

5

Testing the Software with Blinders On

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

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

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

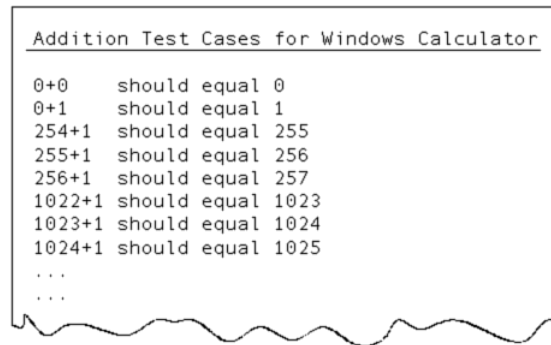
- What is dynamic black-box testing?
- How to reduce the number of test cases by equivalence partitioning
- How to identify troublesome boundary conditions
- Good data values to use to induce bugs
- How to test software states and state transitions
- How to use repetition, stress, and high loads to locate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



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
...		
...		

FIGURE 5.1 Test cases show the different inputs and the steps to test a program.

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.

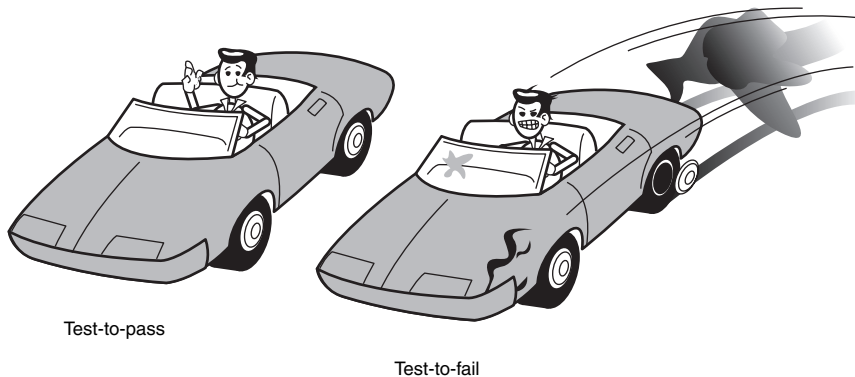


FIGURE 5.2 Use test-to-pass to reveal bugs before you test-to-fail.

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

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) .

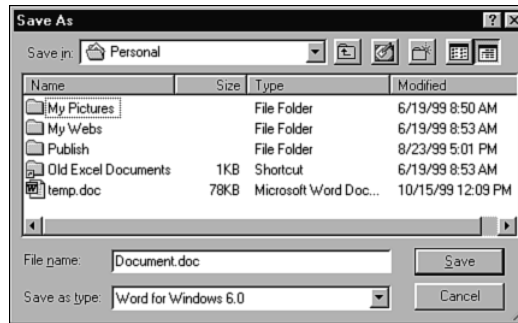


FIGURE 5.4 The File Name text box in the Save As dialog box illustrates several equivalence partition possibilities.

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.

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.



FIGURE 5.5 A software boundary is much like the edge of a cliff.

LISTING 5.1 A Simple BASIC Program Demonstrating a Boundary Condition Bug

```
1: Rem Create a 10 element integer array
2: Rem Initialize each element to -1
3: Dim data(10) As Integer
4: Dim i As Integer
5: For i = 1 To 10
6:   data(i) = -1
7: Next i
8: 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
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:

- First-1/Last+1
- Start-1/Finish+1

- Less than Empty/More than Full
- Even Slower/Even Faster
- Largest+1/Smallest-1
- Min-1/Max+1
- Just Over/Just Under
- Even Shorter/Longer
- Even Sooner/Later
- Highest+1/Lowest-1

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
Nibble	0–15
Byte	0–255
Word	0–4,294,967,295
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 A–Z and a–z, 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?

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.

TIP

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

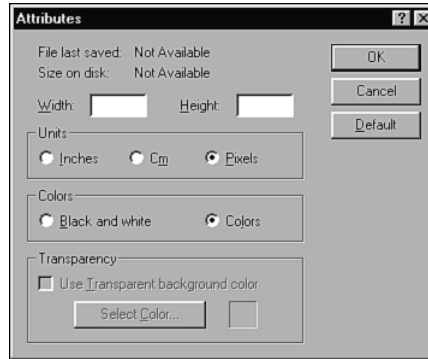


FIGURE 5.6 The Windows Paint Attributes dialog box with the Width and Height text fields blanked out.

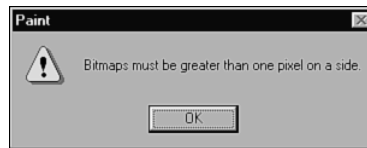


FIGURE 5.7 The error message returned if Enter is pressed with the Width and Height text fields blanked out.

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.

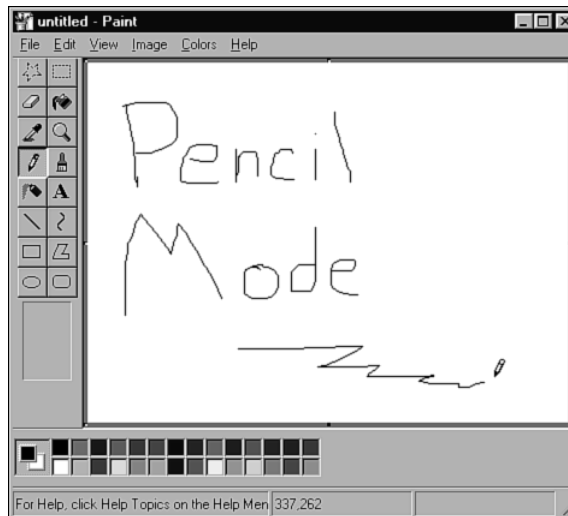


FIGURE 5.8 The Windows Paint program in the pencil drawing state.

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.



FIGURE 5.9 The Windows Paint program in the airbrushing state.

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.

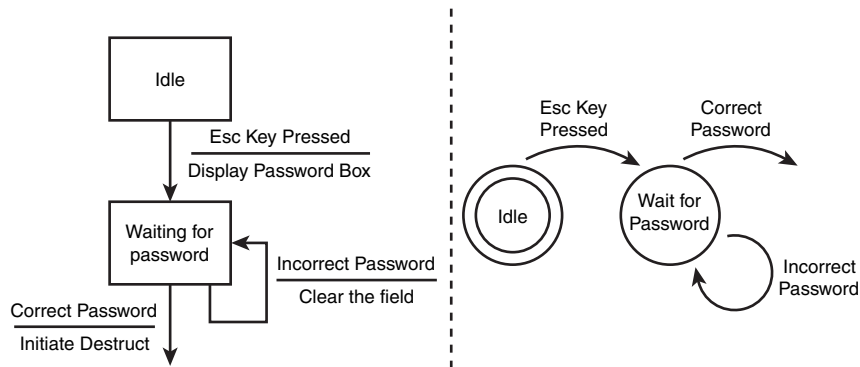


FIGURE 5.10 State transition diagrams can be drawn by using different techniques.

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.

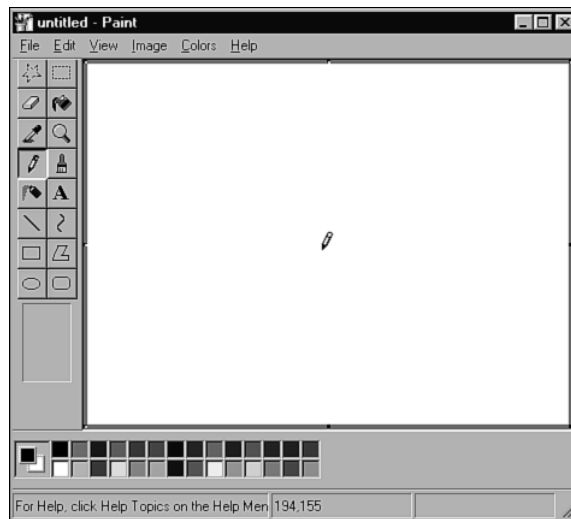


FIGURE 5.11 The Windows Paint opening screen in the startup state.

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

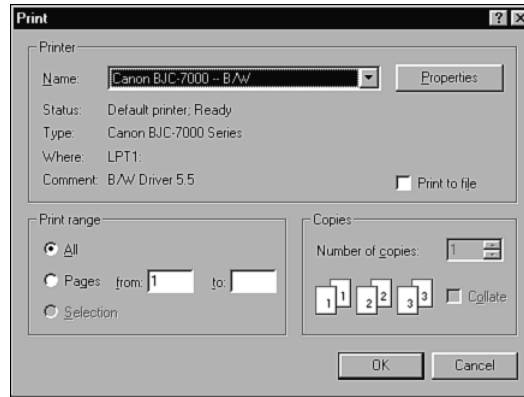


FIGURE 5.12 The Windows Print dialog box showing the Print Range feature.

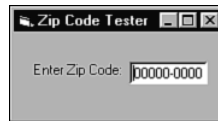


FIGURE 5.13 A sample ZIP code text box that holds up to 10 characters.

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.