So far in Part II you've learned about three of the four fundamental testing techniques: static black box (testing the specification), dynamic black box (testing the software), and static white box (examining the code). In this chapter, you'll learn the fourth fundamental technique—dynamic white-box testing. You'll look into the software "box" with your X-ray glasses as you test the software.

In addition to your X-ray specs, you'll also need to wear your programmer's hat—if you have one. If you don't own one, don't be scared off. The examples used aren't that complex and if you take your time, you'll be able to follow them. Gaining even a small grasp of this type of testing will make you a much more effective black-box tester.

If you do have some programming experience, consider this chapter an introduction to a very wide-open testing field. Most software companies are hiring testers specifically to perform low-level testing of their software. They're looking for people with both programming and testing skills, which is often a rare mix and highly sought after.

Highlights from this chapter include

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

Dynamic White-Box Testing

By now you should be very familiar with the terms *static*, *dynamic*, *white box*, and *black box*. Knowing that this chapter is about dynamic white-box testing should tell you exactly what material it covers. Since it's dynamic, it must be about testing a running program and since it's white-box, it must be about looking inside the box, examining the code, and watching it as it runs. It's like testing the software with X-ray glasses.

Dynamic white-box testing, in a nutshell, is using information you gain from seeing what the code does and how it works to determine what to test, what not to test, and how to approach the testing. Another name commonly used for dynamic white-box testing is *structural testing* because you can see and use the underlying structure of the code to design and run your tests.

Why would it be beneficial for you to know what's happening inside the box, to understand how the software works? Consider Figure 7.1. This figure shows two boxes that perform the basic calculator operations of addition, subtraction, multiplication, and division.

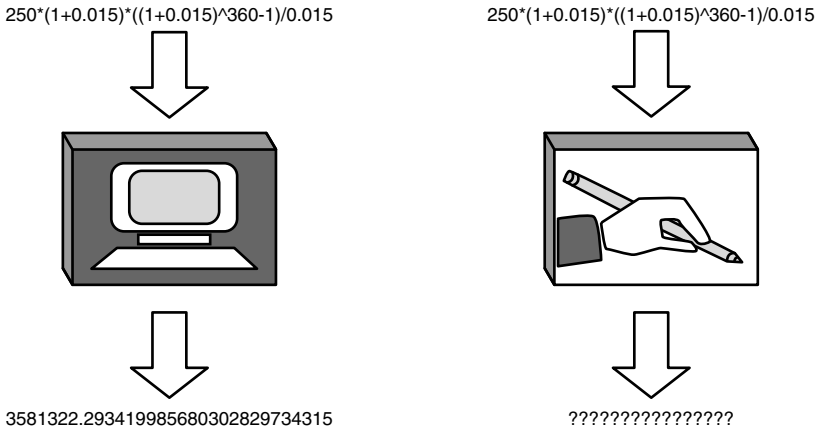


FIGURE 7.1
You would choose different test cases if you knew that one box contained a computer and the other a person with a pencil and paper.

If you didn't know how the boxes worked, you would apply the dynamic black-box testing techniques you learned in Chapter 5, "Testing the Software with Blinders On." But, if you could look in the boxes and see that one contained a computer and the other contained a person with a pencil and paper, you would probably choose a completely different test approach for each one. Of course, this example is very simplistic, but it makes the point that knowing how the software operates will influence how you test.

Dynamic white-box testing isn't limited just to seeing what the code does. It also can involve directly testing and controlling the software. The four areas that dynamic white-box testing encompasses are

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

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

Dynamic White-Box Testing versus Debugging

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

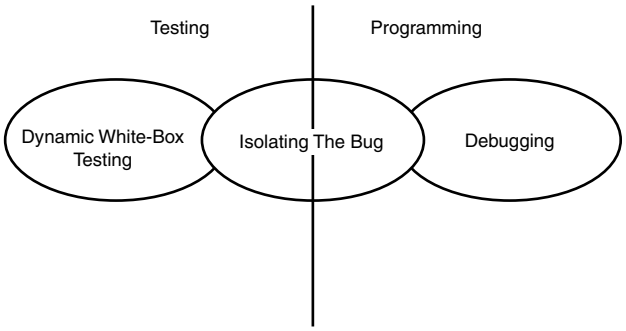


FIGURE 7.2
Dynamic white-box testing and debugging have different goals but they do overlap in the middle.

The goal of dynamic white-box testing is to find bugs. The goal of debugging is to fix them. They do overlap, however, in the area of isolating where and why the bug occurs. You'll learn more about this in Chapter 18, "Reporting What You Find," but for now, think of the overlap this way. As a software tester, you should narrow down the problem to the simplest test case that demonstrates the bug. If it's white-box testing, that could even include information about what lines of code look suspicious. The programmer who does the debugging picks the process up from there, determines exactly what is causing the bug, and attempts to fix it.

NOTE

It's important to have a clear separation between your work and the programmer's work. Programmers write the code, testers find the bugs and may need to write some code to drive their tests, and programmers fix the bugs. Without this separation, issues can arise where tasks are overlooked or work is duplicated.

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

Testing the Pieces

Recall from Chapter 2, "The Software Development Process," the various models for software development. The big-bang model was the easiest but the most chaotic. Everything was put together at once and, with fingers crossed, the team hoped that it all worked and that a product would be born. By now you've probably deduced that testing in such a model would be very difficult. At most, you could perform dynamic black-box testing, taking the product in one entire blob and exploring it to see what you could find.

You've learned that this approach is very costly because the bugs are found late in the game. From a testing perspective, there are two reasons for the high cost:

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

Unit and Integration Testing

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

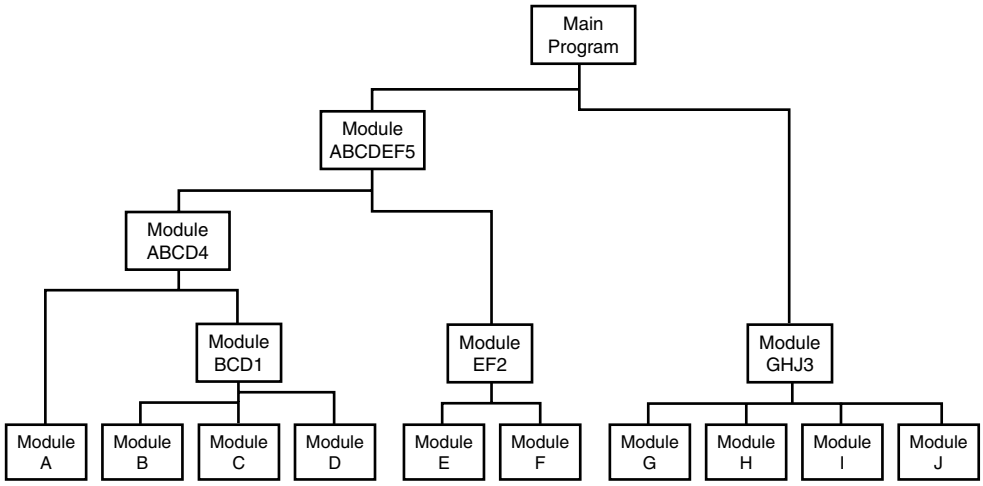


FIGURE 7.3
Individual pieces of code are built up and tested separately, and then integrated and tested again.

Testing that occurs at the lowest level is called *unit testing* or *module testing*. As the units are tested and the low-level bugs are found and fixed, they are integrated and *integration testing* is performed against groups of modules. This process of incremental testing continues, putting together more and more pieces of the software until the entire product—or at least a major portion of it—is tested at once in a process called *system testing*.

With this testing strategy, it's much easier to isolate bugs. When a problem is found at the unit level, the problem must be in that unit. If a bug is found when multiple units are integrated, it must be related to how the modules interact. Of course, there are exceptions to this, but by and large, testing and debugging is much more efficient than testing everything at once.

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

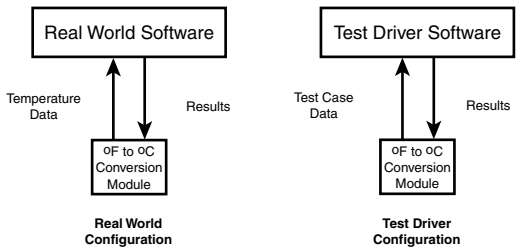


FIGURE 7.4
A test driver can replace the real software and more efficiently test a low-level module.

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

Rather than test the temperature display module by attempting to control the temperature of the thermometer, you could write a small piece of code called a *stub* that acts just like the interface module by feeding temperature values from a file directly to the display module. The display module would read the data and show the temperature just as though it was reading directly

from a real thermometer interface module. It wouldn't know the difference. With this test stub configuration, you could quickly run through numerous test values and validate the operation of the display module.

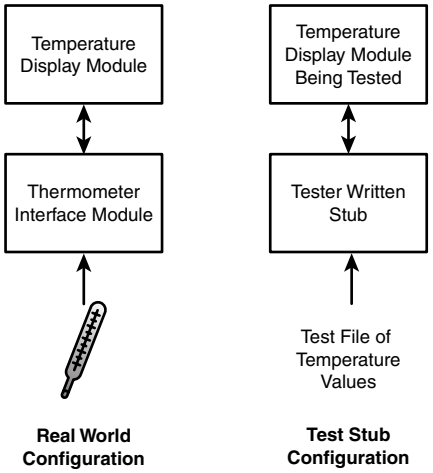


FIGURE 7.5
A test stub sends test data up to the module being tested.

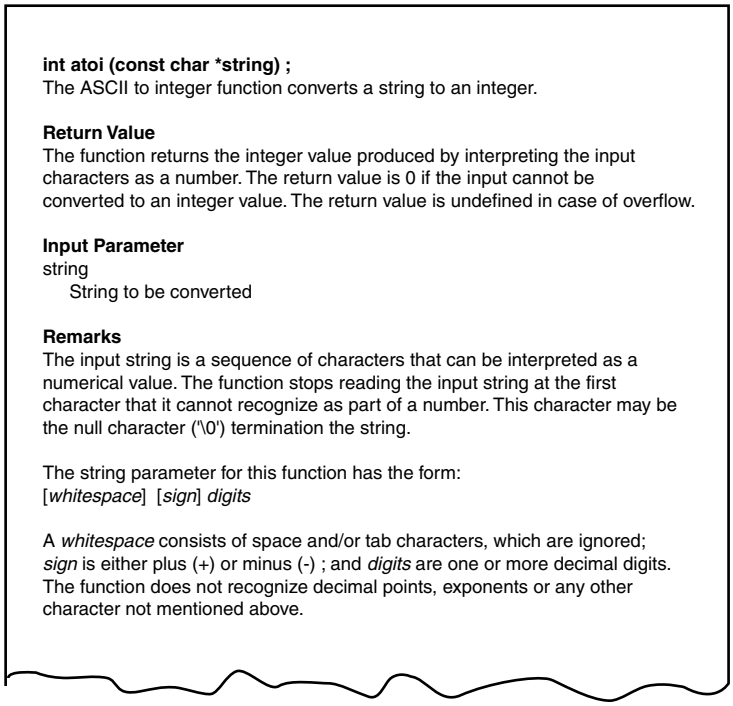
An Example of Unit Testing

A common function available in many compilers is one that converts a string of ASCII characters into an integer value.

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

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

If you're the software tester assigned to perform dynamic white-box testing on this module, what would you do?



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

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

Input Parameter
string
String to be converted

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

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

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

FIGURE 7.6
The specification sheet for the C language `atoi()` *function.*

First, you would probably decide that this module looks like a bottom module in the program, one that's called by higher up modules but doesn't call anything itself. You could confirm this by looking at the internal code. If this is true, the logical approach is to write a test driver to exercise the module independently from the rest of the program.

This test driver would send test strings that you create to the `atoi()` function, read back the return values for those strings, and compare them with your expected results. The test driver would most likely be written in the same language as the function—in this case, C—but it's also possible to write the driver in other languages as long as they interface to the module you're testing.

This test driver can take on several forms. It could be a simple dialog box, as shown in Figure 7.7, that you use to enter test strings and view the results. Or it could be a standalone program that reads test strings and expected results from a file. The dialog box, being user driven, is very interactive and flexible—it could be given to a black-box tester to use. But the standalone driver can be very fast reading and writing test cases directly from a file.

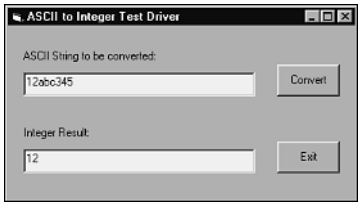


FIGURE 7.7
A dialog box test driver can be used to send test cases to a module being tested.

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

TABLE 7.1 Sample ASCII to Integer Conversion Test Cases

Input String	*Output Integer Value*
"1"	1
"-1"	–1
"+1"	1
"0"	0
"-0"	0
"+0"	0
"1.2"	1
"2-3"	2
"abc"	0
"a123"	0
and so on	

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

NOTE

Creating your black-box testing cases based on the specification, before your white-box cases, is important. That way, you are truly testing what the module is intended

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

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

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

Data Coverage

The previous example of white-box testing the `atoi()` function was greatly simplified and glossed over some of the details of looking at the code to decide what adjustments to make to the test cases. In reality, there's quite a bit more to the process than just perusing the software for good ideas.

The logical approach is to divide the code just as you did in black-box testing—into its data and its states (or program flow). By looking at the software from the same perspective, you can more easily map the white-box information you gain to the black-box cases you've already written.

Consider the data first. Data includes all the variables, constants, arrays, data structures, keyboard and mouse input, files and screen input and output, and I/O to other devices such as modems, networks, and so on.

Data Flow

Data flow coverage involves tracking a piece of data completely through the software. At the unit test level this would just be through an individual module or function. The same tracking could be done through several integrated modules or even through the entire software product—although it would be more time-consuming to do so.

If you test a function at this low level, you would use a debugger and watch variables to view the data as the program runs (see Figure 7.8). With black-box testing, you only know what the value of the variable is at the beginning and at the end. With dynamic white-box testing you could also check intermediate values during program execution. Based on what you see you might decide to change some of your test cases to make sure the variable takes on interesting or even risky interim values.

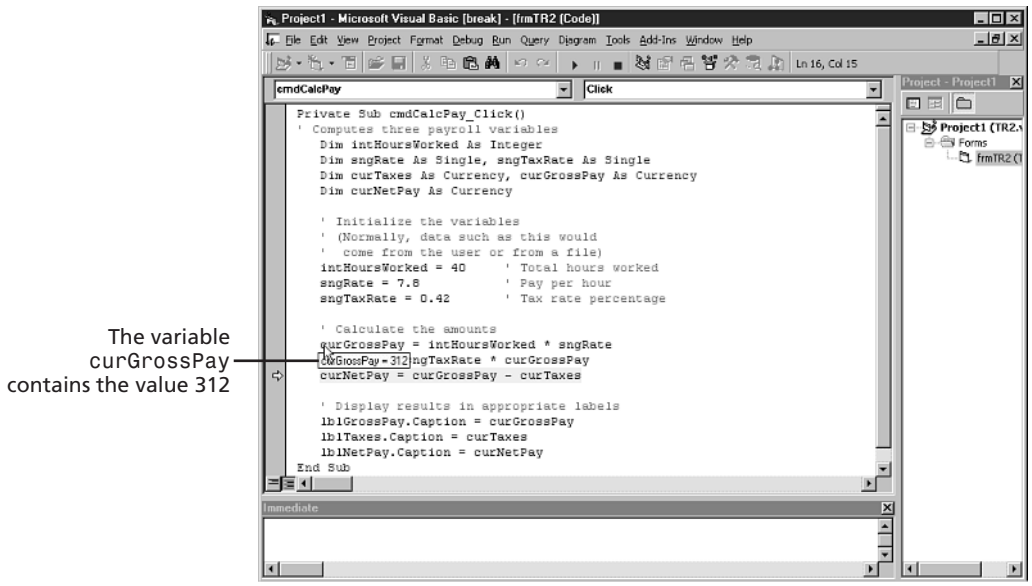


FIGURE 7.8
A debugger and watch variables can help you trace a variable's values through a program.

Sub-Boundaries

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

- A module that computes taxes might switch from using a data table to using a formula at a certain financial cut-off point.
- An operating system running low on RAM may start moving data to temporary storage on the hard drive. This sub-boundary may not even be fixed. It may change depending on how much space remains on the disk.

- To gain better precision, a complex numerical analysis program may switch to a different equation for solving the problem depending on the size of the number.

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

Formulas and Equations

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

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

where

P = principal amount

r = annual interest rate

n = number of times the interest is compounded per year

t = number of years

A = amount after time t

A good black-box tester would hopefully choose a test case of n=0, but a white-box tester, after seeing the formula in the code, would know to try n=0 because that would cause the formula to blow up with a divide-by-zero error.

But, what if n was the result of another computation? Maybe the software sets the value of n based on other user input or algorithmically tries different n values in an attempt to find the lowest payment. You need to ask yourself if there's any way that n can ever become zero and figure out what inputs to feed the program to make that happen.

TIP

Scour your code for formulas and equations, look at the variables they use, and create test cases and equivalence partitions for them in addition to the normal inputs and outputs of the program.

Error Forcing

The last type of data testing covered in this chapter is *error forcing*. If you're running the software that you're testing in a debugger, you don't just have the ability to watch variables and see what values they hold—you can also force them to specific values.

In the preceding compound interest calculation, if you couldn't find a direct way to set the number of compoundings (n) to zero, you could use your debugger to force it to zero. The software would then have to handle it…or not.

NOTE

Be careful if you use error forcing and make sure you aren't creating a situation that can never happen in the real world. If the programmer checked that n was greater than zero at the top of the function and n was never used until the formula, setting it to zero and causing the software to fail would be an invalid test case.

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

Forcing Error Messages

A great way to use error forcing is to cause all the error messages in your software to appear. Most software uses internal error codes to represent each error message. When an internal error condition flag is set, the error handler takes the variable that holds the error code, looks up the code in a table, and displays the appropriate message.

Many errors are difficult to create—like hooking up 2,049 printers. But if all you want to do is test that the error messages are correct (spelling, language, formatting, and so on), using error forcing can be a very efficient way to see all of them. Keep in mind, though, that you aren't testing the code that detects the error, just the code that displays it.

Code Coverage

As with black-box testing, testing the data is only half the battle. For comprehensive coverage you must also test the program's states and the program's flow among them. You must attempt

to enter and exit every module, execute every line of code, and follow every logic and decision path through the software. Examining the software at this level of detail is called *code-coverage analysis*.

Code-coverage analysis is a dynamic white-box testing technique because it requires you to have full access to the code to view what parts of the software you pass through when you run your test cases.

The simplest form of code-coverage analysis is using your compiler's debugger to view the lines of code you visit as you single-step through the program. Figure 7.9 shows an example of the Visual Basic debugger in operation.

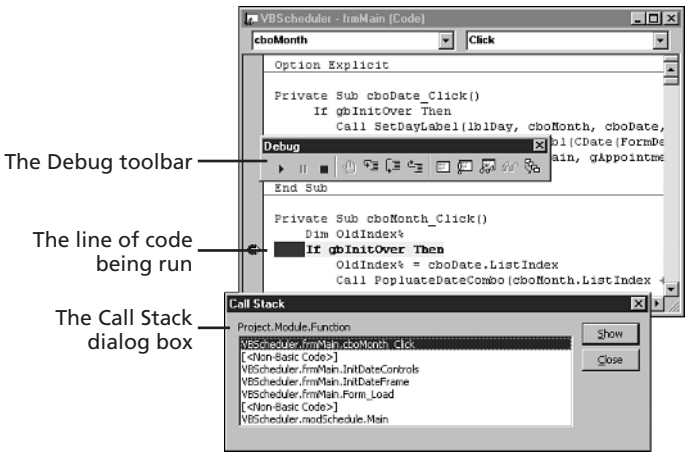


FIGURE 7.9

The debugger allows you to single-step through the software to see what lines of code and modules you execute while running your test cases.

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

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

- What parts of the software your test cases don't cover. If the code in a specific module is never executed, you know that you need to write additional test cases for testing that module's function.

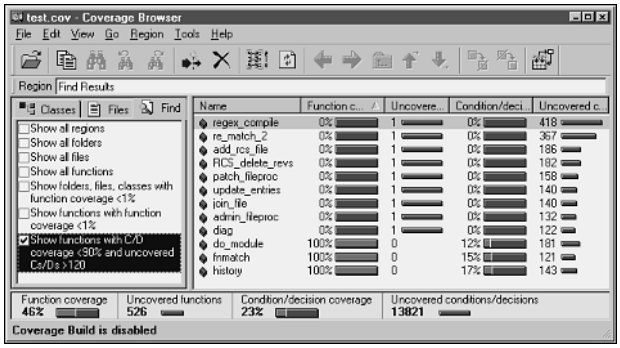


FIGURE 7.10

A code-coverage analyzer provides detailed information about how effective your test cases are. (This figure is copyright and courtesy of Bullseye Testing Technology.)

- Which test cases are redundant. If you run a series of test cases and they don't increase the percentage of code covered, they are likely in the same equivalence partition.
- What new test cases need to be created for better coverage. You can look at the code that has low coverage, see how it works and what it does, and create new test cases that will exercise it.

You will also have a general feel for the quality of the software. If your test cases cover 90 percent of the software and don't find any bugs, the software is in pretty good shape. If, on the other hand, your tests cover only 50 percent of the software and you're still finding bugs, you know you still have work to do.

Program Statement and Line Coverage

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

LISTING 7.1 It's Very Easy to Test Every Line of This Simple Program

```
PRINT "Hello World"
PRINT "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

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

Branch Coverage

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

LISTING 7.2 The `IF` Statement Creates Another Branch Through the Code

```
PRINT "Hello World"
IF Date$ = "01-01-2000" THEN
    PRINT "Happy New Year"
    END IF
PRING "The date is: "; Date$
PRINT "The time is: "; Time$
END
```

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

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

Your code coverage analyzer would state that you tested every statement and achieved 100 percent coverage. You could quit testing, right? Wrong! You may have tested every *statement*, but you didn't test every *branch*.

Your gut may be telling you that you still need to try a test case for a date that's not January 1, 2000. If you did, the program would execute the other path through the program:

Lines 1, 2, 5, 6, 7

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

Condition Coverage

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

LISTING 7.3 The Multiple Conditions in the `IF` Statement Create More Paths Through the Code

```
PRINT "Hello World"
IF Date$ = "01-01-2000" AND Time$ = "00:00:00" THEN
    PRINT "Happy New Year"
    END IF
PRINT "The date is: "; Date$
PRINT "The time is: "; Time$
END
```


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

TABLE 7.2 Test Cases to Achieve Full Coverage of the Multiple `IF` Statement Condition

Date$	*Time$*	*Line # Execution*
01-01-0000	11:11:11	1,2,5,6,7
01-01-0000	00:00:00	1,2,5,6,7
01-01-2000	11:11:11	1,2,5,6,7
01-01-2000	00:00:00	1,2,3,4,5,6,7

If you were concerned only with branch coverage, the first three conditions would be redundant and could be equivalence partitioned into a single test case. But, with condition coverage testing, all four cases are important because they exercise different conditions of the `IF` statement in line 4.

As with branch coverage, code coverage analyzers can be configured to consider conditions when reporting their results. If you test for all the possible conditions, you will achieve branch coverage and therefore achieve statement coverage.

NOTE

If you manage to test every statement, branch, and condition (and that's impossible except for the smallest of programs), you still haven't tested the program completely. Remember, all the data errors discussed in the first part of this chapter are still possible. The program flow and the data together make up the operation of the software.

Summary

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

Chapters 4 through 7 covered the fundamentals of software testing:

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

In a sense, this is all there is to software testing. Of course, reading about it in four chapters and putting it into practice are very different things. Being a good software tester requires lots of dedication and hard work. It takes practice and experience to know when and how to best apply these fundamental techniques.

In Part III, "Applying Your Testing Skills," you'll learn about different types of software testing and how you can apply the skills from your "black and white testing box" to real-world scenarios.

Quiz

These quiz questions are provided for your further understanding. See Appendix A, "Answers to Quiz Questions," for the answers—but don't peek!

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