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 [redacted] differ
- Wha [redacted]
- 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, sometimes called a *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 with the mouse, using access-keys (Alt+C for Copy and Alt+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.

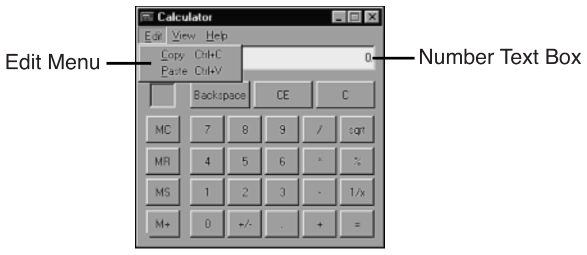

FIGURE 4.1
The standard Windows Calculator displaying the drop-down Edit menu.

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. [redacted] 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.

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.

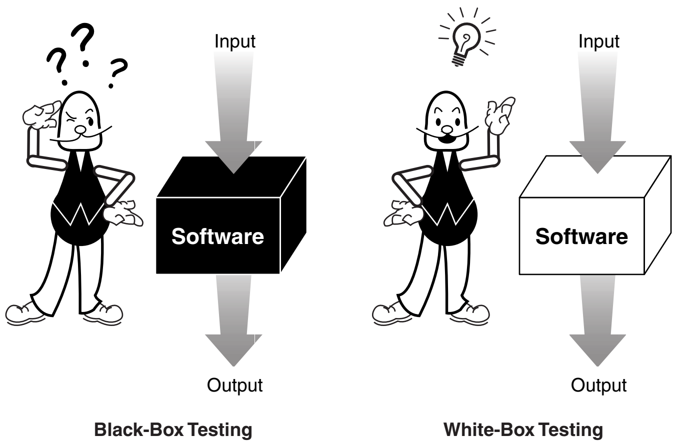

FIGURE 4.2
With black-box testing, the software tester doesn't know the details of how the software works.

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.

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*.

.

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. [redacted]

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.

NOTE

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. [redacted]

Performing a [redacted] 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 [redacted]

The first step in testing the specification isn't to jump in and look for specific bugs. [redacted] 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.

[redacted]

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. Guidelines are optional but should be followed.

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.
- [redacted] The medical, pharmaceutical, industrial, and financial industries have very strict standards that their software must follow.
- [redacted]. The government, especially the military, has strict standards.
- [redacted] 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.
- [redacted] Low-level software and hardware interface standards must be adhered to, to assure compatibility across systems.

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.

[redacted]

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 your software be smaller or larger? Will that size make a difference in your testing?
- [redacted] Will your software be more or less complex? Will this impact your testing?
- [redacted] Will you have the resources, time, and expertise to test software such as this?
- [redacted] Is this software representative of the overall quality planned for your software? Will your software be more or less reliable?

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.

[redacted] 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 mov[redacted] on to testing the specification at a lower level. The remainder of this chapter explains the specifics for doing this.[1]

[redacted]

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

- [redacted] Is anything missing or forgotten? Is it thorough? Does it include everything necessary to make it stand alone?
- [redacted] Is the proposed solution correct? Does it properly define the goal? Are there any errors?
- [redacted] Is the description exact and not vague? Is there a single interpretation? Is it easy to read and understandable?
- [redacted] Is the description of the feature written so that it doesn't conflict with itself or other items in the specification?
- [redacted]. 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?

[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.*

- **Feasible.** Can the feature be implemented with the available personnel, tools, and resources within the specified budget and schedule?
- [redacted] Does the specification stick with defining the product and not the underlying software design, architecture, and code?
- [redacted]. 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.

[redacted]

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.

- [redacted] 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.
- [redacted] These words tend to persuade you into accepting something as a given. Don't fall into the trap.
- [redacted] These words are too vague. It's impossible to test a feature that operates "sometimes."
- [redacted]. 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.
- [redacted]. 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.
- [redacted]. These terms can hide large amounts of functionality that need to be specified.
- [redacted] 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.