Software Testing
By Ron Patton
.....
Publisher: **Sams Publishing**
Pub Date: **July 26, 2005**
ISBN: **0-672-32798-8**
Pages: **408**

Software Testing Background

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, 19941995

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

NASA Mars Polar Lander, 1999

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

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

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

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

The result was catastrophic, but the reason behind it was simple. The lander was tested by multiple teams. One team tested the leg fold-down procedure and another the landing process from that point on. The first team never looked to see if the touch-down bit was setit 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 Defence System, 1991

The U.S. Patriot missile defence system is a scaled-back version of the Strategic Defence Initiative ("Star Wars") program proposed by President Ronald Reagan. It was first put to use in the Gulf War as a defence 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 wrecking 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:

Variance Fault Failure Problem Inconsistency

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 coloured 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.

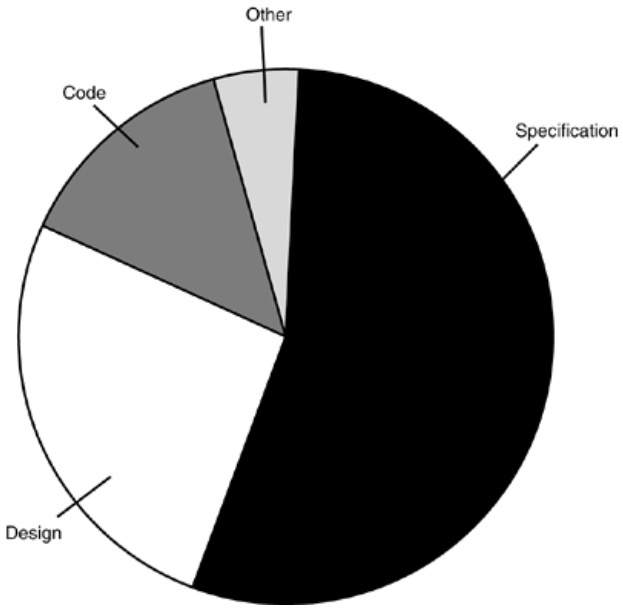
Rule #5 is the catch-all. As a tester you are the first person to really use the software. If you weren't there, it would be the customer using the product for the first time. If you find something that you don't feel is right, for whatever reason, it's a bug. In the case of the calculator, maybe you 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.

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.

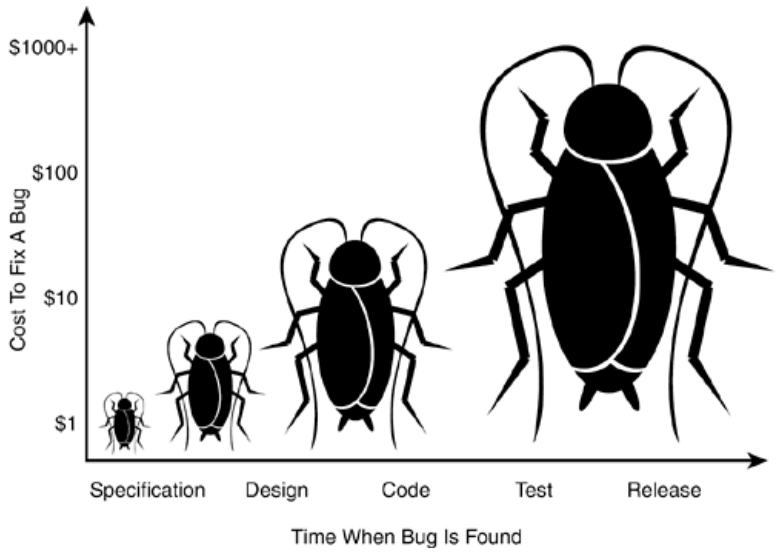
These are greatly simplified examples, so think about how the rules apply to software that you use every day. What is expected, what is unexpected? What do you think was specified and what was forgotten? And, what do you just plain dislike about the software?

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.

The Cost of Bugs

As you will learn in Chapter 2, software doesn't just magically appearthere'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.

Figure 1.2. The cost to fix bugs can increase dramatically over time.



The costs are logarithmic. That is, they increase tenfold as time increases. A bug found and fixed during the early stages when the specification is being written might cost next to nothing, or $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 bugsyou 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.