COMPLEX SYSTEM REQUIRES COMPREHENSIVE THREAT MODELING TO IDENTIFY SECURITY VULNERABILITIES.



It really doesn't matter whether the software or system your team is developing is more or less complex than what's shown in Figure 13.2. The steps of the threat modeling process are the same. The following steps were adapted from msdn.microsoft.com/library.

- Assemble the threat modeling team. We've already touched on this. Besides the standard team members, a vital addition would be someone with an extensive software security background. For a small team, this could be an outside consultant brought in just for the design stage. In a large company, this person could come from a pool of experts who "float" from one project to another offering their knowledge and expertise. It's important for the team to understand that their initial goal is not to solve the security issues; it's to identify them. Later meetings can be held with smaller specific groups (for example, just a few programmers and testers) to isolate the threats and devise solutions.

- Identify the Assets. Think of all the things that your system has that might be of value to an intruder. Is there personal information of your clients? Maybe credit card numbers? How about computing resources? Could someone want to covertly use your system to send spam to your clients? Would you lose business if your company's website was defaced or replaced?

- Create an Architecture Overview. In this stage you'll want to identify the technology you plan to use for your software and how it's interconnected. Your team will create an architectural diagram that shows the major technological pieces and how they communicate. An important aspect of this is identifying the trust boundaries between the different technologies and the authentication and authorizations that must occur to gain access to the data.

- Decompose the Application. This is a formalized process to identify how and where the data flows through the system. Ideally, it would be based on the design's data flow diagrams and state transition maps. If these diagrams don't exist, they will need to be created. Think about the data as containers and what secures the containers. What are the means for entering the container to see the data? Is the data encrypted? Is it password protected?

- Identify the Threats. Once you thoroughly understand what the components (assets, architecture, and data) are, your team can move on to identifying the threats. Each of the components should be considered threat targets and you should assume that they will be attacked. Think about whether each component could be improperly viewed. Could it be modified? Could a hacker prevent authorized users from using the component? Could someone gain access and take control of the system?

- Document the Threats. Each threat must be documented and tracked to ensure that it is addressed. Documentation is a simple matter of describing the threat (in one sentence or less), the target, what form the attack might take, and what countermeasures the system will use to prevent an attack. In Chapter 19, "Reporting What You Find," you'll learn how to properly track the bugs you discover. Using a similar system may help your team manage the security threats they identify.

- Rank the threats. Finally, it's important to understand that all threats are not created equal. Your software's data may be protected with a Department of Defense 128-bit encryption that would take decades on a super computer to break. Would the data be threatened even with that level of security? Sure it would. Would that threat rank higher than an auto-answer phone modem connected to the administrator's computer? Probably not. Your team will need to look at each threat and determine its ranking. A simple way to do this is to use the DREAD Formula defined at msdn.microsoft.com/library under the chapter "Improving Web Application Security". DREAD stands for

    - Damage Potential How much damage (physical, financial, integrity, and so on) could be done if this area is hacked?

    - Reproducibility How easy is it for a hacker to exploit the vulnerability consistently? Would every attempt be successful? One in 100 attempts? One in a million?

    - Exploitability How technically difficult is it to gain access to the system or data? Is it a few lines of basic macro code that can be emailed around the Internet or does it require someone with expert programming skills?

    - Affected Users If the hack succeeds, how many users will be affected? Would it be a single user or, in the case of the stolen AOL screen names, 92 million?

    - Discoverability What's the likelihood that a hacker will discover the vulnerability? A secret "backdoor" login may not sound discoverable, until a disgruntled employee is fired and posts the information on the Web.

A simple 1=Low, 2=Medium, and 3=High system applied to each of the five categories and then summed up for a value between 5 and 15 would work to arrive at a numerical ranking for each threat. Your team could plan to design and test the most severe issues first, and then continue on to the other lower-ranked threats as time permits. In Chapter 19 you'll learn more about rating and ranking software bugs and some techniques to use to make sure they get fixed.

IS SOFTWARE SECURITY A FEATURE? IS SECURITY VULNERABILITY A BUG?

By now you've hopefully come to the conclusion that software security can be viewed as simply another feature of a software product or system. Most people probably don't want their personal financial data made public by a hacker. They would consider their spreadsheet software's ability to keep that information private a necessary feature. That feature is one that would go toward making the software a good quality product (remember our discussion of quality versus reliability in Chapter 3, "The Realities of Software Testing"?) If the software "failed" and allowed a hacker to see personal bank balances and credit card info, most users would consider that a software bug, pure and simple.
REMINDER
It's a good idea, here, to reiterate our definition of a bug from Chapter 1, "Software Testing Background":

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.

And, to revisit our discussion of test-to-pass and test-to-fail from Chapter 5, "Testing the Software with Blinders On."
When you test-to-pass, you really only ensure that the software minimally works. You don't push its capabilities. You don't see what you can do to break it. You treat it with kid gloves, applying the simplest and most straightforward test cases.
After you assure yourself that the software does what it's specified to do in ordinary circumstances, it's time to put on your sneaky, conniving, devious hat and attempt to find bugs by trying things that should force them out - test-to-fail.

As a software tester, you may be responsible for testing the software's overall security or you may just be responsible for testing that your assigned features are secure. No matter, you'll need to consider all five definitions of a bug. You won't necessarily be given a product specification that explicitly defines how software security is to be addressed in your software. Nor will you be able to assume that the threat model is complete and accurate. For these reasons, you'll need to put on your "test-to-fail" hat and attack the software much like a hacker would - assuming that every feature has a security vulnerability and that it's your job to find it and exploit it.
TIP
Testing for security bugs is a test-to-fail activity and one that will often cover areas of the product that haven't been completely understood or specified.

UNDERSTANDING THE BUFFER OVERRUN

It would be impossible in one chapter, or even one book, to adequately cover all the possible means of attacking a software product. After all, a spreadsheet shared over your home's wireless network is very different from a multiplayer video game played over the Web or a distributed Department of Defense computer system. The operating systems and the technologies are unique and therefore will usually have different security vulnerabilities. There is one common problem, however, that is a security issue in any software product - the buffer overrun.
In the Generic Code Review Checklist in Chapter 6, you learned about Data Reference Errors - bugs caused by using a variable, constant, array, string, or record that hasn't been properly declared or initialized for

how it's being used and referenced." A buffer overrun is such an error. It is the result of poor programming, enabled by many languages such as C and C++, that lack safe string handling functions. Consider the sample C code in Listing 13.1.

LISTING 13.1. EXAMPLE OF A SIMPLE BUFFER OVERFLOW

```
void myBufferCopy(char * pSourceStr) {
    char pDestStr[100];
    int nLocalVar1 = 123;
    int nLocalVar2 = 456;
    strcpy(pDestStr, pSourceStr);
...
}
void myValidate()
{
/*
  Assume this function's code validates a user password
  and grants access to millions of private customer records
/*
}
```

Do you see the problem? The size of the input string, pSourceStr, is unknown. The size of the destination string, pDestStr is 100 bytes. What happens if the source string's length is greater than 100? As the code is written, the source string is copied right into the destination string, no matter what the length. If it's more than 100 bytes, it will fill the destination string and then continue overwriting the values stored in the local variables.

Worse, however, is if the source string is long enough, it could also overwrite the return address of the function myBufferCopy and the contents of the executable code in the function myValidate(). In this example, a competent hacker could enter a super long password, stuffed with hand-written assembly code instead of alphanumeric ASCII characters, and override the intended password validation performed in myValidatepossibly gaining access to the system. Suddenly, those code reviews described in Chapter 6 take on a whole new meaning!

NOTE

This is a greatly simplified example of a buffer overrun to demonstrate the potential problem. Exactly what data or program code gets overwritten, or even if it will be overwritten or executed at all, depends on the compiler and the CPU. But, of course, the hackers know that.

Buffer overruns caused by improper handling of strings are by far the most common coding error that can result in a security vulnerability, but any of the error classes described in Chapter 6 are potential problems. As a software tester, your job is to find these types of bugs as early as possible. A code review would find them early in the development cycle, but there's an even better means - and that's to prevent them from happening in the first place.

USING SAFE STRING FUNCTIONS

In 2002, Microsoft began an initiative to identify the common C and C++ functions that were prone to buffer overrun coding errors. These functions are not "bad" by themselves, but to be used securely they require extensive error checking on the part of the programmer. If that error checking is neglected (and it often is), the code will have a security vulnerability. Given the risk of this oversight, it was decided that it would be best to develop and promote a new set of functions to replace those prone to problems with a robust, thoroughly tested, and documented set of new ones.

These new functions, called Safe String Functions, are available from Microsoft for the Windows XP SP1 and later versions of the Windows DDK and Platform SDK. Many other commercial and freeware libraries that implement "safe strings" are available for common operating systems, processors, and compilers.

The following list (adapted from the article, "Using Safe String Functions," on msdn.microsoft.com/library) explains some of the benefits of using the new functions:

- Each function receives the size of the destination buffer as input. The function can thus ensure that it does not write past the end of the buffer.
- The functions null-terminate all output strings, even if the operation truncates the intended result. The code performing an operation on the returned string can safely assume that it will eventually encounter a null - denoting the string's end. The data prior to the null will be valid and the string won't run on, indefinitely.
- All functions return an NTSTATUS value, with only one possible success code. The calling function can easily determine if the function succeeded in performing its operation.
- Each function is available in two versions. One supports single-byte ASCII characters and the other, double-byte Unicode characters. Remember from Chapter 10, "Foreign-Language Testing," that to support all the letters and symbols in multiple foreign languages, characters need to take up more than one byte of space.

Table 13.1 shows a list of the unsafe functions and the safe functions that replace them. When you and your team are performing code reviews or white-box testing, be on the lookout for the unsafe functions and how they are used. Obviously, your team's programmers should be using the safe versions, but, if not, your code reviews will need to be performed with much more rigor to ensure that any possible security vulnerabilities are addressed.

TABLE 13.1. THE OLD "UNSECURE" C STRING FUNCTIONS AND THEIR NEW "SECURE" REPLACEMENTS

Old "Unsafe" Functions	New "Safe" Functions	Purpose
Strcat wcscat	RtlStringCbCat RtlStringCbCatEx RtlStringCchCat RtlStringCchCatEx	Concatenate two strings.

TABLE 13.1. THE OLD "UNSECURE" C STRING FUNCTIONS AND THEIR NEW "SECURE" REPLACEMENTS

Old "Unsafe" Functions	New "Safe" Functions	Purpose
Strncat wcsncat	RtlStringCbCatN RtlStringCbCatNEx RtlStringCchCatN RtlStringCchCatNEx	Concatenate two byte-counted strings, while limiting the size of the appended string.
Strcpy wcscpy	RtlStringCbCopy RtlStringCbCopyEx RtlStringCchCopy RtlStringCchCopyEx	Copy a string into a buffer.
Strncpy wcsncpy	RtlStringCbCopyN RtlStringCbCopyNEx RtlStringCchCopyN RtlStringCchCopyNEx	Copy a byte-counted string into a buffer, while limiting the size of the copied string.
Strlen wcslen	RtlStringCbLength RtlStringCchLength	Determine the length of a supplied string.
sprintf swprintf _snprintf _snwprintf	RtlStringCbPrintf RtlStringCbPrintfEx RtlStringCchPrintf RtlStringCchPrintfEx	Create a formatted text string that is based on a format string and a set of additional function arguments.

TABLE 13.1. THE OLD "UNSECURE" C STRING FUNCTIONS AND THEIR NEW "SECURE" REPLACEMENTS

Old "Unsafe" Functions	New "Safe" Functions	Purpose
vsprintf	RtlStringCbVPrintf	Create a formatted text string that is based on a format string and one additional function argument.
vswprintf	RtlStringCbVPrintfEx	
_vsnprintf	RtlStringCchVPrintf	
_vsnwprintf	RtlStringCchVPrintfEx	

THE JPEG VIRUS

What could be more secure than a picture? After all, it's data, not executable code. That false assumption was broken in September of 2004 when a virus was discovered that was embedded in several pornographic JPEG images posted to an Internet newsgroup. When viewed, a virus was downloaded to the user's PC. No one thought it was possible, but it was. The problem lied in an exploitation of a buffer overflow.

The JPEG file format, besides storing the picture elements, also allows for the storing of embedded comments. Many software packages for editing and organizing pictures use this field for annotating the picture - "Our family at the beach," "House for Sale," and so forth. This comment field starts with a hex value of 0xFFFE followed by a two-byte value. This value specifies the length of the comment, plus 2 bytes (for the field length). Using this encoding method, a comment of up to 65,533 bytes would be valid. If there is no comment, then the field is supposed to contain a value of 2. The problem is that if the value is an illegal entry of 0 or 1, a buffer overflow occurs.

It turns out that the code used to interpret the JPEG data and turn it into a viewable picture normalized the length by subtracting off the 2 bytes before it read out the actual comment. If the length byte was set to 0, subtracting off 2 yielded a length of -2. The code was written to handle positive integers and interpreted the negative 2 as a positive 4GB. The next 4GB of "comment" data was then loaded, improperly overwriting valid data and program. If that "comment" data was carefully crafted, hand coded, assembly, it could be used to gain access to the viewer's PC. Microsoft had to issue a critical update to all the components that loaded and viewed JPEG images.

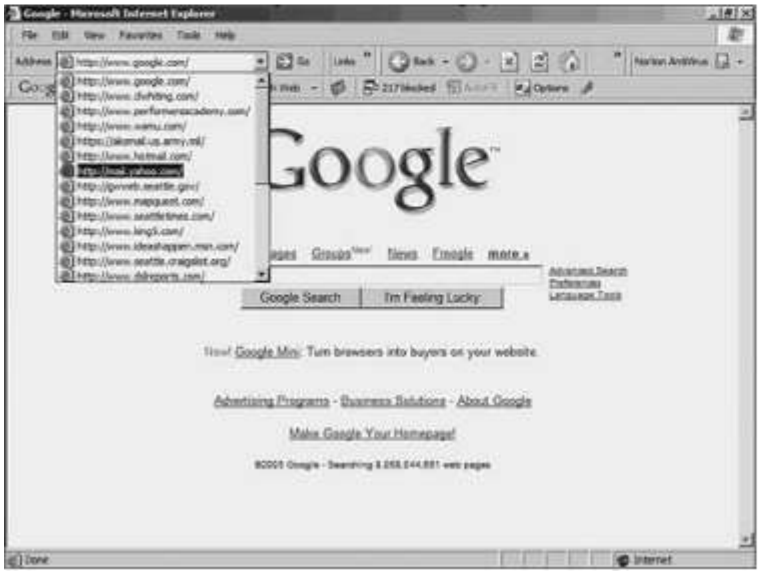
Software vulnerabilities can occur where you never expect them. For this reason, it's imperative to consider software security in all aspects of a software product or system. From a user perspective, security is measure of quality. They may not ask for it, but they know they want it, and will consider lack of software security a bug (a huge one) if they don't have it. We'll close out this chapter with a brief visit to another aspect of computer security, one that's related to privacy, computer forensics.

COMPUTER FORENSICS

Up to this point, we've discussed software security from an active standpoint. We looked at it from the perspective that a hacker may try to actively manipulate your software by finding security vulnerabilities and exploiting them to create access to data or to control the system. Another perspective is that it doesn't have to be that difficult. Sometimes, the data is just lying around for the viewing by those who know where to look.

The first example of this is with features that we're all familiar with in browsing the Web. Figure 13.3 show an example of Internet Explorer's drop-down address bar displaying the history list of websites that have been recently visited. For most users this is not a problem; it's actually useful, allowing you to go back and quickly return to sites you've visited without retyping their full URLs. But, what if this screen shot came from a public access terminal? The person behind you in line could know what you were viewing through a single mouse click.

FIGURE 13.3. THE LIST OF WEBSITES YOU'VE VIEWED COULD BE A SECURITY VULNERABILITY.

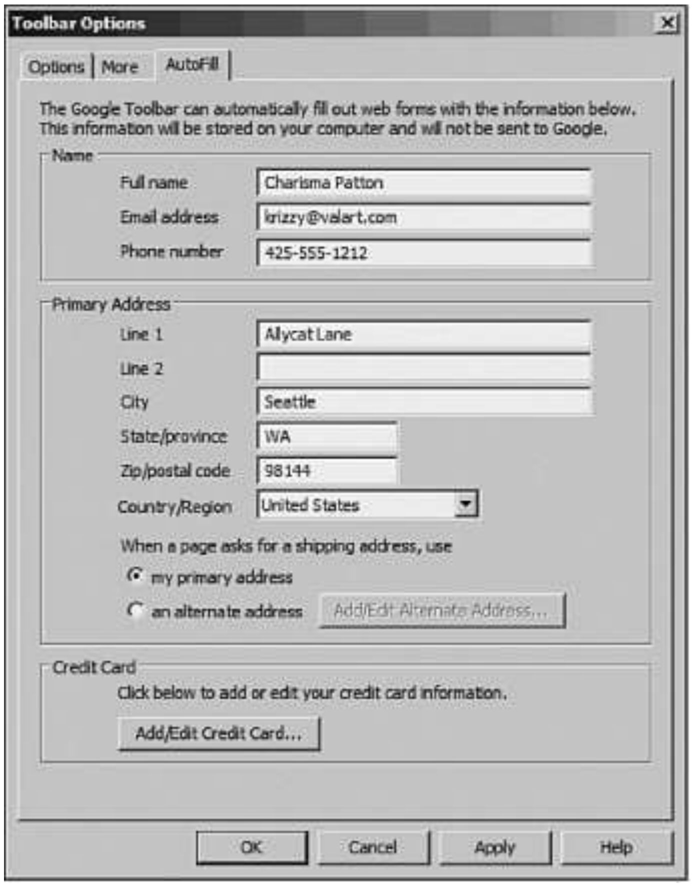


NOTE

Data that "stays around" and isn't deleted from user to user is known as latent data. It should be considered a potential security vulnerability and needs to be discussed in any threat modeling your team does. It may not be considered a problem for your product. Or, it could be a major issue.

Another example of latent data is the Google Toolbar AutoFill feature shown in Figure 13.4. This feature allows you to store information such as your name, address, phone number, email address, and so on so that when a blank form is displayed (such as on an ecommerce order page) you can populate all the fields with a single click. That's a great feature that many of us frequently use. But, if you're testing a product for software security, you need to think like a user and decide if such a feature needs some type of "override" to hide or delete the data so it's not accessible to others.

FIGURE 13.4. THE GOOGLE TOOLBAR AUTOFILL FEATURE STORES INFORMATION FOR QUICKLY FILLING IN WEB FORMS. SECURITY VULNERABILITY?



Toolbar Options

Options | More | AutoFill

The Google Toolbar can automatically fill out web forms with the information below. This information will be stored on your computer and will not be sent to Google.

Name

Full name	Charisma Patton
Email address	krizzy@valart.com
Phone number	425-555-1212

Primary Address

Line 1	Allycat Lane
Line 2	
City	Seattle
State/province	WA
Zip/postal code	98144
Country/Region	United States

When a page asks for a shipping address, use

[✓ my primary address]

an alternate address — Add/Edit Alternate Address...

Credit Card

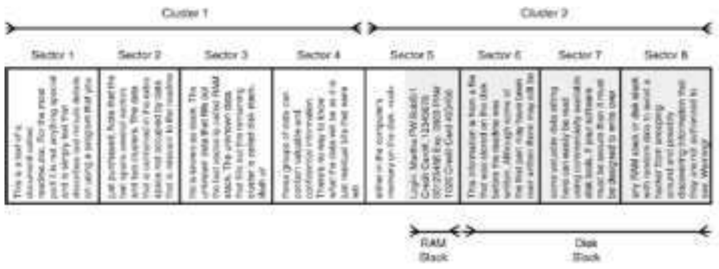
Click below to add or edit your credit card information.

Add/Edit Credit Card...

OK | Cancel | Apply | Help

A more complex example of latent data is one used by computer security experts to discover evidence that could be used in a crime investigation. When data is written to a disk, it is written in blocks. The size of these blocks, called sectors, varies depending on the operating system. MSDOS/Windows uses 512-byte blocks. Depending on the file system being used, the sectors are written out in groups called clusters. The Windows FAT filesystem uses clusters of 2048 bytes, each made up of four 512-byte clusters. Figure 13.5 shows what two clusters might look like on a disk drive after a text file called readme.doc has been written to it. The file is 2200 bytes in length and is shown by the white area spanning Sector 1 through the midpoint of Sector 5. So, if the file is 2200 bytes, what's in the gray area from the Sector 5 through Sector 8? That information is known as latent data.

FIGURE 13.5. THE DATA IN THE FILE README.DOC IS NOT THE ONLY DATA WRITTEN TO DISK.



If the file is 2200 bytes long, it will take up 4.3 512-byte blocks (2200/512=4.3). The data at the end of Sector 5 is called RAM slack because what is contained there is information that happened to reside in the system's random access memory when the file was created. It could be nothing, or it could be

administration passwords or credit card numbers. There's no way to know, but what is a given is that data other than what was in the file was written from the computer's memory onto its disk drive.
The remainder of the gray area, from Sector 6 through the end of Sector 8, is known as disk slack. It exists because the file system writes in 2048-byte clusters and our file only contained enough data to partially fill two of them. The data stored here is typically data that existed on the drive before the file was written. It could be the remnants of another file or a previous, longer, version of readme.doc. This latent data could be benign, or it could contain information that was intentionally deleted or is very private.
NOTE
Although this example uses a disk drive to illustrate the concept of latent data, the security issues of RAM slack and disk slack apply to writable CDs, DVDs, memory cards, and virtually any sort of storage media.

Simple, publicly available, tools can easily view and extract latent data from a disk. Some can even piece together scattered data fragments into their original complete files. When you're testing a software product, you'll need to work with your team to decide if latent data is a security vulnerability issue. If it is, you and your team will need to devise ways to prevent it from occurring.

CHAPTER 14. WEBSITE TESTING

IN THIS CHAPTER

- Web Page Fundamentals
- Black-Box Testing
- Gray-Box Testing
- White-Box Testing
- Configuration and Compatibility Testing
- Usability Testing
- Introducing Automation

The testing techniques that you've learned in previous chapters have been fairly generic. They've been presented by using small programs such as Windows WordPad, Calculator, and Paint to demonstrate testing fundamentals and how to apply them. This final chapter of Part III, "Applying Your Testing Skills," is geared toward testing a specific type of software - Internet web pages. It's a fairly timely topic, something that you're likely familiar with, and a good real-world example to apply the techniques that you've learned so far.
What you'll find in this chapter is that website testing encompasses many areas, including configuration testing, compatibility testing, usability testing, documentation testing, security, and, if the site is intended for a worldwide audience, localization testing. Of course, black-box, white-box, static, and dynamic testing are always a given.
This chapter isn't meant to be a complete how-to guide for testing Internet websites, but it will give you a straightforward practical example of testing something real and give you a good head start if your first job happens to be looking for bugs in someone's website.
Highlights of this chapter include

- What fundamental parts of a web page need testing
- What basic white-box and black-box techniques apply to web page testing
- How configuration and compatibility testing apply
- Why usability testing is the primary concern of web pages
- How to use tools to help test your website

WEB PAGE FUNDAMENTALS

In the simplest terms, Internet web pages are just documents of text, pictures, sounds, video, and hyperlinks much like the CD-ROM multimedia titles that were popular in the mid 1990s. As in those programs, web users can navigate from page to page by clicking hyperlinked text or pictures, searching for words or phrases, and viewing the information they find.

The Internet, though, has introduced two twists to the multimedia document concept that revolutionizes the technology:

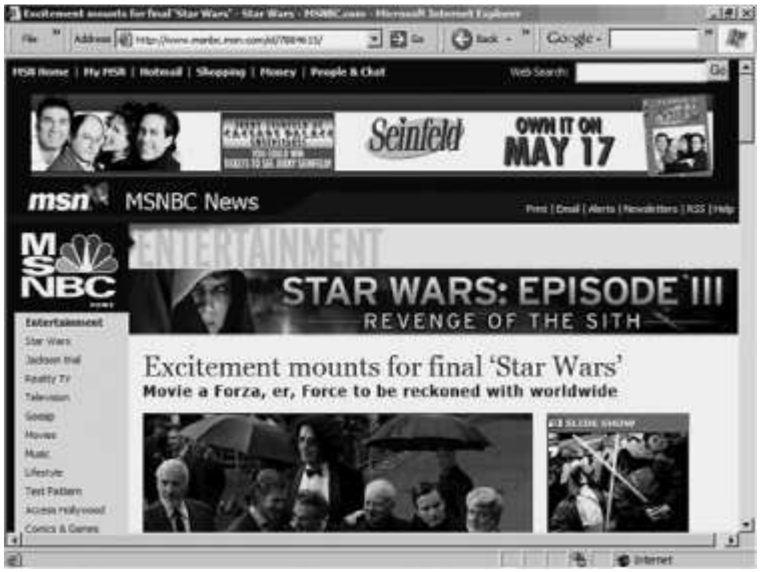
- Unlike data that is stored solely on a CD-ROM, web pages aren't constrained to a single PC. Users can link to and search worldwide across the entire Internet for information on any website.
- Web page authoring isn't limited to programmers using expensive and technical tools. The average person can create a simple web page almost as easily as writing a letter in a word processor.

But, just as giving someone a paint brush doesn't make him an artist, giving someone the ability to create web pages doesn't make him an expert in multimedia publishing. Couple that with the technology explosion that continually adds new website features, and you have the perfect opportunity for a software tester.

Figure 14.1 shows a popular news website that demonstrates many of the possible web page features. A partial list of them includes

- Text of different sizes, fonts, and colors (okay, you can't see the colors in this book)
- Graphics and photos
- Hyperlinked text and graphics
- Rotating advertisements
- Text that has Drop-down selection boxes
- Fields in which the users can enter data

FIGURE 14.1. A TYPICAL WEB PAGE HAS MANY TESTABLE FEATURES.



A great deal of functionality also isn't as obvious, features that make the website much more complex:

- Customizable layout that allows users to change where information is positioned onscreen
- Customizable content that allows users to select what news and information they want to see

- Dynamic drop-down selection boxes
- Dynamically changing text
- Dynamic layout and optional information based on screen resolution
- Compatibility with different web browsers, browser versions, and hardware and software platforms
- Lots of hidden formatting, tagging, and embedded information that enhances the web page's usability

Granted, short of a secure e-commerce website, this is probably one of the more complex and feature-rich web pages on the Internet. If you have the tester mentality (and hopefully you've gained it by reading this far in the book), looking at such a web page should whet your appetite to jump in and start finding bugs. The remainder of this chapter will give you clues on where to look.

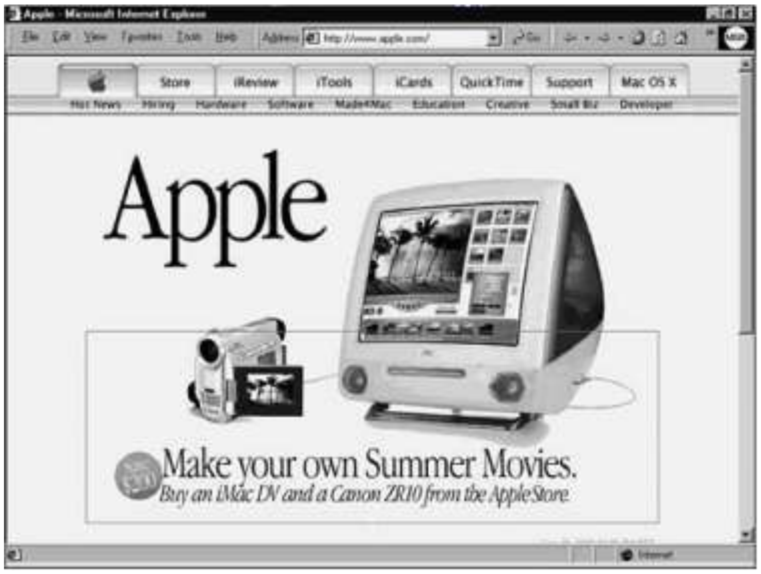
BLACK-BOX TESTING

Remember all the way back to Chapters 4 through 7, the ones that covered the fundamentals of testing? In those vitally important chapters, you learned about black-box, white-box, static, and dynamic testing - the raw skills of a software tester. Web pages are the perfect means to practice what you've learned. You don't have to go out and buy different programs - you can simply jump to a web page, one of your favorites or a completely new one, and begin testing.

The easiest place to start is by treating the web page or the entire website as a black box. You don't know anything about how it works, you don't have a specification, you just have the website in front of you to test. What do you look for?

Figure 14.2 shows a screen image of Apple's website, www.apple.com, a fairly straightforward and typical website. It has all the basic elements - text, graphics, hyperlinks to other pages on the site, and hyperlinks to other websites. A few of the pages have form fields in which users can enter information and a few pages play videos. One interesting thing about this site that's not so common is that it's localized for 27 different locales, from Asia to the UK.

FIGURE 14.2. WHAT WOULD YOU TEST IN A STRAIGHTFORWARD WEBSITE SUCH AS THIS?



If you have access to the Internet, take some time now and explore Apple's website. Think about how you would approach testing it. What would you test? What would your equivalence partitions be? What would you choose not to test?

After exploring a bit, what did you decide? Hopefully you realized that it's a pretty big job. If you looked at the site map (www.apple.com/find/sitemap.html), you found links to more than 100 different sub-sites, each one with several pages.

NOTE

When testing a website, you first should create a state table (see Chapter 5, "Testing the Software with Blinders On"), treating each page as a different state with the hyperlinks as the lines connecting them. A completed state map will give you a better view of the overall task.

Thankfully, most of the pages are fairly simple, made up of just text, graphics, links, and the occasional form. Testing them isn't difficult. The following sections give some ideas of what to look for.

TEXT

Web page text should be treated just like documentation and tested as described in Chapter 12, "Testing the Documentation." Check the audience level, the terminology, the content and subject matter, the accuracy - especially of information that can become outdated - and always, always check spelling.

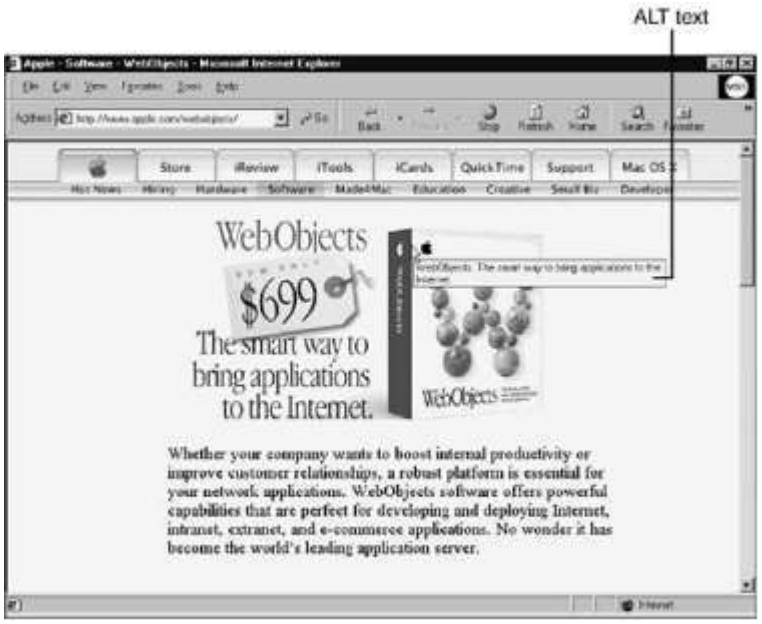
NOTE

Don't rely on spell checkers to be perfect, especially when they're used on web page content. They might only check the regular text but not what's contained in the graphics, scrolling marquees, forms, and so on. You could perform what you think is a complete spell check and still have misspellings on the page.

If there is contact information such as email addresses, phone numbers, or postal addresses, check them to make sure that they're correct. Make sure that the copyright notices are correct and dated appropriately. Test that each page has a correct title. This text appears in the browser's title bar (upper-left corner of Figure 14.2) and what is listed by default when you add the page to your favorites or bookmarks.

An often overlooked type of text is called ALT text, for ALTernate text. Figure 14.3 shows an example of ALT text. When a user puts the mouse cursor over a graphic on the page he gets a pop-up description of what the graphic represents. Web browsers that don't display graphics use ALT text. Also, with ALT text blind users can use graphically rich websites - an audible reader interprets the ALT text and reads it out through the computer's speakers.

FIGURE 14.3. ALT TEXT PROVIDES TEXTUAL DESCRIPTIONS OF GRAPHICS IMAGES ON WEB PAGES.



Whether your company wants to boost internal productivity or improve customer relationships, a robust platform is essential for your network applications. WebObjects software offers powerful capabilities that are perfect for developing and deploying Internet, intranet, extranet, and e-commerce applications. No wonder it has become the world's leading application server.

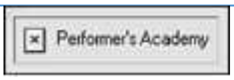
NOTE
Not all browsers support display of the ALT text. Some only show simple TITLE text in the tooltip - or nothing at all. Since this could prevent the blind from using the Web, it should be considered a serious accessibility bug.

Check for text layout issues by resizing your browser window to be very small or very large. This will reveal bugs where the designer or programmer assumed a fixed page width or height. It will also reveal hard-coded formatting such as line breaks that might look good with certain layouts but not with others.

HYPERLINKS
Links can be tied to text or graphics. Each link should be checked to make sure that it jumps to the correct destination and opens in the correct window. If you don't have a specification for the website, you'll need to test whether the jump worked correctly.
Make sure that hyperlinks are obvious. Text links are usually underlined, and the mouse pointer should change (usually to a hand pointer) when it's over any kind of hyperlink - text or graphic.
If the link opens up an email message, fill out the message, send it, and make sure you get a response.
Look for orphan pages, which are part of the website but can't be accessed through a hyperlink because someone forgot to hook them up. You'll likely need to get a list from the web page's designer of the expected pages and compare that with your own state table.

GRAPHICS
Many possible bugs with graphics are covered later under usability testing, but you can check a few obvious things with a simple black-box approach. For example, do all graphics load and display properly? If a graphic is missing or is incorrectly named, it won't load and the web page will display an error where the graphic was to be placed (see Figure 14.4).

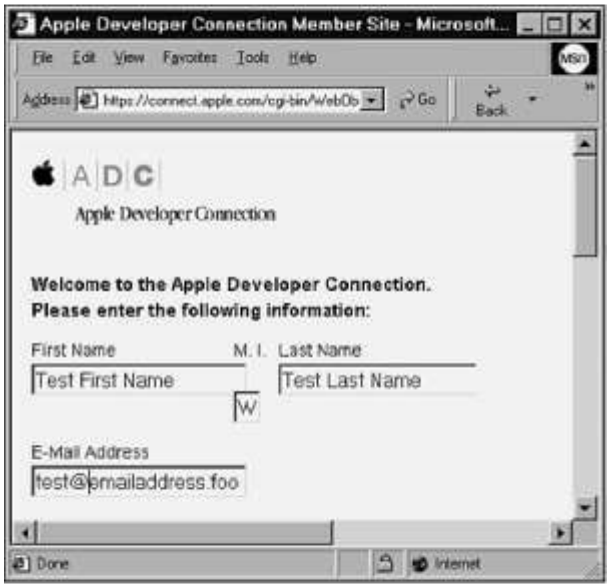
FIGURE 14.4. IF A GRAPHIC CAN'T LOAD ONTO A WEB PAGE AN ERROR IS PUT IN ITS LOCATION.



If text and graphics are intermixed on the page, make sure that the text wraps properly around the graphics. Try resizing the browser's window to see if strange wrapping occurs around the graphic.
How's the performance of loading the page? Are there so many graphics on the page, resulting in a large amount of data to be transferred and displayed, that the website's performance is too slow? What happens if you test on a slow dialup connection instead of your company's fast local area network?

FORMS
Forms are the text boxes, list boxes, and other fields for entering or selecting information on a web page. Figure 14.5 shows a simple example from Apple's website. It's a signup form for potential Mac developers. There are fields for entering your first name, middle initial, last name, and email address. There's an obvious bug on this page - hopefully it's fixed by the time you read this.

FIGURE 14.5. MAKE SURE YOUR WEBSITE'S FORM FIELDS ARE POSITIONED PROPERLY. NOTICE IN THIS APPLE DEVELOPER SIGNUP FORM THAT THE MIDDLE INITIAL (M.I.) FIELD IS MISPLACED.


Apple Developer Connection Member Site - Microsoft...
File Edit View Favorites Tools Help
Address https://connect.apple.com/cgi-bin/WebOb
Go
Back
A D C
Apple Developer Connection
Done
Internet

Welcome to the Apple Developer Connection.
Please enter the following information:

First Name: Test First Name
M. I.: W
Last Name: Test Last Name
E-Mail Address: test@emailaddress.foo

Test forms just as you would if they were fields in a regular software program - remember Chapter 5? Are the fields the correct size? Do they accept the correct data and reject the wrong data? Is there proper confirmation when you finally press Enter? Are optional fields truly optional and the required ones truly required? What happens if you enter 9999999999999999999999999999?

OBJECTS AND OTHER SIMPLE MISCELLANEOUS FUNCTIONALITY

Your website may contain features such as a hit counter, scrolling marquee text, changing advertisements, or internal site searches (not to be confused with search engines that search the entire Web). When planning your tests for a website, take care to identify all the features present on each page. Treat each unique feature as you would a feature in a regular program and test it individually with the standard testing techniques that you've learned. Does it have its own states? Does it handle data? Could it have ranges or boundaries? What test cases apply and how should they be equivalence classed? A web page is just like any other software.

GRAY-BOX TESTING

You're already familiar with black-box and white-box testing, but another type of testing, gray-box testing, is a mixture of the two - hence the name. With gray-box testing, you straddle the line between black-box and white-box testing. You still test the software as a black-box, but you supplement the work by taking a peek (not a full look, as in white-box testing) at what makes the software work.
Web pages lend themselves nicely to gray-box testing. Most web pages are built with HTML (Hypertext Markup Language). Listing 14.1 shows a few lines of the HTML used to create the web page shown in Figure 14.6.

LISTING 14.1. SAMPLE HTML SHOWING SOME OF WHAT'S BEHIND A WEB PAGE

WEB PAGE FUNDAMENTALS

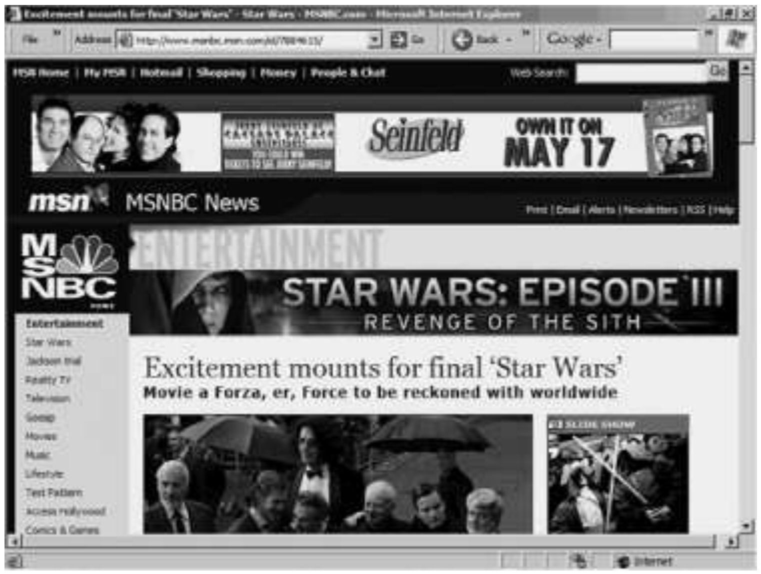
In the simplest terms, Internet web pages are just documents of text, pictures, sounds, video, and hyperlinks - much like the CD-ROM multimedia titles that were popular in the mid 1990s. As in those programs, web users can navigate from page to page by clicking hyperlinked text or pictures, searching for words or phrases, and viewing the information they find.
The Internet, though, has introduced two twists to the multimedia document concept that revolutionizes the technology:

- Unlike data that is stored solely on a CD-ROM, web pages aren't constrained to a single PC. Users can link to and search worldwide across the entire Internet for information on any website.
- Web page authoring isn't limited to programmers using expensive and technical tools. The average person can create a simple web page almost as easily as writing a letter in a word processor.

But, just as giving someone a paint brush doesn't make him an artist, giving someone the ability to create web pages doesn't make him an expert in multimedia publishing. Couple that with the technology explosion that continually adds new website features, and you have the perfect opportunity for a software tester.
Figure 14.1 shows a popular news website that demonstrates many of the possible web page features. A partial list of them includes

- Text of different sizes, fonts, and colors (okay, you can't see the colors in this book)
- Graphics and photos
- Hyperlinked text and graphics
- Rotating advertisements
- Text that has Drop-down selection boxes
- Fields in which the users can enter data

FIGURE 14.1. A TYPICAL WEB PAGE HAS MANY TESTABLE FEATURES.



A great deal of functionality also isn't as obvious, features that make the website much more complex:

- Customizable layout that allows users to change where information is positioned onscreen
- Customizable content that allows users to select what news and information they want to see
- Dynamic drop-down selection boxes
- Dynamically changing text
- Dynamic layout and optional information based on screen resolution
- Compatibility with different web browsers, browser versions, and hardware and software platforms
- Lots of hidden formatting, tagging, and embedded information that enhances the web page's usability

Granted, short of a secure e-commerce website, this is probably one of the more complex and feature-rich web pages on the Internet. If you have the tester mentality (and hopefully you've gained it by reading this far in the book), looking at such a web page should whet your appetite to jump in and start finding bugs. The remainder of this chapter will give you clues on where to look.

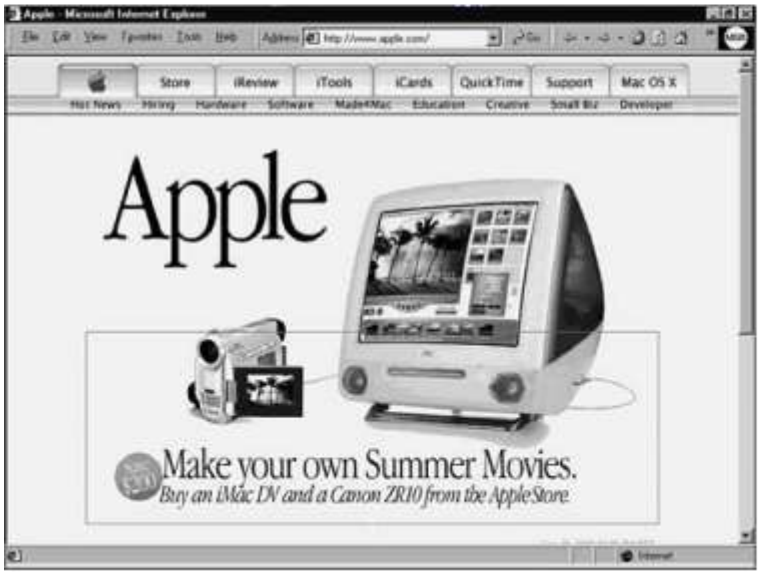
BLACK-BOX TESTING

Remember all the way back to Chapters 4 through 7, the ones that covered the fundamentals of testing? In those vitally important chapters, you learned about black-box, white-box, static, and dynamic testing - the raw skills of a software tester. Web pages are the perfect means to practice what you've learned. You don't have to go out and buy different programs - you can simply jump to a web page, one of your favorites or a completely new one, and begin testing.

The easiest place to start is by treating the web page or the entire website as a black box. You don't know anything about how it works, you don't have a specification, you just have the website in front of you to test. What do you look for?

Figure 14.2 shows a screen image of Apple's website, www.apple.com, a fairly straightforward and typical website. It has all the basic elements - text, graphics, hyperlinks to other pages on the site, and hyperlinks to other websites. A few of the pages have form fields in which users can enter information and a few pages play videos. One interesting thing about this site that's not so common is that it's localized for 27 different locales, from Asia to the UK.

FIGURE 14.2. WHAT WOULD YOU TEST IN A STRAIGHTFORWARD WEBSITE SUCH AS THIS?



If you have access to the Internet, take some time now and explore Apple's website. Think about how you would approach testing it. What would you test? What would your equivalence partitions be? What would you choose not to test?

After exploring a bit, what did you decide? Hopefully you realized that it's a pretty big job. If you looked at the site map (www.apple.com/find/sitemap.html), you found links to more than 100 different sub-sites, each one with several pages.

NOTE

When testing a website, you first should create a state table (see Chapter 5, "Testing the Software with Blinders On"), treating each page as a different state with the hyperlinks as the lines connecting them. A completed state map will give you a better view of the overall task.

Thankfully, most of the pages are fairly simple, made up of just text, graphics, links, and the occasional form. Testing them isn't difficult. The following sections give some ideas of what to look for.

TEXT

Web page text should be treated just like documentation and tested as described in Chapter 12, "Testing the Documentation." Check the audience level, the terminology, the content and subject matter, the accuracy - especially of information that can become outdated - and always, always check spelling.

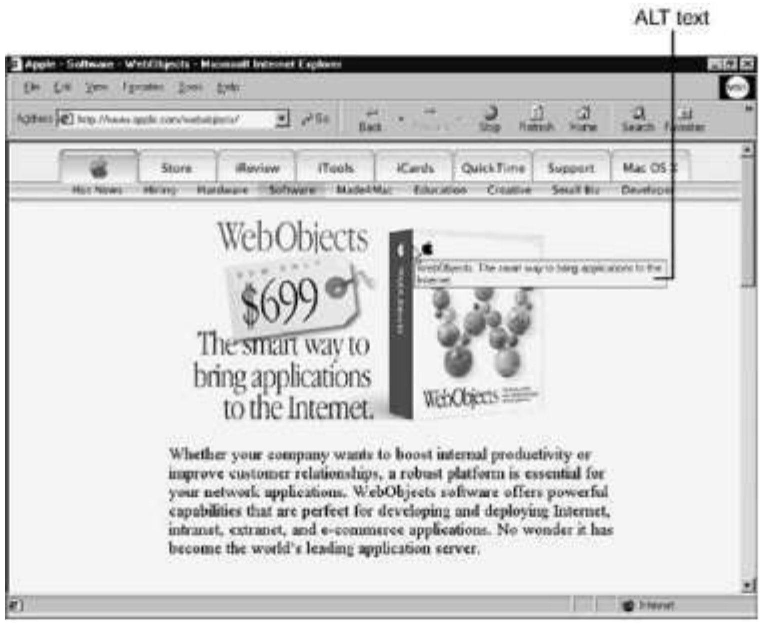
NOTE

Don't rely on spell checkers to be perfect, especially when they're used on web page content. They might only check the regular text but not what's contained in the graphics, scrolling marquees, forms, and so on. You could perform what you think is a complete spell check and still have misspellings on the page.

If there is contact information such as email addresses, phone numbers, or postal addresses, check them to make sure that they're correct. Make sure that the copyright notices are correct and dated appropriately. Test that each page has a correct title. This text appears in the browser's title bar (upper-left corner of Figure 14.2) and what is listed by default when you add the page to your favorites or bookmarks.

An often overlooked type of text is called ALT text, for ALTernate text. Figure 14.3 shows an example of ALT text. When a user puts the mouse cursor over a graphic on the page he gets a pop-up description of what the graphic represents. Web browsers that don't display graphics use ALT text. Also, with ALT text blind users can use graphically rich websites - an audible reader interprets the ALT text and reads it out through the computer's speakers.

FIGURE 14.3. ALT TEXT PROVIDES TEXTUAL DESCRIPTIONS OF GRAPHICS IMAGES ON WEB PAGES.


ALT text
Apple - Software - WebObjects - Microsoft Internet Explorer
Store
iReview
iTools
iCards
QuickTime
Support
Mac OS X
WebObjects
$699
The smart way to bring applications to the Internet.

Whether your company wants to boost internal productivity or improve customer relationships, a robust platform is essential for your network applications. WebObjects software offers powerful capabilities that are perfect for developing and deploying Internet, intranet, extranet, and e-commerce applications. No wonder it has become the world's leading application server.

NOTE

Not all browsers support display of the ALT text. Some only show simple TITLE text in the tooltip - or nothing at all. Since this could prevent the blind from using the Web, it should be considered a serious accessibility bug.

Check for text layout issues by resizing your browser window to be very small or very large. This will reveal bugs where the designer or programmer assumed a fixed page width or height. It will also reveal hard-coded formatting such as line breaks that might look good with certain layouts but not with others.

HYPERLINKS

Links can be tied to text or graphics. Each link should be checked to make sure that it jumps to the correct destination and opens in the correct window. If you don't have a specification for the website, you'll need to test whether the jump worked correctly.

Make sure that hyperlinks are obvious. Text links are usually underlined, and the mouse pointer should change (usually to a hand pointer) when it's over any kind of hyperlink - text or graphic.

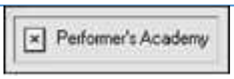
If the link opens up an email message, fill out the message, send it, and make sure you get a response.

Look for orphan pages, which are part of the website but can't be accessed through a hyperlink because someone forgot to hook them up. You'll likely need to get a list from the web page's designer of the expected pages and compare that with your own state table.

GRAPHICS

Many possible bugs with graphics are covered later under usability testing, but you can check a few obvious things with a simple black-box approach. For example, do all graphics load and display properly? If a graphic is missing or is incorrectly named, it won't load and the web page will display an error where the graphic was to be placed (see Figure 14.4).

FIGURE 14.4. IF A GRAPHIC CAN'T LOAD ONTO A WEB PAGE AN ERROR IS PUT IN ITS LOCATION.



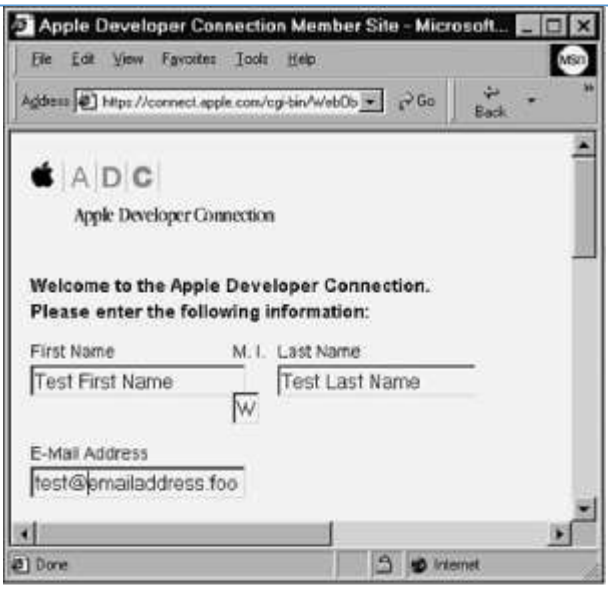
If text and graphics are intermixed on the page, make sure that the text wraps properly around the graphics. Try resizing the browser's window to see if strange wrapping occurs around the graphic.

How's the performance of loading the page? Are there so many graphics on the page, resulting in a large amount of data to be transferred and displayed, that the website's performance is too slow? What happens if you test on a slow dialup connection instead of your company's fast local area network?

FORMS

Forms are the text boxes, list boxes, and other fields for entering or selecting information on a web page. Figure 14.5 shows a simple example from Apple's website. It's a signup form for potential Mac developers. There are fields for entering your first name, middle initial, last name, and email address. There's an obvious bug on this page - hopefully it's fixed by the time you read this.

FIGURE 14.5. MAKE SURE YOUR WEBSITE'S FORM FIELDS ARE POSITIONED PROPERLY. NOTICE IN THIS APPLE DEVELOPER SIGNUP FORM THAT THE MIDDLE INITIAL (M.I.) FIELD IS MISPLACED.


Apple Developer Connection Member Site - Microsoft...

File Edit View Favorites Tools Help

Address https://connect.apple.com/cgi-bin/WebOb

A D C

Apple Developer Connection

Welcome to the Apple Developer Connection.
Please enter the following information:

First Name: Test First Name
M. I.: W
Last Name: Test Last Name

E-Mail Address: test@emailaddress.foo

Done | Internet

Test forms just as you would if they were fields in a regular software program - remember Chapter 5? Are the fields the correct size? Do they accept the correct data and reject the wrong data? Is there proper confirmation when you finally press Enter? Are optional fields truly optional and the required ones truly required? What happens if you enter 9999999999999999999999999999?

OBJECTS AND OTHER SIMPLE MISCELLANEOUS FUNCTIONALITY

Your website may contain features such as a hit counter, scrolling marquee text, changing advertisements, or internal site searches (not to be confused with search engines that search the entire Web). When planning your tests for a website, take care to identify all the features present on each page. Treat each unique feature as you would a feature in a regular program and test it individually with the standard testing techniques that you've learned. Does it have its own states? Does it handle data? Could it have ranges or boundaries? What test cases apply and how should they be equivalence classed? A web page is just like any other software.

GRAY-BOX TESTING

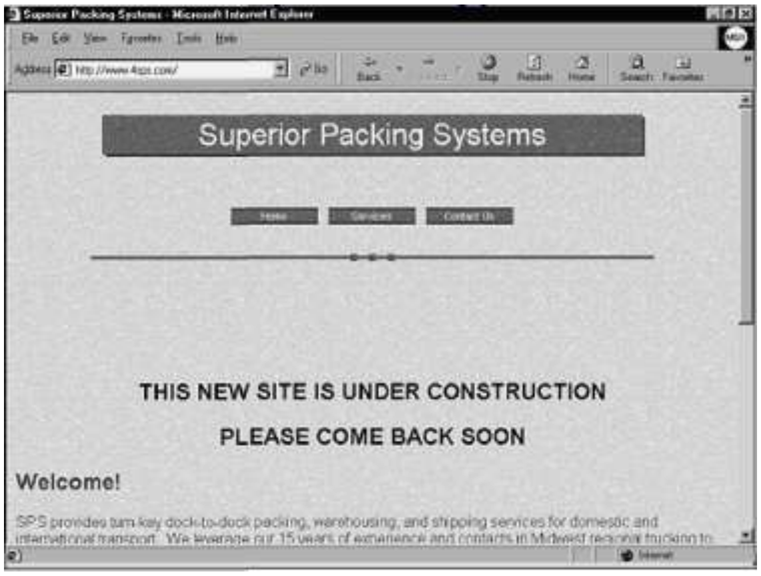
You're already familiar with black-box and white-box testing, but another type of testing, gray-box testing, is a mixture of the two - hence the name. With gray-box testing, you straddle the line between black-box and white-box testing. You still test the software as a black-box, but you supplement the work by taking a peek (not a full look, as in white-box testing) at what makes the software work.

Web pages lend themselves nicely to gray-box testing. Most web pages are built with HTML (Hypertext Markup Language). Listing 14.1 shows a few lines of the HTML used to create the web page shown in Figure 14.6.

LISTING 14.1. SAMPLE HTML SHOWING SOME OF WHAT'S BEHIND A WEB PAGE

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
<meta name="GENERATOR" content="Microsoft FrontPage 4.0">
<title>Superior Packing Systems</title>
<meta name="Microsoft Theme" content="sandston 111, default">
<meta name="Microsoft Border" content="t, default">
</head>
<body background="_themes/sandston/stonbk.jpg" bgcolor="#FFFFCC"
  text="#333333" link="#993300" vlink="#666633" alink="#CC6633">
  <!--msnavigation--><table border="0" cellpadding="0" cellspacing="0"
  width="100%"><tr><td><!--mstheme--><font face="Arial, Helvetica">
<h1 align="center"><!--mstheme--><font color="#660000">
  <img src="_derived/index.htm_cmp_sandston110_bnr.gif" width="600"
  height="60" border="0" alt="Superior Packing Systems"><br>
&nbsp; <br>
<a href="./"><img src="_derived/home_cmp_sandston110_gbtn.gif" width="95"
  height="20" border="0" alt="Home" align="middle"></a> <a href="services.ht
  <img src="_derived/services.htm_cmp_sandston110_gbtn.gif" width="95"
  height="20" border="0" alt="Services" align="middle"></a>
  <a href="contact.htm"><img src="_derived/contact.htm_cmp_sandston110_gbtn.
  width="95" height="20" border="0" alt="Contact Us" align="middle">
  </a><!--mstheme--></font></h1>
```

FIGURE 14.6. PART OF THIS WEB PAGE IS CREATED BY THE HTML IN LISTING 14.1.


Superior Packing Systems - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.4sps.com/
Back Stop Refresh Home Search Favorites
Home Services Contact Us
Internet

Superior Packing Systems

THIS NEW SITE IS UNDER CONSTRUCTION

PLEASE COME BACK SOON

Welcome!

SPS provides turn-key dock-to-dock packing, warehousing, and shipping services for domestic and international transport. We leverage our 15 years of experience and contacts in Midwest regional trucking to

NOTE

If you're not familiar with creating your own website, you might want to read a little on the subject. An introductory book such as Sams Teach Yourself to Create Web Pages in 24 Hours, would be a great way to learn the basics and help you discover a few ways to apply gray-box testing techniques.

HTML and web pages can be simply tested as a gray box because HTML isn't a programming language that's been compiled and inaccessible to the tester - it's a markup language. In the early days of word processors, you couldn't just select text and make it bold or italic. You had to embed markups, sometimes called field tags, in the text. For example, to create the bolded phrase

    This is bold text.

you would enter something such as this into your word processor:

    [begin bold]This is bold text.[end bold]

HTML works the same way. To create the line in HTML you would enter

This is bold text.

HTML has evolved to where it now has hundreds of different field tags and options, as evidenced by the HTML in Listing 14.1. But, in the end, HTML is nothing but a fancy old-word-processor-like markup language. The difference between HTML and a program is that HTML doesn't execute or run, it just determines how text and graphics appear onscreen.

TIP

To see the HTML for a web page, right-click a blank area of the page (not on a graphic) and select View Source (IE) or View Page Source (Firefox) from the menu.

Since HTML isn't a programming language and is so easy for you, as the tester, to view, you might as well take advantage of it and supplement your testing. If you're a black-box tester, it's the perfect opportunity to start moving toward white-box testing.

Start by learning to create your own simple web pages. Learn the basic and common HTML tags. Look at the HTML for many different pages on the Web, see what techniques are used and how those techniques make things work on the page. Once you become familiar with HTML, you'll be able to look at the web pages you're testing in a whole new way and be a more effective tester.

WHITE-BOX TESTING

In Figure 14.1, you saw an example of a web page with much static content in the form of text and images. This static content was most likely created with straight HTML. That same web page also has customizable and dynamic changing content. Remember, HTML isn't a programming language - it's merely a tagging system for text and graphics. To create these extra dynamic features requires the HTML to be supplemented with programming code that can execute and follow decision paths.

You've likely heard of the popular web programming languages and technologies that can be used to create these types of features: DHTML, Java, JavaScript, ActiveX, VBScript, Perl, CGI, ASP, and XML. As explained in Chapters 6, "Examining the Code," and 7, "Testing the Software with X-Ray Glasses," to apply white-box testing, you don't necessarily need to become an expert in these languages, just familiar enough to be able to read and understand them and to devise test cases based on what you see in the code.

This chapter can't possibly go into all the details of white-box testing a website, but several features could be more effectively tested with a white-box approach. Of course, they could also be tested as a black-box, but the potential complexity is such that to really make sure you find the important bugs that you have some knowledge of the website's system structure and programming:

- Dynamic Content. Dynamic content is graphics and text that changes based on certain conditions - for example, the time of day, the user's preferences, or specific user actions. It's possible that the programming for the content is done in a simple scripting language such as JavaScript and is embedded within the HTML. This is known as client-side programming. If it is, you can apply gray-box testing techniques when you examine the script and view the HTML. For efficiency, most dynamic content programming is located on the website's server; it's called server-side programming and would require you to have access to the web server to view the code.
- Database-Driven Web Pages. Many e-commerce web pages that show catalogs or inventories are database driven. The HTML provides a simple layout for the web content and then pictures, text descriptions, pricing information, and so on are pulled from a database on the website's server and plugged into the pages.
- Programmatically Created Web Pages. Many web pages, especially ones with dynamic content, are programmatically generated - that is, the HTML and possibly even the programming is created by software. A web page designer may type entries in a database and drag and drop elements in a layout program, press a button, and out comes the HTML that displays a web page. If this sounds scary, it's really no different than a computer language compiler creating machine code. If you're testing such a system, you have to check that the HTML it creates is what the designer expects.
- Server Performance and Loading. Popular websites might receive millions of individual hits a day. Each one requires a download of data from the website's server to the browser's computer. If you wanted to test a system for performance and loading, you'd have to find a way to simulate the millions of connections and downloads. Chapter 15, "Automated Testing and Test Tools," introduces the techniques and tools you can use to do this.

- Security. As you learned in the previous chapter, website security issues are always in the news as hackers try new and different ways to gain access to a website's internal data. Financial, medical, and other websites that contain personal data are especially at risk and require intimate knowledge of server technology to test them for proper security.

CONFIGURATION AND COMPATIBILITY TESTING

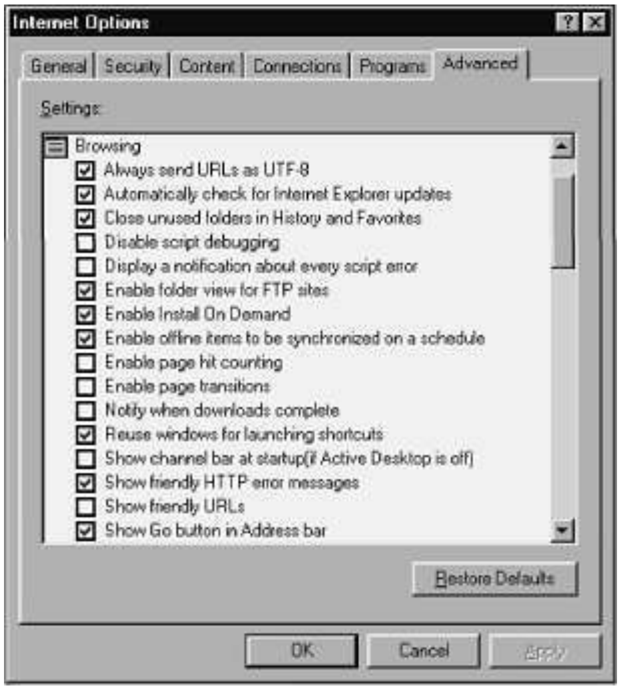
It's time to get back to what you can do, today, to test a web page. Recall from Chapters 8, "Configuration Testing," and 9, "Compatibility Testing," what configuration and compatibility testing are. Configuration testing is the process of checking the operation of your software with various types of hardware and software platforms and their different settings. Compatibility testing is checking your software's operation with other software. Web pages are perfect examples of where you can apply this type of testing.
Assume that you have a website to test. You need to think about what the possible hardware and software configurations might be that could affect the operation or appearance of the site. Here's a list to consider:

- Hardware Platform. Is it a Mac, PC, PDA, MSNTV, or a WiFi wristwatch? Each hardware device has its own operating system, screen layout, communications software, and so on. Each can affect how the website appears onscreen.
- Browser Software and Version. There are many different web browsers and browser versions. Some run on only one type of hardware platform, others run on multiple platforms. Some examples are AOL 9.0, Firefox 1.0, Internet Explorer 5.0 and 6.0, Pocket IE, Netscape 7.2, and Opera 7.54.

  Each browser and version supports a different set of features. A website may look great under one browser and not display at all under another. Web designers can choose to design a site using the least common denominator of features so that it looks the same on all of them, or write specialized code to make the site work best on each one. They may even choose to only support a few of the most popular browsers. How would this impact your testing?

- Browser Plug-Ins. Many browsers can accept plug-ins or extensions to gain additional functionality. An example of this would be to play specific types of audio or video files.
- Browser Options. Most web browsers allow for a great deal of customization. Figure 14.7 shows an example of this. You can select security options, choose how ALT text is handled, decide what plug-ins to enable, and so on. Each option has potential impact on how your website operates - and, hence, is a test scenario to consider.

FIGURE 14.7. THIS EXAMPLE SHOWS HOW CONFIGURABLE THE INTERNET EXPLORER WEB BROWSER IS.



Browsing
- ☑ Always send URLs as UTF-8
- ☑ Automatically check for Internet Explorer updates
- ☑ Close unused folders in History and Favorites
- ☐ Disable script debugging
- ☐ Display a notification about every script error
- ☑ Enable folder view for FTP sites
- ☑ Enable Install On Demand
- ☑ Enable offline items to be synchronized on a schedule
- ☐ Enable page hit counting
- ☐ Enable page transitions
- ☐ Notify when downloads complete
- ☑ Reuse windows for launching shortcuts
- ☐ Show channel bar at startup(if Active Desktop is off)
- ☑ Show friendly HTTP error messages
- ☐ Show friendly URLs
- ☑ Show Go button in Address bar

- Video Resolution and Color Depth. Many platforms can display in various screen resolutions and colors. A PC running Windows, for example, can have screen dimensions of 640x480, 800x600, 1,024x768, 1280x1024, and up. Mobile devices have tiny screens with very different resolutions. Your website may look different, or even wrong, in one resolution, but not in another. Text and graphics can wrap differently, be cut off, or not appear at all.

  The number of colors that the platform supports can also impact the look of your site. PCs typically support as few as 256 colors and as many as 2^{32}. Could your website be used on a mobile system with only 16 colors?

- Text Size. Did you know that a user can change the size of the text used in the browser? Could your site be used with very small or very large text? What if it was being run on a small screen, in a low resolution, with large text?
- Modem Speeds. Enough can't be said about performance. Someday everyone will have high-speed connections with website data delivered as fast as you can view it. Until then, you need to test that your website works well at a wide range of modem speeds.

If you consider all the possibilities outlined here, testing even the simplest website can become a huge task. It's not enough that the website looks good on your PC - if you want to ensure that it works well for its intended audience, you need to research the possible configurations they might have. With that information, you can create equivalence partitions of the configurations you feel are most important to test.
A good place to start your research is www.websidestory.com and www.upsdell.com/BrowserNews/stat.htm. These sites have frequently updated surveys related to technology, browsers, video resolutions, etc. These are a great first step in deciding what configurations are popular and which direction they are trending.