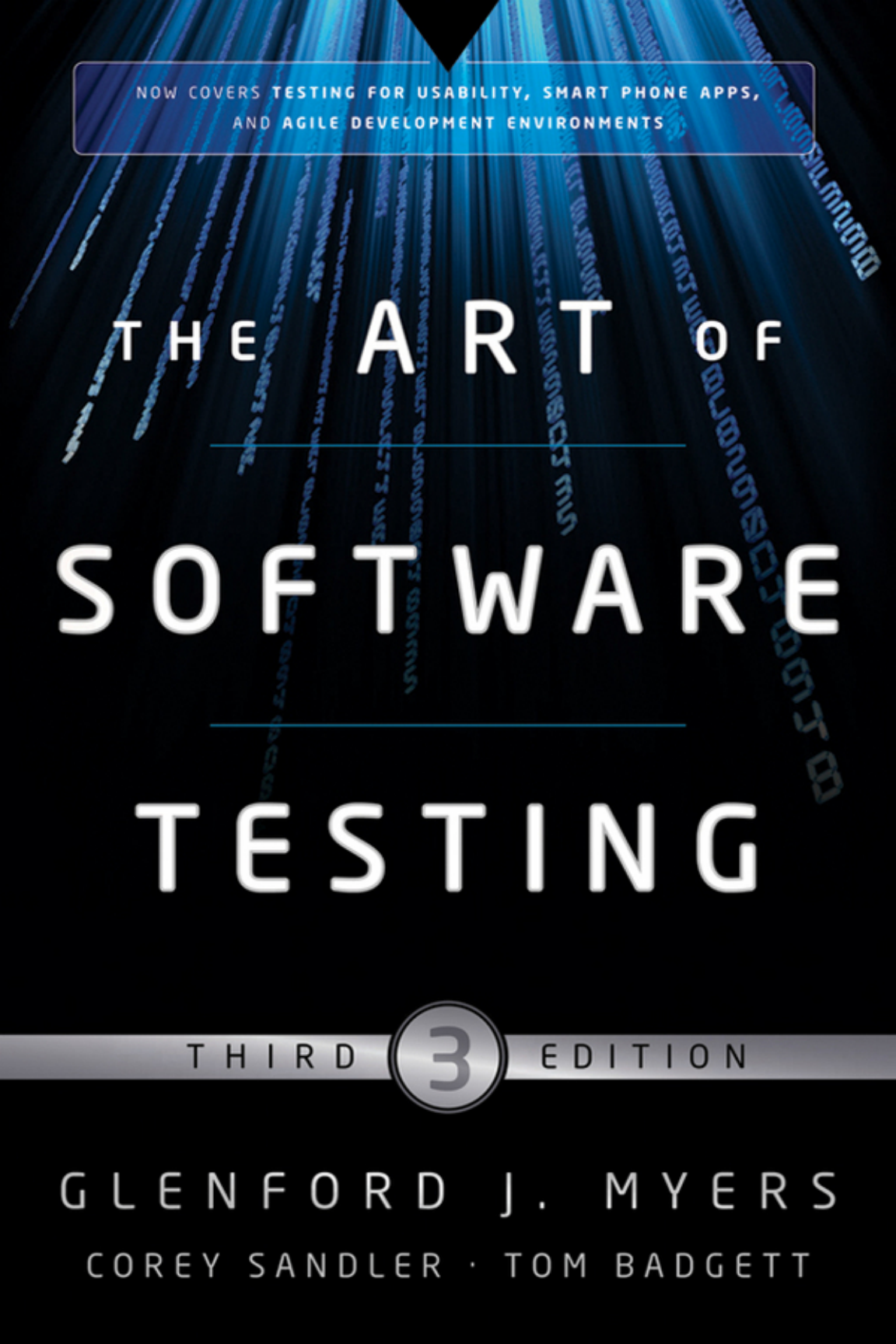
NOW COVERS TESTING FOR USABILITY, SMART PHONE APPS,
AND AGILE DEVELOPMENT ENVIRONMENTS
THE ART OF
SOFTWARE
TESTING
THIRD 3 EDITION
GLENFORD J. MYERS
COREY SANDLER · TOM BADGETT

THE ART OF SOFTWARE TESTING

THE ART OF SOFTWARE TESTING

Third Edition

GLENFORD J. MYERS
TOM BADGETT
COREY SANDLER

John Wiley & Sons, Inc.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Myers, Glenford J., 1946-
The art of software testing / Glenford J. Myers, Corey Sandler, Tom Badgett. — 3rd ed.
p. cm.
Includes index.

ISBN 978-1-118-03196-4 (cloth); ISBN 978-1-118-13313-2 (ebk); ISBN 978-1-118-13314-9 (ebk); ISBN 978-1-118-13315-6 (ebk)

1. Computer software—Testing. 2. Debugging in computer science. I. Sandler, Corey, 1950- II. Badgett, Tom. III. Title.
QA76.76.T48M894 2011
005.1'4—dc23

2011017548

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

Preface

In 1979, Glenford Myers published a book that turned out to be a classic. *The Art of Software Testing* has stood the test of time—25 years on the publisher's list of available books. This fact alone is a testament to the solid, essential, and valuable nature of his work.

During that same time, the authors of this edition (the third) of *The Art of Software Testing* published, collectively, more than 200 books, most of them on computer software topics. Some of these titles sold very well and, like this one, have gone through multiple versions. Corey Sandler's *Fix Your Own PC*, for example, is in its eighth edition as this book goes to press; and Tom Badgett's books on Microsoft PowerPoint and other Office titles have gone through four or more editions. However, unlike Myers's book, none of these remained current for more than a few years.

What is the difference? The newer books covered more transient topics—operating systems, applications software, security, communications technology, and hardware configurations. Rapid changes in computer hardware and software technology during the 1980s and 1990s necessitated frequent changes and updates to these topics.

Also during that period hundreds of books about software testing were published. They, too, took a more transient approach to the topic. *The Art of Software Testing* alone gave the industry a long-lasting, foundational guide to one of the most important computer topics: How do you ensure that all of the software you produce does what it was designed to do, and—just as important—doesn't do what it isn't supposed to do?

The edition you are reading today retains the foundational philosophy laid by Myers more than three decades ago. But we have updated the examples to include more current programming languages, and we have addressed topics that were not yet topics when Myers wrote the first edition: Web programming, e-commerce, Extreme (Agile) programming and testing, and testing applications for mobile devices.

Along the way, we never lost sight of the fact that a new classic must stay true to its roots, so our version also offers you a software testing philosophy, and a process that works across current and unforeseeable future hardware and software platforms. We hope that the third edition of *The Art of Software Testing*, too, will span a generation of software designers and developers.

Introduction

At the time this book was first published, in 1979, it was a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed.

Today, a third of a century and two book updates later, the same holds true. There are new development systems, languages with built-in tools, and programmers who are used to developing more on the fly. But testing continues to play an important part in any software development project.

Given these facts, you might expect that by this time program testing would have been refined into an exact science. This is far from the case. In fact, less seems to be known about software testing than about any other aspect of software development. Furthermore, testing has been an out-of-vogue subject; it was so when this book was first published and, unfortunately, this has not changed. Today there *are* more books and articles about software testing—meaning that, at least, the topic has greater visibility than it did when this book was first published—but testing remains among the "dark arts" of software development.

This would be more than enough reason to update this book on the art of software testing, but we have additional motivations. At various times, we have heard professors and teaching assistants say, "Our students graduate and move into industry without any substantial knowledge of how to go about testing a program. Moreover, we rarely have any advice to offer in our introductory courses on how a student should go about testing and debugging his or her exercises."

Thus, the purpose of this updated edition of *The Art of Software Testing* is the same as it was in 1979 and in 2004: to fill these knowledge gaps for the professional programmer and the student of computer science. As the title implies, the book is a practical, rather than theoretical, discussion of the subject, complete with updated language and process discussions.

Although it is possible to discuss program testing in a theoretical vein, this book is intended to be a practical, "both feet on the ground" handbook. Hence, many subjects related to program testing, such as the idea of mathematically proving the correctness of a program, were purposefully excluded.

Chapter 1 "assigns" a short self-assessment test that every reader should take before reading further. It turns out that the most important practical information you must understand about program testing is a set of philosophical and economic issues; these are discussed in Chapter 2. Chapter 3 introduces the important concept of noncomputer-based code walkthroughs, or inspections. Rather than focus attention on the procedural or managerial aspects of this concept, as most such discussions do, this chapter addresses it from a technical, how-to-find-errors point of view.

The alert reader will realize that the most important component in a program tester's bag of tricks is the knowledge of how to write effective test cases; this is the subject of Chapter 3. Chapter 4 discusses the testing of individual modules or subroutines, followed in Chapter 5 by the testing of larger entities. Chapter 6 takes on the concept of user or usability testing, a component of software testing that always has been important, but is even more relevant today due to the advent of more complex software targeted at an ever broadening audience. Chapter 7 offers some practical advice on program debugging, while Chapter 8 delves into the concepts of extreme programming testing with emphasis on what has come to be called the "agile environment." Chapter 9 shows how to use other features of program testing, which are detailed elsewhere in this book, with Web programming, including e-commerce systems, and the all new, highly interactive social networking sites. Chapter 10 describes how to test software developed for the mobile environment.

We direct this book at three major audiences. First, the professional programmer. Although we hope that not everything in this book will be new information to this audience, we believe it will add to the professional's knowledge of testing techniques. If the material allows this group to detect just one more bug in one program, the price of the book will have been recovered many times over.

The second audience is the project manager, who will benefit from the book's practical information on the management of the testing process. The third audience is the programming and computer science student, and our goal for them is twofold: to expose them to the problems of

program testing, and provide a set of effective techniques. For this third group, we suggest the book be used as a supplement in programming courses such that students are exposed to the subject of software testing early in their education.

1 A Self-Assessment Test

Since this book was first published over 30 years ago, software testing has become more difficult and easier than ever.

Software testing is more difficult because of the vast array of programming languages, operating systems, and hardware platforms that have evolved in the intervening decades. And while relatively few people used computers in the 1970s, today virtually no one can complete a day's work without using a computer. Not only do computers exist on your desk, but a "computer," and consequently software, is present in almost every device we use. Just try to think of the devices today that society relies on that are *not* software driven. Sure there are some—hammers and wheelbarrows come to mind—but the vast majority use some form of software to operate. Software is pervasive, which raises the value of testing it. The machines themselves are hundreds of times more powerful, and smaller, than those early devices, and today's concept of "computer" is much broader and more difficult to define. Televisions, telephones, gaming systems, and automobiles all contain computers and computer software, and in some cases can even be considered computers themselves.

Therefore, the software we write today potentially touches millions of people, either enabling them to do their jobs effectively and efficiently, or causing them untold frustration and costing them in the form of lost work or lost business. This is not to say that software is more important today than it was when the first edition of this book was published, but it is safe to say that computers—and the software that drives them—certainly affect more people and more businesses now than ever before.

Software testing is easier, too, in some ways, because the array of software and operating systems is much more sophisticated than in the past, providing intrinsic, well-tested routines that can be incorporated into applications without the need for a programmer to develop them from scratch. Graphical User Interfaces (GUIs), for example, can be built from a development language's libraries, and since they are preprogrammed objects that have been debugged and tested previously, the need for testing them as part of a custom application is much reduced.

And, despite the plethora of software testing tomes available on the market today, many developers seem to have an attitude that is counter to extensive testing. Better development tools, pretested GUIs, and the pressure of tight deadlines in an ever more complex development environment can lead to avoidance of all but the most obvious testing protocols. Whereas low-level impacts of bugs may only inconvenience the end user, the worst impacts can result in large financial loses, or even cause harm to people. The procedures in this book can help designers, developers, and project managers better understand the value of comprehensive testing, and provide guidelines to help them achieve required testing goals.

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended. Software should be predictable and consistent, presenting no surprises to users. In this book, we will look at many approaches to achieving this goal.

Now, before we start the book, we'd like you to take a short exam. We want you to write a set of test cases—specific sets of data—to test properly a relatively simple program. Create a set of test data for the program—data the program must handle correctly to be considered a successful program. Here's a description of the program:

> **The program reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.**

Remember that a scalene triangle is one where no two sides are equal, whereas an isosceles triangle has two equal sides, and an equilateral triangle has three sides of equal length. Moreover, the angles opposite the

equal sides in an isosceles triangle also are equal (it also follows that the sides opposite equal angles in a triangle are equal), and all angles in an equilateral triangle are equal.

Evaluate your set of test cases by using it to answer the following questions. Give yourself one point for each yes answer.

1. Do you have a test case that represents a *valid* scalene triangle? (Note that test cases such as 1, 2, 3 and 2, 5, 10 do not warrant a yes answer because a triangle having these dimensions is not valid.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (Note that a test case representing 2, 2, 4 would not count because it is not a valid triangle.)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (such as, 3, 3, 4; 3, 4, 3; and 4, 3, 3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said that 1, 2, 3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (e.g., 1, 2, 3; 1, 3, 2; and 3, 1, 2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (such as 1, 2, 4 or 12, 15, 30)?
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (e.g., 1, 2, 4; 1, 4, 2; and 4, 1, 2)?
11. Do you have a test case in which all sides are zero (0, 0, 0)?
12. Do you have at least one test case specifying noninteger values (such as 2.5, 3.5, 5.5)?
13. Do you have at least one test case specifying the wrong number of values (two rather than three integers, for example)?
14. For each test case did you specify the expected output from the program in addition to the input values?

Of course, a set of test cases that satisfies these conditions does not guarantee that you will find all possible errors, but since questions 1 through 13 represent errors that actually have occurred in different versions of this program, an adequate test of this program should expose at least these errors.

Now, before you become concerned about your score, consider this: In our experience, highly qualified professional programmers score, on the average, only 7.8 out of a possible 14. If you've done better, congratulations; if not, we're here to help.

The point of the exercise is to illustrate that the testing of even a trivial program such as this is not an easy task. Given this is true, consider the difficulty of testing a 100,000-statement air traffic control system, a compiler, or even a mundane payroll program. Testing also becomes more difficult with the object-oriented languages, such as Java and C++. For example, your test cases for applications built with these languages must expose errors associated with object instantiation and memory management.

It might seem from working with this example that thoroughly testing a complex, real-world program would be impossible. Not so! Although the task can be daunting, adequate program testing is a very necessary—and achievable—part of software development, as you will learn in this book.

2 The Psychology and Economics of Software Testing

Software testing is a technical task, yes, but it also involves some important considerations of economics and human psychology.

In an ideal world, we would want to test every possible permutation of a program. In most cases, however, this simply is not possible. Even a seemingly simple program can have hundreds or thousands of possible input and output combinations. Creating test cases for all of these possibilities is impractical. Complete testing of a complex application would take too long and require too many human resources to be economically feasible.

In addition, the software tester needs the proper attitude (perhaps "vision" is a better word) to successfully test a software application. In some cases, the tester's attitude may be more important than the actual process itself. Therefore, we will start our discussion of software testing with these issues before we delve into the more technical nature of the topic.

The Psychology of Testing

One of the primary causes of poor application testing is the fact that most programmers begin with a false definition of the term. They might say:

"Testing is the process of demonstrating that errors are not present."
"The purpose of testing is to show that a program performs its intended functions correctly."
"Testing is the process of establishing confidence that a program does what it is supposed to do."

These definitions are upside down.

When you test a program, you want to add some value to it. Adding value through testing means raising the quality or reliability of the program. Raising the reliability of the program means finding and removing errors.

Therefore, don't test a program to show that it works; rather, start with the assumption that the program contains errors (a valid assumption for almost any program) and then test the program to find as many of the errors as possible.

Thus, a more appropriate definition is this:

Testing is the process of executing a program with the intent of finding errors.

Although this may sound like a game of subtle semantics, it's really an important distinction. Understanding the true definition of software testing can make a profound difference in the success of your efforts.

Human beings tend to be highly goal-oriented, and establishing the proper goal has an important psychological effect on them. If our goal is to demonstrate that a program has no errors, then we will be steered subconsciously toward this goal; that is, we tend to select test data that have a low probability of causing the program to fail. On the other hand, if our goal is to demonstrate that a program has errors, our test data will have a higher probability of finding errors. The latter approach will add more value to the program than the former.

This definition of testing has myriad implications, many of which are scattered throughout this book. For instance, it implies that testing is a destructive, even sadistic, process, which explains why most people find it difficult. That may go against our grain; with good fortune, most of us have a constructive, rather than a destructive, outlook on life. Most people are inclined toward making objects rather than ripping them apart. The definition also has implications for how test cases (test data) should be designed, and who should and who should not test a given program.

Another way of reinforcing the proper definition of testing is to analyze the use of the words "successful" and "unsuccessful"—in particular, their use by project managers in categorizing the results of test cases. Most project managers refer to a test case that did not find an error a "successful test run," whereas a test that discovers a new error is usually called "unsuccessful."

Once again, this is upside down. "Unsuccessful" denotes something undesirable or disappointing. To our way of thinking, a well-constructed and

executed software test is successful when it finds errors that can be fixed. That same test is also successful when it eventually establishes that there are no more errors to be found. The only unsuccessful test is one that does not properly examine the software; and, in the majority of cases, a test that found no errors likely would be considered unsuccessful, since the concept of a program without errors is basically unrealistic.

A test case that finds a new error can hardly be considered unsuccessful; rather, it has proven to be a valuable investment. An unsuccessful test case is one that causes a program to produce the correct result without finding any errors.

Consider the analogy of a person visiting a doctor because of an overall feeling of malaise. If the doctor runs some laboratory tests that do not locate the problem, we do not call the laboratory tests "successful"; they were unsuccessful tests in that the patient's net worth has been reduced by the expensive laboratory fees, the patient is still ill, and the patient may question the doctor's ability as a diagnostician. However, if a laboratory test determines that the patient has a peptic ulcer, the test is successful because the doctor can now begin the appropriate treatment. Hence, the medical profession seems to use these words in the proper sense. The analogy, of course, is that we should think of the program, as we begin testing it, as the sick patient.

A second problem with such definitions as "testing is the process of demonstrating that errors are not present" is that such a goal is impossible to achieve for virtually all programs, even trivial programs.

Again, psychological studies tell us that people perform poorly when they set out on a task that they know to be infeasible or impossible. For instance, if you were instructed to solve the crossword puzzle in the Sunday *New York Times* in 15 minutes, you probably would achieve little, if any, progress after 10 minutes because, if you are like most people, you would be resigned to the fact that the task seems impossible. If you were asked for a solution in four hours, however, we could reasonably expect to see more progress in the initial 10 minutes. Defining program testing as the process of uncovering errors in a program makes it a feasible task, thus overcoming this psychological problem.

A third problem with the common definitions such as "testing is the process of demonstrating that a program does what it is supposed to do" is that programs that do what they are supposed to do still can contain errors. That is, an error is clearly present if a program *does not do what it is supposed to do*; but errors are also present if a program *does what it is not supposed to do*. Consider the triangle program of Chapter 1. Even if we

could demonstrate that the program correctly distinguishes among all scalene, isosceles, and equilateral triangles, the program still would be in error if it does something it is not supposed to do (such as representing 1, 2, 3 as a scalene triangle or saying that 0, 0, 0 represents an equilateral triangle). We are more likely to discover the latter class of errors if we view program testing as the process of finding errors than if we view it as the process of showing that a program does what it is supposed to do.

To summarize, program testing is more properly viewed as the destructive process of trying to find the errors in a program (whose presence is assumed). A successful test case is one that furthers progress in this direction by causing the program to fail. Of course, you eventually want to use program testing to establish some degree of confidence that a program does what it is supposed to do and does not do what it is not supposed to do, but this purpose is best achieved by a diligent exploration for errors.

Consider someone approaching you with the claim that "my program is perfect" (i.e., error free). The best way to establish some confidence in this claim is to try to refute it, that is, to try to find imperfections rather than just confirm that the program works correctly for some set of input data.

The Economics of Testing

Given our definition of program testing, an appropriate next step is to determine whether it is possible to test a program to find *all* of its errors. We will show you that the answer is negative, even for trivial programs. In general, it is impractical, often impossible, to find all the errors in a program. This fundamental problem will, in turn, have implications for the economics of testing, assumptions that the tester will have to make about the program, and the manner in which test cases are designed.

To combat the challenges associated with testing economics, you should establish some strategies before beginning. Two of the most prevalent strategies include black-box testing and white-box testing, which we will explore in the next two sections.

Black-Box Testing

One important testing strategy is *black-box testing (also known as data-driven* or *input/output-driven* testing). To use this method, view the program as a black box. Your goal is to be completely unconcerned about the

internal behavior and structure of the program. Instead, concentrate on finding circumstances in which the program does not behave according to its specifications.

In this approach, test data are derived solely from the specifications (i.e., without taking advantage of knowledge of the internal structure of the program).

If you want to use this approach to find all errors in the program, the criterion is *exhaustive input testing*, making use of every possible input condition as a test case. Why? If you tried three equilateral-triangle test cases for the triangle program, that in no way guarantees the correct detection of all equilateral triangles. The program could contain a special check for values 3842, 3842, 3842 and denote such a triangle as a scalene triangle. Since the program is a black box, the only way to be sure of detecting the presence of such a statement is by trying every input condition.

To test the triangle program exhaustively, you would have to create test cases for all valid triangles up to the maximum integer size of the development language. This in itself is an astronomical number of test cases, but it is in no way exhaustive: It would not find errors where the program said that −3, 4, 5 is a scalene triangle and that 2, A, 2 is an isosceles triangle. To be sure of finding all such errors, you have to test using not only all *valid* inputs, but all *possible* inputs. Hence, to test the triangle program exhaustively, you would have to produce virtually an infinite number of test cases, which, of course, is not possible.

If this sounds difficult, exhaustive input testing of larger programs is even more problematic. Consider attempting an exhaustive black-box test of a C++ compiler. Not only would you have to create test cases representing all valid C++ programs (again, virtually an infinite number), but you would have to create test cases for all invalid C++ programs (an infinite number) to ensure that the compiler detects them as being invalid. That is, the compiler has to be tested to ensure that it does not do what it is not supposed to do—for example, successfully compile a syntactically incorrect program.

The problem is even more onerous for transaction-base programs such as database applications. For example, in a database application such as an airline reservation system, the execution of a transaction (such as a database query or a reservation for a plane flight) is dependent upon what happened in previous transactions. Hence, not only would you have to try all unique valid and invalid transactions, but also all possible sequences of transactions.

This discussion shows that exhaustive input testing is impossible. Two important implications of this: (1) You cannot test a program to guarantee that it is error free; and (2) a fundamental consideration in program testing is one of economics. Thus, since exhaustive testing is out of the question, the objective should be to maximize the yield on the testing investment by maximizing the number of errors found by a finite number of test cases. Doing so will involve, among other things, being able to peer inside the program and make certain reasonable, but not airtight, assumptions about the program (e.g., if the triangle program detects 2, 2, 2 as an equilateral triangle, it seems reasonable that it will do the same for 3, 3, 3). This will form part of the test case design strategy in Chapter 4.

White-Box Testing

Another testing strategy, *white-box* (or *logic-driven*) testing, permits you to examine the internal structure of the program. This strategy derives test data from an examination of the program's logic (and often, unfortunately, at the neglect of the specification).

The goal at this point is to establish for this strategy the analog to exhaustive input testing in the black-box approach. Causing every statement in the program to execute at least once might appear to be the answer, but it is not difficult to show that this is highly inadequate. Without belaboring the point here, since this matter is discussed in greater depth in Chapter 4, the analog is usually considered to be *exhaustive path testing*. That is, if you execute, via test cases, all possible paths of control flow through the program, then possibly the program has been completely tested.

There are two flaws in this statement, however. One is that the number of unique logic paths through a program could be astronomically large. To see this, consider the trivial program represented in Figure 2.1. The diagram is a control-flow graph. Each node or circle represents a segment of statements that execute sequentially, possibly terminating with a branching statement. Each edge or arc represents a transfer of control (branch) between segments. The diagram, then, depicts a 10- to 20-statement program consisting of a `DO` loop that iterates up to 20 times. Within the body of the `DO` loop is a set of nested `IF` statements. Determining the number of unique logic paths is the same as determining the total number of unique ways of moving from point *a* to point *b* (assuming that all decisions in the program are independent from one another). This number is approximately 10^{14}, or

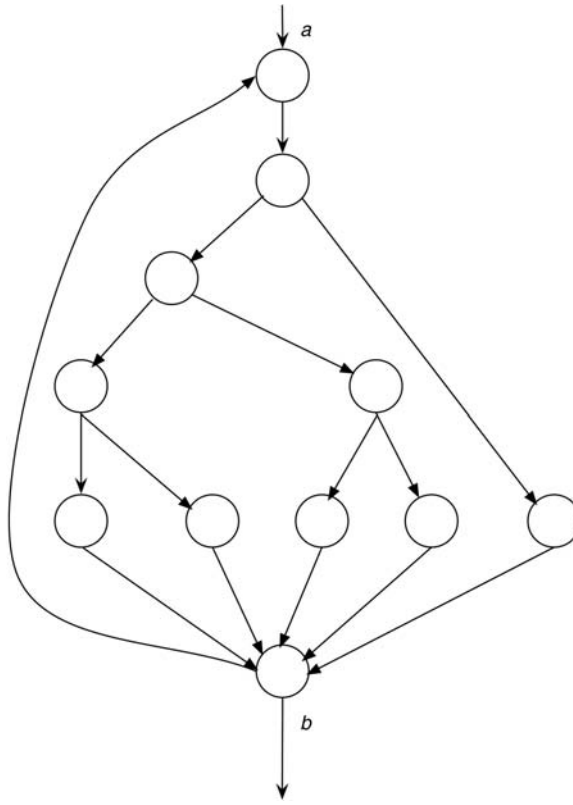

FIGURE 2.1 Control-Flow Graph of a Small Program.

100 trillion. It is computed from $5^{20} + 5^{19} + \ldots 5^{1}$, where 5 is the number of paths through the loop body. Most people have a difficult time visualizing such a number, so consider it this way: If you could write, execute, and verify a test case every five minutes, it would take approximately 1 billion years to try every path. If you were 300 times faster, completing a test once per second, you could complete the job in 3.2 million years, give or take a few leap years and centuries.

Of course, in actual programs every decision is not independent from every other decision, meaning that the number of possible execution paths would be somewhat fewer. On the other hand, actual programs are much larger than the simple program depicted in Figure 2.1. Hence, exhaustive path testing, like exhaustive input testing, appears to be impractical, if not impossible.

The second flaw in the statement "exhaustive path testing means a complete test" is that every path in a program could be tested, yet the program might still be loaded with errors. There are three explanations for this.

The first is that an exhaustive path test in no way guarantees that a program matches its specification. For example, if you were asked to write an ascending-order sorting routine but mistakenly produced a descending-order sorting routine, exhaustive path testing would be of little value; the program still has one bug: It is the wrong program, as it does not meet the specification.

Second, a program may be incorrect because of *missing paths*. Exhaustive path testing, of course, would not detect the absence of necessary paths.

Third, an exhaustive path test might not uncover *data-sensitivity* errors. There are many examples of such errors, but a simple one should suffice. Suppose that in a program you have to compare two numbers for convergence, that is, to see if the difference between the two numbers is less than some predetermined value. For example, you might write a Java *IF* statement as

```
if (a-b<c)
   System.out.println("a-b<c");
```

Of course, the statement contains an error because it should compare `c` to the absolute value of `a-b`. Detection of this error, however, is dependent upon the values used for `a` and `b` and would not necessarily be detected by just executing every path through the program.

In conclusion, although exhaustive input testing is superior to exhaustive path testing, neither proves to be useful because both are infeasible. Perhaps, then, there are ways of combining elements of black-box and white-box testing to derive a reasonable, but not airtight, testing strategy. This matter is pursued further in Chapter 4.

Software Testing Principles

Continuing with the major premise of this chapter, that the most important considerations in software testing are issues of psychology, we can identify a set of vital testing principles or guidelines. Most of these principles may seem obvious, yet they are all too often overlooked. Table 2.1 summarizes these important principles, and each is discussed in more detail in the paragraphs that follow.

TABLE 2.1 Vital Program Testing Guidelines

Principle Number	Principle
1	A necessary part of a test case is a definition of the expected output or result.
2	A programmer should avoid attempting to test his or her own program.
3	A programming organization should not test its own programs.
4	Any testing process should include a thorough inspection of the results of each test.
5	Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6	Examining a program to see if it *does not do what it is supposed to do* is only half the battle; the other half is seeing whether the program *does what it is not supposed to do*.
7	Avoid throwaway test cases unless the program is truly a throwaway program.
8	Do not plan a testing effort under the tacit assumption that no errors will be found.
9	The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10	Testing is an extremely creative and intellectually challenging task.

Principle 1: A necessary part of a test case is a definition of the expected output or result.

This principle, though obvious, when overlooked is the cause of one of the most frequent mistakes in program testing. Again, it is something that is based on human psychology. If the expected result of a test case has not been predefined, chances are that a plausible, but erroneous, result will be interpreted as a correct result because of the phenomenon of "the eye seeing what it wants to see." In other words, in spite of the proper destructive definition of testing, there is still a subconscious desire to see the correct result. One way of

combating this is to encourage a detailed examination of all output by precisely spelling out, in advance, the expected output of the program. Therefore, a test case must consist of two components:

1. A description of the input data to the program.
2. A precise description of the correct output of the program for that set of input data.

A problem may be characterized as a fact or group of facts for which we have no acceptable explanation, that seem unusual, or that fail to fit in with our expectations or preconceptions. It should be obvious that some prior beliefs are required if anything is to appear problematic. If there are no expectations, there can be no surprises.

Principle 2: A programmer should avoid attempting to test his or her own program.

Any writer knows—or should know—that it's a bad idea to attempt to edit or proofread his or her own work. They know what the piece is *supposed* to say, hence may not recognize when it says otherwise. And they really don't want to find errors in their own work. The same applies to software authors.

Another problem arises with a change in focus on a software project. After a programmer has *constructively* designed and coded a program, it is extremely difficult to suddenly change perspective to look at the program with a *destructive* eye.

As many homeowners know, removing wallpaper (a destructive process) is not easy, but it is almost unbearably depressing if it was your hands that hung the paper in the first place. Similarly, most programmers cannot effectively test their own programs because they cannot bring themselves to shift mental gears to attempt to expose errors. Furthermore, a programmer may subconsciously avoid finding errors for fear of retribution from peers or a supervisor, a client, or the owner of the program or system being developed.

In addition to these psychological issues, there is a second significant problem: The program may contain errors due to the programmer's misunderstanding of the problem statement or specification. If this is the case, it is likely that the programmer will carry the same misunderstanding into tests of his or her own program.

This does not mean that it is impossible for a programmer to test his or her own program. Rather, it implies that testing is more effective and successful if someone else does it. However, as we will

discuss in more detail in Chapter 3, developers can be valuable members of the testing team when the program specification and the program code itself are being evaluated.

Note that this argument does not apply to debugging (correcting known errors); debugging is more efficiently performed by the original programmer.

Principle 3: A programming organization should not test its own programs.

The argument here is similar to that made in the previous principle. A project or programming organization is, in many senses, a living organization with psychological problems similar to those of individual programmers. Furthermore, in most environments, a programming organization or a project manager is largely measured on the ability to produce a program by a given date and for a certain cost. One reason for this is that it is easy to measure time and cost objectives, whereas it is extremely difficult to quantify the reliability of a program. Therefore, it is difficult for a programming organization to be objective in testing its own programs, because the testing process, if approached with the proper definition, may be viewed as decreasing the probability of meeting the schedule and the cost objectives.

Again, this does not say that it is impossible for a programming organization to find some of its errors, because organizations do accomplish this with some degree of success. Rather, it implies that it is more economical for testing to be performed by an objective, independent party.

Principle 4: Any testing process should include a thorough inspection of the results of each test.

This is probably the most obvious principle, but again it is something that is often overlooked. We've seen numerous experiments that show many subjects failed to detect certain errors, even when symptoms of those errors were clearly observable on the output listings. Put another way, errors that are found in later tests were often missed in the results from earlier tests.

Principle 5: Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.

There is a natural tendency when testing a program to concentrate on the valid and expected input conditions, to the neglect of the

invalid and unexpected conditions. For instance, this tendency frequently appears in the testing of the triangle program in Chapter 1.

Few people, for instance, feed the program the numbers 1, 2, 5 to ensure that the program does not erroneously interpret this as an equalateral triangle instead of a scalene triangle. Also, many errors that are suddenly discovered in production software turn up when it is used in some new or unexpected way. It is hard, if not impossible, to define all the use cases for software testing. Therefore, test cases representing unexpected and invalid input conditions seem to have a higher error-detection yield than do test cases for valid input conditions.

Principle 6: Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.

This is a corollary to the previous principle. Programs must be examined for unwanted side effects. For instance, a payroll program that produces the correct paychecks is still an erroneous program if it also produces extra checks for nonexistent employees, or if it overwrites the first record of the personnel file.

Principle 7: Avoid throwaway test cases unless the program is truly a throwaway program.

This problem is seen most often with interactive systems to test programs. A common practice is to sit at a terminal and invent test cases on the fly, and then send these test cases through the program. The major issue is that test cases represent a valuable investment that, in this environment, disappears after the testing has been completed. Whenever the program has to be tested again (e.g., after correcting an error or making an improvement), the test cases must be reinvented. More often than not, since this reinvention requires a considerable amount of work, people tend to avoid it. Therefore, the retest of the program is rarely as rigorous as the original test, meaning that if the modification causes a previously functional part of the program to fail, this error often goes undetected. Saving test cases and running them again after changes to other components of the program is known as *regression testing*.

Principle 8: Do not plan a testing effort under the tacit assumption that no errors will be found.

This is a mistake project managers often make and is a sign of the use of the incorrect definition of testing—that is, the assumption that

testing is the process of showing that the program functions correctly. Once again, the definition of testing is the *process of executing a program with the intent of finding errors*. And it should be obvious from our previous discussions that it is impossible to develop a program that is completely error free. Even after extensive testing and error correction, it is safe to assume that errors still exist; they simply have not yet been found.

Principle 9: The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.

This phenomenon is illustrated in Figure 2.2. At first glance this concept may seem nonsensical, but it is a phenomenon present in many programs. For instance, if a program consists of two modules, classes, or subroutines, *A* and *B*, and five errors have been found in module *A*, and only one error has been found in module *B*, and if module *A* has not been purposely subjected to a more rigorous test, then this principle tells us that the likelihood of more errors in module *A* is greater than the likelihood of more errors in module *B*.

Another way of stating this principle is to say that errors tend to come in clusters and that, in the typical program, some sections seem to be much more prone to errors than other sections, although nobody has supplied a good explanation of why this occurs. The phenomenon is useful in that it gives us insight or feedback in the testing process. If a particular section of a program seems to be much more prone to errors than other sections, then this phenomenon tells us

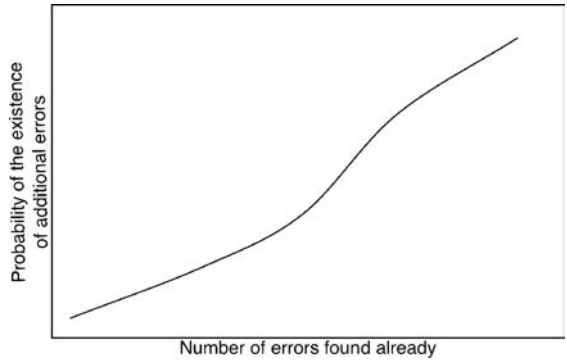

FIGURE 2.2 The Surprising Relationship between Errors Remaining and Errors Found.

that, in terms of yield on our testing investment, additional testing efforts are best focused against this error-prone section.

Principle 10: Testing is an extremely creative and intellectually challenging task.

It is probably true that the creativity required in testing a large program exceeds the creativity required in designing that program. We already have seen that it is impossible to test a program sufficiently to guarantee the absence of all errors. Methodologies discussed later in this book help you develop a reasonable set of test cases for a program, but these methodologies still require a significant amount of creativity.

Summary

As you proceed through this book, keep in mind these important principles of testing:

- Testing is the process of executing a program with the intent of finding errors.
- Testing is more successful when not performed by the developer(s).
- A good test case is one that has a high probability of detecting an undiscovered error.
- A successful test case is one that detects an undiscovered error.
- Successful testing includes carefully defining expected output as well as input.
- Successful testing includes carefully studying test results.

3 Program Inspections, Walkthroughs, and Reviews

For many years, most of us in the programming community worked under the assumptions that programs are written solely for machine execution, and are not intended for people to read, and that the only way to test a program is to execute it on a machine. This attitude began to change in the early 1970s through the efforts of program developers who first saw the value in reading code as part of a comprehensive testing and debugging regimen.

Today, not all testers of software applications read code, but the concept of studying program code as part of a testing effort certainly is widely accepted. Several factors may affect the likelihood that a given testing and debugging effort will include people actually reading program code: the size or complexity of the application, the size of the development team, the timeline for application development (whether the schedule is relaxed or intense, for example), and, of course, the background and culture of the programming team.

For these reasons, we will discuss the process of noncomputer-based testing ("human testing") before we delve into the more traditional computer-based testing techniques. Human testing techniques are quite effective in finding errors—so much so that every programming project should use one or more of these techniques. You should apply these methods between the time the program is coded and when computer-based testing begins. You also can develop and apply analogous methods

at earlier stages in the programming process (such as at the end of each design stage), but these are outside the scope of this book.

Before we begin the discussion of human testing techniques, take note of this important point: Because the involvement of humans results in less formal methods than mathematical proofs conducted by a computer, you may feel skeptical that something so simple and informal can be useful. Just the opposite is true. These informal techniques don't get in the way of successful testing; rather, they contribute substantially to productivity and reliability in two major ways.

First, it is generally recognized that the earlier errors are found, the lower the costs of correcting the errors and the higher the probability of correcting them correctly. Second, programmers seem to experience a psychological shift when computer-based testing commences. Internally induced pressures seem to build rapidly and there is a tendency to want to "fix this darn bug as soon as possible." Because of these pressures, programmers tend to make more mistakes when correcting an error found *during* computer-based testing than they make when correcting an error found earlier.

Inspections and Walkthroughs

The three primary human testing methods are *code inspections*, *walkthroughs* and *user* (or *usability*) *testing*. We cover the first two of these, which are code-oriented methods, in this chapter. These methods can be used at virtually any stage of software development, after an application is deemed to be complete or as each module or unit is complete (see Chapter 5 for more information on module testing). We discuss user testing in detail in Chapter 7.

The two code inspection methods have a lot in common, so we will discuss their similarities together. Their differences are enumerated in subsequent sections.

Inspections and walkthroughs involve a team of people reading or visually inspecting a program. With either method, participants must conduct some preparatory work. The climax is a "meeting of the minds," at a participant conference. The objective of the meeting is to find errors but not to find solutions to the errors—that is, to test, not debug.

Code inspections and walkthroughs have been widely used for some time. In our opinion, the reason for their success is related to some of the principles identified in Chapter 2.

In a walkthrough, a group of developers—with three or four being an optimal number—performs the review. Only one of the participants is the author of the program. Therefore, the majority of program testing is conducted by people other than the author, which follows testing principle 2, which states that an individual is usually ineffective in testing his or her own program. (Refer to Chapter 2, Table 2.1, and the subsequent discussion for all 10 program testing principles.)

An inspection or walkthrough is an improvement over the older desk-checking process (whereby a programmer reads his or her own program before testing it). Inspections and walkthroughs are more effective, again because people other than the program's author are involved in the process.

Another advantage of walkthroughs, resulting in lower debugging (error-correction) costs, is the fact that when an error is found it usually is located precisely in the code as opposed to black box testing where you only receive an unexpected result. Moreover, this process frequently exposes a batch of errors, allowing the errors to be corrected later en masse. Computer-based testing, on the other hand, normally exposes only a symptom of the error (e.g., the program does not terminate or the program prints a meaningless result), and errors are usually detected and corrected one by one.

These human testing methods generally are effective in finding from 30 to 70 percent of the logic-design and coding errors in typical programs. They are not effective, however, in detecting high-level design errors, such as errors made in the requirements analysis process. Note that a success rate of 30 to 70 percent doesn't mean that up to 70 percent of all errors might be found. Recall from Chapter 2 that we can never know the total number of errors in a program. Thus, what this means is that these methods are effective in finding up to 70 percent of all errors found by the end of the testing process.

Of course, a possible criticism of these statistics is that the human processes find only the "easy" errors (those that would be trivial to find with computer-based testing) and that the difficult, obscure, or tricky errors can be found only by computer-based testing. However, some testers using these techniques have found that the human processes tend to be *more* effective than the computer-based testing processes in finding certain types of errors, while the opposite is true for other types of errors (e.g., uninitialized variables versus divide by zero errors).

The implication is that inspections/walkthroughs and computer-based testing are complementary; error-detection efficiency will suffer if one or the other is not present.

Finally, although these processes are invaluable for testing new programs, they are of equal, or even higher, value in testing modifications to programs. In our experience, modifying an existing program is a process that is more error prone (in terms of errors per statement written) than writing a new program. Therefore, program modifications also should be subjected to these testing processes as well as regression testing techniques.

Code Inspections

A code inspection is a set of procedures and error-detection techniques for group code reading. Most discussions of code inspections focus on the procedures, forms to be filled out, and so on. Here, after a short summary of the general procedure, we will focus on the actual error-detection techniques.

Inspection Team

An inspection team usually consists of four people. The first of the four plays the role of moderator, which in this context is tantamount to that of a quality-control engineer. The moderator is expected to be a competent programmer, but he or she is not the author of the program and need not be acquainted with the details of the program. Moderator duties include:

- Distributing materials for, and scheduling, the inspection session.
- Leading the session.
- Recording all errors found.
- Ensuring that the errors are subsequently corrected.

The second team member is the programmer. The remaining team members usually are the program's designer (if different from the programmer) and a test specialist. The specialist should be well versed in software testing and familiar with the most common programming errors, which we discuss later in this chapter.

Inspection Agenda

Several days in advance of the inspection session, the moderator distributes the program's listing and design specification to the other participants. The participants are expected to familiarize themselves with the material prior to the session. During the session, two activities occur:

1. The programmer narrates, statement by statement, the logic of the program. During the discourse, other participants should raise questions, which should be pursued to determine whether errors exist. It is likely that the programmer, rather than the other team members, will find many of the errors identified during this narration. In other words, *the simple act of reading aloud a program to an audience seems to be a remarkably effective error-detection technique.*
2. The program is analyzed with respect to checklists of historically common programming errors (such a checklist is discussed in the next section).

The moderator is responsible for ensuring that the discussions proceed along productive lines and that the participants focus their attention on finding errors, not correcting them. (The programmer corrects errors *after* the inspection session.)

Upon the conclusion of the inspection session, the programmer is given a list of the errors uncovered. If more than a few errors were found, or if any of the errors require a substantial correction, the moderator might make arrangements to reinspect the program after those errors have been corrected. This subsequent list of errors is also analyzed, categorized, and used to refine the error checklist to improve the effectiveness of future inspections.

As stated, this inspection process usually concentrates on discovering errors, not correcting them. That said, some teams may find that when a minor problem is discovered, two or three people, including the programmer responsible for the code, may propose design changes to handle this special case. The discussion of this minor problem may, in turn, focus the group's attention on that particular area of the design. During the discussion of the best way to alter the design to handle this minor problem, someone may notice a second problem. Now that the group has seen two problems related to the same aspect of the design, comments likely will

come thick and fast, with interruptions every few sentences. In a few minutes, this whole area of the design could be thoroughly explored, and any problems made obvious.

The time and location of the inspection should be planned to prevent all outside interruptions. The optimal amount of time for the inspection session appears to be from 90 to 120 minutes. The session is a mentally taxing experience, thus longer sessions tend to be less productive. Most inspections proceed at a rate of approximately 150 program statements per hour. For that reason, large programs should be examined over multiple inspections, each dealing with one or several modules or subroutines.

Human Agenda

Note that for the inspection process to be effective, the testing group must adopt an appropriate attitude. If, for example, the programmer views the inspection as an attack on his or her character and adopts a defensive posture, the process will be ineffective. Rather, the programmer must a leave his or her ego at the door and place the process in a positive and constructive light, keeping in mind that the objective of the inspection is to find errors in the program and, thus, improve the quality of the work. For this reason, most people recommend that the results of an inspection be a confidential matter, shared only among the participants. In particular, if managers somehow make use of the inspection results (to assume or imply that the programmer is inefficient or incompetent, for example), the purpose of the process may be defeated.

Side Benefits of the Inspection Process

The inspection process has several beneficial side effects, in addition to its main effect of finding errors. For one, the programmer usually receives valuable feedback concerning programming style, choice of algorithms, and programming techniques. The other participants gain in a similar way by being exposed to another programmer's errors and programming style. In general, this type of software testing helps reinforce a team approach to this particular project and to projects that involve these participants in general. Reducing the potential for the evolution of an adversarial relationship, in favor of a cooperative, team approach to projects, can lead to more efficient and reliable program development.

Finally, the inspection process is a way of identifying early the most error-prone sections of the program, helping to focus attention more directly on these sections during the computer-based testing processes (number 9 of the testing principles given in Chapter 2).

An Error Checklist for Inspections

An important part of the inspection process is the use of a checklist to examine the program for common errors. Unfortunately, some checklists concentrate more on issues of style than on errors (e.g., "Are comments accurate and meaningful?" and "Are `if-else` code blocks, and `do-while` groups aligned?"), and the error checks are too nebulous to be useful (such as, "Does the code meet the design requirements?"). The checklist in this section, divided into six categories, was compiled after many years of study of software errors. It is largely language-independent, meaning that most of the errors can occur with any programming language. You may wish to supplement this list with errors peculiar to your programming language and with errors detected after completing the inspection process.

Data Reference Errors

- ❒ Does a referenced variable have a value that is unset or uninitialized? This probably is the most frequent programming error, occurring in a wide variety of circumstances. For each reference to a data item (variable, array element, field in a structure), attempt to "prove" informally that the item has a value at that point.
- ❒ For all array references, is each subscript value within the defined bounds of the corresponding dimension?
- ❒ For all array references, does each subscript have an integer value? This is not necessarily an error in all languages, but, in general, working with noninteger array references is a dangerous practice.
- ❒ For all references through pointer or reference variables, is the referenced memory currently allocated? This is known as the "dangling reference" problem. It occurs in situations where the lifetime of a pointer is greater than the lifetime of the referenced memory. One instance occurs where a pointer references a local variable within a procedure, the pointer value is assigned to an output parameter or a global variable, the procedure returns (freeing the referenced

location), and later the program attempts to use the pointer value. In a manner similar to checking for the prior errors, try to prove informally that, in each reference using a pointer variable, the referenced memory exists.

- When a memory area has alias names with differing attributes, does the data value in this area have the correct attributes when referenced via one of these names? Situations to look for are the use of the *EQUIVALENCE* statement in Fortran and the *REDEFINES* clause in COBOL. As an example, a Fortran program contains a real variable *A* and an integer variable *B*; both are made aliases for the same memory area by using an *EQUIVALENCE* statement. If the program stores a value into *A* and then references variable *B*, an error is likely present since the machine would use the floating-point bit representation in the memory area as an integer.

Sidebar 3.1: History of COBOL and Fortran

COBOL and Fortran are older programming languages that have fueled business and scientific software development for generations of computer hardware, operating systems and programmers.

COBOL (an acronym for **CO**mmon **B**usiness **O**riented **L**anguage) first was defined about 1959 or 1960, and was designed to support business application development on mainframe class computers. The original specification included aspects of other existing languages at the time. Big-name computer manufacturers and representatives of the federal government participated in this effort to create a business-oriented programming language that could run on a variety of hardware and operating system platforms.

COBOL language standards have been reviewed and updated over the years. By 2002, COBOL was available for most current operating platforms and object-oriented versions supporting the .NET development environment.

As the time of this writing, the latest version of COBOL is Visual COBOL 2010.

Fortran (originally FORTRAN, but modern references generally follow the uppercase/lowercase syntax) is a little older than COBOL,

with early specifications defined in the early to middle 1950s. Like COBOL, Fortran was designed for specific types of mainframe application development, but in the scientific and numerical management arenas. The name derives from an existing IBM system at the time, Mathematical **FOR**mula **TRAN**slating System. Although the original Fortran contained only 32 statements, it marked a significant improvement over assembly-level programming that preceded it.

The current version as of the publication date of this book is Fortran 2008, formally approved by the appropriate standard committees in 2010. Like COBOL, the evolution of Fortran added support for a broad range of hardware and operating system platforms. However, Fortran is probably used more in current development—as well as older system maintenance—than COBOL.

- Does a variable's value have a type or attribute other than what the compiler expects? This situation might occur where a C or C++ program reads a record into memory and references it by using a structure, but the physical representation of the record differs from the structure definition.
- Are there any explicit or implicit addressing problems if, on the computer being used, the units of memory allocation are smaller than the units of addressable memory? For instance, in some environments, fixed-length bit strings do not necessarily begin on byte boundaries, but address only point-to-byte boundaries. If a program computes the address of a bit string and later refers to the string through this address, the wrong memory location may be referenced. This situation also could occur when passing a bit-string argument to a subroutine.
- If pointer or reference variables are used, does the referenced memory location have the attributes the compiler expects? An example of such an error is where a C++ pointer upon which a data structure is based is assigned the address of a different data structure.
- If a data structure is referenced in multiple procedures or subroutines, is the structure defined identically in each procedure?
- When indexing into a string, are the limits of the string off by one in indexing operations or in subscript references to arrays?

- ❒ For object-oriented languages, are all inheritance requirements met in the implementing class?

Data Declaration Errors

- ❒ Have all variables been explicitly declared? A failure to do so is not necessarily an error, but is, nevertheless, a common source of trouble. For instance, if a program subroutine receives an array parameter, and fails to define the parameter as an array (as in a *DIMENSION* statement), a reference to the array (such as *C=A(I)*) is interpreted as a function call, leading to the machine's attempting to execute the array as a program. Also, if a variable is not explicitly declared in an inner procedure or block, is it understood that the variable is shared with the enclosing block?
- ❒ If all attributes of a variable are not explicitly stated in the declaration, are the defaults well understood? For instance, the default attributes received in Java are often a source of surprise when not properly declared.
- ❒ Where a variable is initialized in a declarative statement, is it properly initialized? In many languages, initialization of arrays and strings is somewhat complicated and, hence, error prone.
- ❒ Is each variable assigned the correct length and data type?
- ❒ Is the initialization of a variable consistent with its memory type? For instance, if a variable in a Fortran subroutine needs to be reinitialized each time the subroutine is called, it must be initialized with an assignment statement rather than a *DATA* statement.
- ❒ Are there any variables with similar names (e.g., *VOLT* and *VOLTS*)? This is not necessarily an error, but it should be seen as a warning that the names may have been confused somewhere within the program.

Computation Errors

- ❒ Are there any computations using variables having inconsistent (such as nonarithmetic) data types?
- ❒ Are there any mixed-mode computations? An example is when working with floating-point and integer variables. Such occurrences are not necessarily errors, but they should be explored carefully to ensure that the conversion rules of the language are understood.

Consider the following Java snippet showing the rounding error that can occur when working with integers:

```
int x = 1;
int y = 2;
int z = 0;
z = x/y;
System.out.println ("z = " + z);

OUTPUT:
z = 0
```

- Are there any computations using variables having the same data type but of different lengths?
- Is the data type of the target variable of an assignment smaller than the data type or a result of the right-hand expression?
- Is an overflow or underflow expression possible during the computation of an expression? That is, the end result may appear to have valid value, but an intermediate result might be too big or too small for the programming language's data types.
- Is it possible for the divisor in a division operation to be zero?
- If the underlying machine represents variables in base-2 form, are there any sequences of the resulting inaccuracy? That is, `10 * 0.1` is rarely equal to 1.0 on a binary machine.
- Where applicable, can the value of a variable go outside the meaningful range? For example, statements assigning a value to the variable *PROBABILITY* might be checked to ensure that the assigned value will always be positive and not greater than 1.0.
- For expressions containing more than one operator, are the assumptions about the order of evaluation and precedence of operators correct?
- Are there any invalid uses of integer arithmetic, particularly divisions? For instance, if *i* is an integer variable, whether the expression `2*i/2 == i` depends on whether *i* has an odd or an even value and whether the multiplication or division is performed first.

Comparison Errors

- Are there any comparisons between variables having different data types, such as comparing a character string to an address, date, or number?

- Are there any mixed-mode comparisons or comparisons between variables of different lengths? If so, ensure that the conversion rules are well understood.
- Are the comparison operators correct? Programmers frequently confuse such relations as *at most, at least, greater than, not less than,* and *less than or equal.*
- Does each Boolean expression state what it is supposed to state? Programmers often make mistakes when writing logical expressions involving *and, or*, and *not.*
- Are the operands of a Boolean operator Boolean? Have comparison and Boolean operators been erroneously mixed together? This represents another frequent class of mistakes. Examples of a few typical mistakes are illustrated here:
  - If you want to determine whether *i* is between 2 and 10, the expression `2<i<10` is incorrect. Instead, it should be `(2<i)&&(i<10)`.
  - If you want to determine whether *i* is greater than *x* or *y*, `i>x||y` is incorrect. Instead, it should be `(i>x)||(i>y)`.
  - If you want to compare three numbers for equality, `if(a==b==c)` does something quite different.
  - If you want to test the mathematical relation `x>y>z`, the correct expression is `(x>y)&&(y>z)`.
- Are there any comparisons between fractional or floating-point numbers that are represented in base-2 by the underlying machine? This is an occasional source of errors because of truncation and base-2 approximations of base-10 numbers.
- For expressions containing more than one Boolean operator, are the assumptions about the order of evaluation and the precedence of operators correct? That is, if you see an expression such as `if((a==2)&&(b==2)||(c==3))`, is it well understood whether the *and* or the *or* is performed first?
- Does the way in which the compiler evaluates Boolean expressions affect the program? For instance, the statement

```
if(x==0&&(x/y)>z)
```

  may be acceptable for compilers that end the test as soon as one side of an *and* is false, but may cause a division-by-zero error with other compilers.

Control-Flow Errors

- If the program contains a multipath branch such as a computed `GOTO`, can the index variable ever exceed the number of branch possibilities? For example, in the statement

```
GOTO(200,300,400),i
```

  will `i` always have the value of 1, 2, or 3?
- Will every loop eventually terminate? Devise an informal proof or argument showing that each loop will terminate.
- Will the program, module, or subroutine eventually terminate?
- Is it possible that, because of the conditions upon entry, a loop will never execute? If so, does this represent an oversight? For instance, if you had the following `for` loop and `while` loop headed by the following statements:

```
for (i=x;i<=z;i++){
...
}
```

  or . . .

```
while (NOTFOUND) {
...
}
```

  what happens if `x` is greater than `z` or if `NOTFOUND` is initially false?
- For a loop controlled by both iteration and a Boolean condition (e.g., a searching loop) what are the consequences of loop fall-through? For example, for the psuedo-code loop headed by

```
DO I=1 to TABLESIZE WHILE (NOTFOUND)
```

  what happens if `NOTFOUND` never becomes false?
- Are there any off-by-one errors, such as one too many or too few iterations? This is a common error in zero-based loops. You will often forget to count 0 as a number. For example, if you want to create Java code for a loop that iterates 10 times, the following would be wrong, as it performs 11 iterations:

```
for (int i=0;i<=10;i++){
    System.out.println(i);
}
```

Correct, the loop is iterated 10 times:

```
for (int i=0; i<10;i++) {
    System.out.println(i);
}
```

- If the language contains a concept of statement groups or code blocks (e.g., `do-while` or `{...}`), is there an explicit `while` for each group, and do the instances of *do* correspond to their appropriate groups? Is there a closing bracket for each open bracket? Most modern compilers will complain of such mismatches.
- Are there any nonexhaustive decisions? For instance, if an input parameter's expected values are 1, 2, or 3, does the logic assume that it must be 3 if it is not 1 or 2? If so, is the assumption valid?

Interface Errors

- Does the number of parameters received by this module equal the number of arguments sent by each of the calling modules? Also, is the order correct?
- Do the attributes (e.g., data type and size) of each parameter match the attributes of each corresponding argument?
- Does the units system of each parameter match the units system of each corresponding argument? For example, is the parameter expressed in degrees but the argument expressed in radians?
- Does the number of arguments passed by this module to another module equal the number of parameters expected by that module?
- Do the attributes of each argument passed to another module match the attributes of the corresponding parameter in that module?
- Does the units system of each argument passed to another module match the units system of the corresponding parameter in that module?
- If built-in functions are invoked, are the number, attributes, and order of the arguments correct?
- If a module or class has multiple entry points, is a parameter ever referenced that is not associated with the current point of entry? Such an error exists in the second assignment statement in the following PL/1 program:

```
A:    PROCEDURE (W,X);
       W=X+1;
       RETURN
B:    ENTRY (Y,Z);
       Y=X+Z;
       END;
```

- ❒ Does a subroutine alter a parameter that is intended to be only an input value?
- ❒ If global variables are present, do they have the same definition and attributes in all modules that reference them?
- ❒ Are constants ever passed as arguments? In some Fortran implementations a statement such as

```
CALL SUBX(J,3)
```

  is dangerous, because if the subroutine SUBX assigns a value to its second parameter, the value of the constant 3 will be altered.

Input/Output Errors

- ❒ If files are explicitly declared, are their attributes correct?
- ❒ Are the attributes on the file's *OPEN* statement correct?
- ❒ Does the format specification agree with the information in the I/O statement? For instance, in Fortran, does each *FORMAT* statement agree (in terms of the number and attributes of the items) with the corresponding *READ* or *WRITE* statement?
- ❒ Is sufficient memory available to hold the file your program will read?
- ❒ Have all files been opened before use?
- ❒ Have all files been closed after use?
- ❒ Are end-of-file conditions detected and handled correctly?
- ❒ Are I/O error conditions handled correctly?
- ❒ Are there spelling or grammatical errors in any text that is printed or displayed by the program?
- ❒ Does the program properly handle "File not Found" errors?

Other Checks

- ❒ If the compiler produces a cross-reference listing of identifiers, examine it for variables that are never referenced or are referenced only once.

- ❒ If the compiler produces an attribute listing, check the attributes of each variable to ensure that no unexpected default attributes have been assigned.
- ❒ If the program compiled successfully, but the computer produced one or more "warning" or "informational" messages, check each one carefully. Warning messages are indications that the compiler suspects you are doing something of questionable validity: Review all of these suspicions. Informational messages may list undeclared variables or language uses that impede code optimization.
- ❒ Is the program or module sufficiently robust? That is, does it check its input for validity?
- ❒ Is a function missing from the program?

This checklist is summarized in Tables 3.1 and 3.2.

Walkthroughs

The code walkthrough, like the inspection, is a set of procedures and error-detection techniques for group code reading. It shares much in common with the inspection process, but the procedures are slightly different, and a different error-detection technique is employed.

Like the inspection, the walkthrough is an uninterrupted meeting of one to two hours in duration. The walkthrough team consists of three to five people. One of these people plays a role similar to that of the moderator in the inspection process; another person plays the role of a secretary (a person who records all errors found); and a third person plays the role of a tester. Suggestions as to who the three to five people should be vary. Of course, the programmer is one of those people. Suggestions for the other participants include:

- A highly experienced programmer
- A programming-language expert
- A new programmer (to give a fresh, unbiased outlook)
- The person who will eventually maintain the program
- Someone from a different project
- Someone from the same programming team as the programmer

TABLE 3.1 Inspection Error Checklist Summary, Part I

Data Reference	Computation
1. Unset variable used?	1. Computations on nonarithmetic variables?
2. Subscripts within bounds?	2. Mixed-mode computations?
3. Noninteger subscripts?	3. Computations on variables of different lengths?
4. Dangling references?	4. Target size less than size of assigned value?
5. Correct attributes when aliasing?	5. Intermediate result overflow or underflow?
6. Record and structure attributes match?	6. Division by zero?
7. Computing addresses of bit strings? Passing bit-string arguments?	7. Base-2 inaccuracies?
8. Based storage attributes correct?	8. Variable's value outside of meaningful range?
9. Structure definitions match across procedures?	9. Operator precedence understood?
10. Off-by-one errors in indexing or subscripting operations?	10. Integer divisions correct?
11. Inheritance requirements met?	
Data Declaration	**Comparison**
1. All variables declared?	1. Comparisons between inconsistent variables?
2. Default attributes understood?	2. Mixed-mode comparisons?
3. Arrays and strings initialized properly?	3. Comparison relationships correct?
4. Correct lengths, types, and storage classes assigned?	4. Boolean expressions correct?
5. Initialization consistent with storage class?	5. Comparison and Boolean expressions mixed?
6. Any variables with similar names?	6. Comparisons of base-2 fractional values?
	7. Operator precedence understood?
	8. Compiler evaluation of Boolean expressions understood?

TABLE 3.2 Inspection Error Checklist Summary, Part II

Control Flow	Input/Output
1. Multiway branches exceeded?	1. File attributes correct?
2. Will each loop terminate?	2. OPEN statements correct?
3. Will program terminate?	3. Format specification matches I/O statement?
4. Any loop bypasses because of entry conditions?	4. Buffer size matches record size?
5. Possible loop fall-throughs correct?	5. Files opened before use?
6. Off-by-one iteration errors?	6. Files closed after use?
7. DO/END statements match?	7. End-of-file conditions handled?
8. Any nonexhaustive decisions?	8. I/O errors handled?
9. Any textual or grammatical errors in output information?	
Interfaces	**Other Checks**
1. Number of input parameters equal to number of arguments?	1. Any unreferenced variables in cross-reference listing?
2. Parameter and argument attributes match?	2. Attribute list what was expected?
3. Parameter and argument units system match?	3. Any warning or informational messages?
4. Number of arguments transmitted to called modules equal to number of parameters?	4. Input checked for validity?
5. Attributes of arguments transmitted to called modules equal to attributes of parameters?	5. Missing function?
6. Units system of arguments transmitted to called modules equal to units system of parameters?	
7. Number, attributes, and order of arguments to built-in functions correct?	
8. Any references to parameters not associated with current point of entry?	
9. Input-only arguments altered?	
10. Global variable definitions consistent across modules?	
11. Constants passed as arguments?	

The initial procedure is identical to that of the inspection process: The participants are given the materials several days in advance, to allow them time to bone up on the program. However, the procedure in the meeting is different. Rather than simply reading the program or using error checklists, the participants "play computer." The person designated as the tester comes to the meeting armed with a small set of paper test cases—representative sets of inputs (and expected outputs) for the program or module. During the meeting, each test case is mentally executed; that is, the test data are "walked through" the logic of the program. The state of the program (i.e., the values of the variables) is monitored on paper or a whiteboard.

Of course, the test cases must be simple in nature and few in number, because people execute programs at a rate that is many orders of magnitude slower than a machine. Hence, the test cases themselves do not play a critical role; rather, they serve as a vehicle for getting started and for questioning the programmer about his or her logic and assumptions. In most walkthroughs, more errors are found during the process of questioning the programmer than are found directly by the test cases themselves.

As in the inspection, the attitude of the participants is critical. Comments should be directed toward the program rather than the programmer. In other words, errors are not regarded as weaknesses in the person who committed them. Rather, they are viewed as inherent to the difficulty of the program development.

The walkthrough should have a follow-up process similar to that described for the inspection process. Also, the side effects observed from inspections (identification of error-prone sections and education in errors, style, and techniques) also apply to the walkthrough process.

Desk Checking

A third human error-detection process is the older practice of desk checking. A desk check can be viewed as a one-person inspection or walkthrough: A person reads a program, checks it with respect to an error list, and/or walks test data through it.

For most people, desk checking is relatively unproductive. One reason is that it is a completely undisciplined process. A second, and more important, reason is that it runs counter to testing principle 2 (see Chapter 2),

which states that people are generally ineffective in testing their own programs. For this reason, you could deduce that desk checking is best performed by a person other than the author of the program (e.g., two programmers might swap programs rather than desk check their own), but even this is less effective than the walkthrough or inspection process. The reason is the synergistic effect of the walkthrough or inspection team. The team session fosters a healthy environment of competition; people like to show off by finding errors. In a desk-checking process, there is no one to whom you can show off, thereby precluding this apparently valuable effect. In short, desk checking may be more valuable than doing nothing at all, but it is much less effective than the inspection or walkthrough.

Peer Ratings

The last human review process is not associated with program testing (i.e., its objective is not to find errors). Nevertheless, we include this process here because it is related to the idea of code reading.

Peer rating is a technique of evaluating anonymous programs in terms of their overall quality, maintainability, extensibility, usability, and clarity. The purpose of the technique is to provide programmer self-evaluation.

A programmer is selected to serve as an administrator of the process. The administrator, in turn, selects approximately 6 to 20 participants (6 is the minimum to preserve anonymity). The participants are expected to have similar backgrounds (e.g., don't group Java application programmers with assembly language system programmers). Each participant is asked to select two of his or her own programs to be reviewed. One program should be representative of what the participant considers to be his or her finest work; the other should be a program that the programmer considers to be poorer in quality.

Once the programs have been collected, they are randomly distributed to the participants. Each participant is given four programs to review. Two of the programs are the "finest" programs and two are "poorer" programs, but the reviewer is not told which is which. Each participant spends 30 minutes reviewing each program and then completes an evaluation form. After reviewing all four programs, each participant rates the relative quality of the four programs. The evaluation form asks the reviewer to

answer, on a scale from 1 to 10 (1 meaning definitely yes and 10 meaning definitely no), such questions as:

- Was the program easy to understand?
- Was the high-level design visible and reasonable?
- Was the low-level design visible and reasonable?
- Would it be easy for you to modify this program?
- Would you be proud to have written this program?

The reviewer also is asked for general comments and suggested improvements.

After the review, the participants are given the anonymous evaluation forms for their two contributed programs. They also are given a statistical summary showing the overall and detailed ranking of their original programs across the entire set of programs, as well as an analysis of how their ratings of other programs compared with those ratings of other reviewers of the same program. The purpose of the process is to allow programmers to self-assess their programming skills. As such, the process appears to be useful in both industrial and classroom environments.

Summary

This chapter discussed a form of testing that developers do not often consider: human code testing. Most people assume that because programs are written for machine execution, machines should test programs as well. This assumption is invalid. Human testing techniques are very effective at revealing errors. In fact, most programming projects should include the following human testing techniques:

- Code inspections using checklists
- Group walkthroughs
- Desk checking
- Peer reviews

Another form of human testing is user or usability testing, a black-box technique that evaluates software from a hands-on, end-user perspective. We cover this topic in detail in Chapter 7.

4 Test-Case Design

Moving beyond the psychological issues discussed in Chapter 2, the most important consideration in program testing is the design and creation of effective test cases.

Testing, however creative and seemingly complete, cannot guarantee the absence of all errors. Test-case design is so important because complete testing is impossible. Put another way, a test of any program must be necessarily incomplete. The obvious strategy, then, is to try to make tests as complete as possible.

Given constraints on time and cost, the key issue of testing becomes:

What subset of all possible test cases has the highest probability of detecting the most errors?

The study of test-case design methodologies supplies answers to this question.

In general, the least effective methodology of all is random-input testing—the process of testing a program by selecting, at random, some subset of all possible input values. In terms of the likelihood of detecting the most errors, a randomly selected collection of test cases has little chance of being an optimal, or even close to optimal, subset. Therefore, in this chapter, we want to develop a set of thought processes that enable you to select test data more intelligently.

Chapter 2 showed that exhaustive black-box and white-box testing are, in general, impossible; at the same time, it suggested that a reasonable testing strategy might feature elements of both. This is the strategy developed in this chapter. You can develop a reasonably rigorous test by using certain black-box–oriented test-case design methodologies and then supplementing these test cases by examining the logic of the program, using white-box methods.

The methodologies discussed in this chapter are:

Black Box	White Box
Equivalence partitioning	Statement coverage
Boundary value analysis	Decision coverage
Cause-effect graphing	Condition coverage
Error guessing	Decision/condition coverage
	Multiple-condition coverage

Although we will discuss these methods separately, we recommend that you use a combination of most, if not all, of them to design a rigorous test of a program, since each method has distinct strengths and weaknesses. One method may find errors another method overlooks, for example.

Nobody ever promised that software testing would be easy. To quote an old sage, "If you thought designing and coding that program was hard, you ain't seen nothing yet."

The recommended procedure is to develop test cases using the black-box methods and then develop supplementary test cases, as necessary, with white-box methods. We'll discuss the more widely known white-box methods first.

White-Box Testing

White-box testing is concerned with the degree to which test cases exercise or cover the logic (source code) of the program. As we saw in Chapter 2, the ultimate white-box test is the execution of every path in the program; but complete path testing is not a realistic goal for a program with loops.

Logic Coverage Testing

If you back completely away from path testing, it may seem that a worthy goal would be to execute every statement in the program at least once. Unfortunately, this is a weak criterion for a reasonable white-box test. This concept is illustrated in Figure 4.1. Assume that this figure represents a small program to be tested. The equivalent Java code snippet follows:

```
public void foo(int A,int B,int X) {
    if(A>1 && B==0) {
        X=X/A;
    }
    if(A==2 || X>1) {
        X=X+1;
    }
}
```

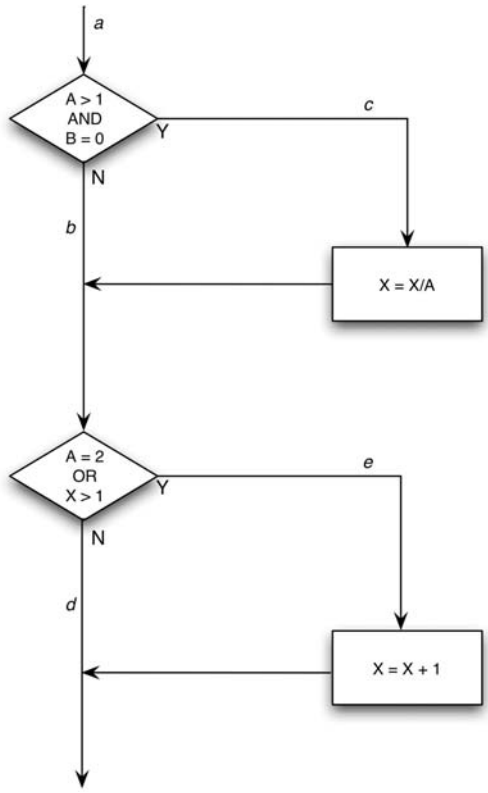

FIGURE 4.1 A Small Program to Be Tested.

You could execute every statement by writing a single test case that traverses path *ace*. That is, by setting A=2, B=0, and X=3 at point a, every statement would be executed once (actually, *X* could be assigned any integer value >1).

Unfortunately, this criterion is a rather poor one. For instance, perhaps the first decision should be an *or* rather than an *and*. If so, this error would go undetected. Perhaps the second decision should have stated X>0; this error would not be detected. Also, there is a path through the program in which *X* goes unchanged (the path *abd*). If this were an error, it would go undetected. In other words, the statement coverage criterion is so weak that it generally is useless.

A stronger logic coverage criterion is known as *decision coverage* or *branch coverage*. This criterion states that you must write enough test cases that each decision has a true and a false outcome at least once. In other words, each branch direction must be traversed at least once. Examples of branch or decision statements are *switch-case, do-while*, and *if-else* statements. Multipath *GOTO* statements qualify in some programming languages such as Fortran.

Decision coverage usually can satisfy statement coverage. Since every statement is on some subpath emanating either from a branch statement or from the entry point of the program, every statement must be executed if every branch direction is executed. There are, however, at least three exceptions:

- Programs with no decisions.
- Programs or subroutines/methods with multiple entry points. A given statement might be executed only if the program is entered at a particular entry point.
- Statements within *ON*-units. Traversing every branch direction will not necessarily cause all *ON*-units to be executed.

Since we have deemed statement coverage to be a necessary condition, decision coverage, a seemingly better criterion, should be defined to include statement coverage. Hence, decision coverage requires that each decision have a true and a false outcome, and that each statement be executed at least once. An alternative and easier way of expressing it is that each decision has a true and a false outcome, and that each point of entry (including *ON*-units) be invoked at least once.

This discussion considers only two-way decisions or branches and has to be modified for programs that contain multipath decisions. Examples are Java programs containing *switch-case* statements, Fortran programs containing arithmetic (three-way) *IF* statements or computed or arithmetic *GOTO* statements, and COBOL programs containing altered *GOTO* statements or *GO-TO-DEPENDING-ON* statements. For such programs, the criterion is exercising each possible outcome of all decisions at least once and invoking each point of entry to the program or subroutine at least once.

In Figure 4.1, decision coverage can be met by two test cases covering paths *ace* and *abd* or, alternatively, *acd* and *abe*. If we choose the latter alternative, the two test-case inputs are A=3, B=0, X=3 and A=2, B=1, and X=1.

Decision coverage is a stronger criterion than statement coverage, but it still is rather weak. For instance, there is only a 50 percent chance that we would explore the path where *x* is not changed (i.e., only if we chose the former alternative). If the second decision were in error (if it should have said X<1 instead of X>1), the mistake would not be detected by the two test cases in the previous example.

A criterion that is sometimes stronger than decision coverage is *condition coverage*. In this case, you write enough test cases to ensure that each condition in a decision takes on all possible outcomes at least once. But, as with decision coverage, this does not always lead to the execution of each statement, so an addition to the criterion is that each point of entry to the program or subroutine, as well as *ON*-units, be invoked at least once. For instance, the branching statement:

```
DO K=0 to 50 WHILE (J+K<QUEST)
```

contains two conditions: Is K less than or equal to 50, and is J+K less than QUEST? Hence, test cases would be required for the situations K<=50, K>50 (to reach the last iteration of the loop), J+K<QUEST, and J+K>=QUEST.

Figure 4.1 has four conditions: A>1, B=0, A=2, and X>1. Hence, enough test cases are needed to force the situations where A>1, A<=1, B=0, and B<>0 are present at point a and where A=2, A<>2, X>1, and X<=1 are present at point b. A sufficient number of test cases satisfying the criterion, and the paths traversed by each, are:

```
A=2, B=0, X=4  ace
A=1, B=1, X=1  adb
```

Note that although the same number of test cases was generated for this example, condition coverage usually is superior to decision coverage in that it *may* (but does not always) cause every individual condition in a decision to be executed with both outcomes, whereas decision coverage does not. For instance, in the same branching statement

```
DO K=0 to 50 WHILE (J+K<QUEST)
```

is a two-way branch (execute the loop body or skip it). If you are using decision testing, the criterion can be satisfied by letting the loop run from `K=0 to 51`, *without ever exploring the circumstance where the* `WHILE` *clause becomes false*. With the condition criterion, however, a test case would be needed to generate a *false* outcome for the conditions `J+K<QUEST`.

Although the condition coverage criterion appears, at first glance, to satisfy the decision coverage criterion, it does not always do so. If the decision `IF(A & B)` is being tested, the condition coverage criterion would let you write two test cases—`A` is *true,* `B` is *false,* and `A` is *false*, `B` is *true*—but this would not cause the `THEN` clause of the `IF` to execute. The condition coverage tests for the earlier example covered all decision outcomes, but this was only by chance. For instance, two alternative test cases

```
A=1, B=0, X=3
A=2, B=1, X=1
```

cover all condition outcomes but only two of the four decision outcomes (both of them cover path ***abe*** and, hence, do not exercise the *true* outcome of the first decision and the *false* outcome of the second decision).

The obvious way out of this dilemma is a criterion called *decision/condition coverage*. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once.

A weakness with decision/condition coverage is that although it may appear to exercise all outcomes of all conditions, it frequently does not, because certain conditions mask other conditions. To see this, examine Figure 4.2. The flowchart in this figure is the way a compiler would generate machine code for the program in Figure 4.1. The multicondition decisions in the source program have been broken into individual decisions and branches because most machines do not have a single instruction that makes multicondition decisions. A more thorough test coverage, then,

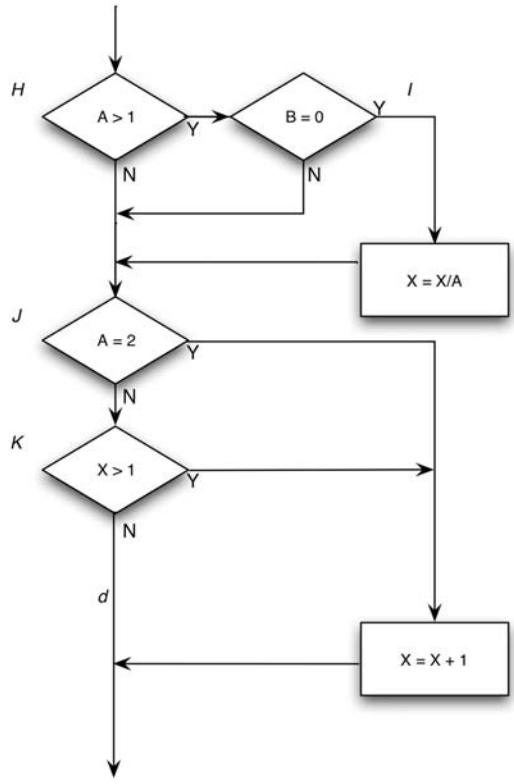

FIGURE 4.2 Machine Code for the Program in Figure 4.1.

appears to be the exercising of all possible outcomes of each primitive decision. The two previous decision coverage test cases do not accomplish this; they fail to exercise the *false* outcome of decision H and the *true* outcome of decision K.

The reason, as shown in Figure 4.2, is that results of conditions in the *and* and the *or* expressions can mask or block the evaluation of other conditions. For instance, if an *and* condition is false, none of the subsequent conditions in the expression need be evaluated. Likewise, if an *or* condition is true, none of the subsequent conditions need be evaluated. Hence, errors in logical expressions are not necessarily revealed by the condition coverage and decision/condition coverage criteria.

A criterion that covers this problem, and then some, is *multiple-condition coverage.* This criterion requires that you write sufficient test cases such that all possible combinations of condition outcomes in each decision, and all

points of entry, are invoked at least once. For instance, consider the following sequence of pseudo-code.

```
NOTFOUND=TRUE;
DO I=1 to TABSIZE WHILE (NOTFOUND); /*SEARCH TABLE*/
    .. . searching logic.. . ;
END
```

The four situations to be tested are:

1. I<=TABSIZE and NOTFOUND is *true*.
2. I<=TABSIZE and NOTFOUND is *false* (finding the entry before hitting the end of the table).
3. I>TABSIZE and NOTFOUND is *true* (hitting the end of the table without finding the entry).
4. I>TABSIZE and NOTFOUND is *false* (the entry is the last one in the table).

It should be easy to see that a set of test cases satisfying the multiple-condition criterion also satisfies the decision coverage, condition coverage, and decision/condition coverage criteria.

Returning to Figure 4.1, test cases must cover eight combinations:

1. A>1, B=0	5. A=2, X>1
2. A>1, B<>0	6. A=2, X<=1
3. A<=1, B=0	7. A<>2, X>1
4. A<=1, B<>0	8. A<>2, X<=1

Note Recall from the Java code snippet presented earlier that test cases 5 through 8 express values at the point of the second *if* statement. Since *X* may be altered above this *if* statement, the values needed at this *if* statement must be backed up through the logic to find the corresponding input values.

These combinations to be tested do not necessarily imply that eight test cases are needed. In fact, they can be covered by four test cases. The test-case input values, and the combinations they cover, are as follows:

A=2, B=0, X=4	Covers 1, 5
A=2, B=1, X=1	Covers 2, 6

```
A=1, B=0, X=2      Covers 3, 7
A=1, B=1, X=1      Covers 4, 8
```

The fact that there are four test cases and four distinct paths in Figure 4.1 is just coincidence. In fact, these four test cases do not cover every path; they miss the path *acd*. For instance, you would need eight test cases for the following decision:

```
if(x==y && length(z)==0 && FLAG) {
       j=1;
else
       i=1;
}
```

although it contains only two paths. In the case of loops, the number of test cases required by the multiple-condition criterion is normally much less than the number of paths.

In summary, for programs containing only one condition per decision, a minimum test criterion is a sufficient number of test cases to: (1) invoke all outcomes of each decision at least once, and (2) invoke each point of entry (such as entry point or *ON*-unit) at least once, to ensure that all statements are executed at least once. For programs containing decisions having multiple conditions, the minimum criterion is a sufficient number of test cases to invoke all possible combinations of condition outcomes in each decision, and all points of entry to the program, at least once. (The word "possible" is inserted because some combinations may be found to be impossible to create.)

Black-Box Testing

As we discussed in Chapter 2, black-box (data-driven or input/output driven) testing is based on program specifications. The goal is to find areas wherein the program does not behave according to its specifications.

Equivalence Partitioning

Chapter 2 described a good test case as one that has a reasonable probability of finding an error; it also stated that an exhaustive input test of a program is impossible. Hence, when testing a program, you are limited to a

small subset of all possible inputs. Of course, then, you want to select the "right" subset, that is, the subset with the highest probability of finding the most errors.

One way of locating this subset is to realize that a well-selected test case also should have two other properties:

1. It reduces, by more than a count of one, the number of other test cases that must be developed to achieve some predefined goal of "reasonable" testing.
2. It covers a large set of other possible test cases. That is, it tells us something about the presence or absence of errors over and above this specific set of input values.

These properties, although they appear to be similar, describe two distinct considerations. The first implies that each test case should invoke as many different input considerations as possible to minimize the total number of test cases necessary. The second implies that you should try to partition the input domain of a program into a finite number of *equivalence classes* such that you can reasonably assume (but, of course, not be absolutely sure) that a test of a representative value of each class is equivalent to a test of any other value. That is, if one test case in an equivalence class detects an error, all other test cases in the equivalence class would be expected to find the same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the equivalence class would fall within another equivalence class, since equivalence classes may overlap one another.

These two considerations form a black-box methodology known as *equivalence partitioning*. The second consideration is used to develop a set of "interesting" conditions to be tested. The first consideration is then used to develop a minimal set of test cases covering these conditions.

An example of an equivalence class in the triangle program of Chapter 1 is the set "three equal-valued numbers having integer values greater than zero." By identifying this as an equivalence class, we are stating that if no error is found by a test of one element of the set, it is unlikely that an error would be found by a test of another element of the set. In other words, our testing time is best spent elsewhere: in different equivalence classes.

Test-case design by equivalence partitioning proceeds in two steps: (1) identifying the equivalence classes and (2) defining the test cases.

External condition	Valid equivalence classes	Invalid equivalence classes

FIGURE 4.3 A Form for Enumerating Equivalence Classes.

Identifying the Equivalence Classes The equivalence classes are identified by taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups. You can use the table in Figure 4.3 to do this. Notice that two types of equivalence classes are identified: *valid equivalence* classes represent valid inputs to the program, and *invalid equivalence* classes represent all other possible states of the condition (i.e., erroneous input values). Thus, we are adhering to principle 5, discussed in Chapter 2, which stated you must focus attention on invalid or unexpected conditions.

Given an input or external condition, identifying the equivalence classes is largely a heuristic process. Follow these guidelines:

1. If an input condition specifies a range of values (e.g., "the item count can be from 1 to 999"), identify one valid equivalence class (`1<item count<999`) and two invalid equivalence classes (`item count<1` and `item count>999`).
2. If an input condition specifies the number of values (e.g., "one through six owners can be listed for the automobile"), identify one valid equivalence class and two invalid equivalence classes (no owners and more than six owners).
3. If an input condition specifies a set of input values, and there is reason to believe that the program handles each differently ("type

of vehicle must be BUS, TRUCK, TAXICAB, PASSENGER, or MOTORCYCLE"), identify a valid equivalence class for each and one invalid equivalence class ("TRAILER," for example).
4. If an input condition specifies a "must-be" situation, such as "first character of the identifier must be a letter," identify one valid equivalence class (it is a letter) and one invalid equivalence class (it is not a letter).

If there is any reason to believe that the program does not handle elements in an equivalence class identically, split the equivalence class into smaller equivalence classes. We will illustrate an example of this process shortly.

Identifying the Test Cases

The second step is the use of equivalence classes to identify the test cases. The process is as follows:

1. Assign a unique number to each equivalence class.
2. Until all valid equivalence classes have been covered by (incorporated into) test cases, write a new test case covering as many of the uncovered valid equivalence classes as possible.
3. Until your test cases have covered all invalid equivalence classes, write a test case that covers one, and only one, of the uncovered invalid equivalence classes.

The reason that individual test cases cover invalid cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states "enter book type (HARDCOVER, SOFTCOVER, or LOOSE) and amount (1–999)," the test case, (*XYZ 0*), expressing two error conditions (invalid book type and amount) will probably not exercise the check for the amount, since the program may say "XYZ IS UNKNOWN BOOK TYPE" and not bother to examine the remainder of the input.

An Example

As an example, assume that we are developing a compiler for a subset of the Fortran language, and we wish to test the syntax checking of the *DIMENSION* statement. The specification is listed below. (Note: This is not

the full Fortran *DIMENSION* statement; it has been edited considerably to make it textbook size. Do not be deluded into thinking that the testing of actual programs is as easy as the examples in this book.) In the specification, items in italics indicate syntactic units for which specific entities must be substituted in actual statements; brackets are used to indicate option items; and an ellipsis indicates that the preceding item may appear multiple times in succession.

> A *DIMENSION* statement is used to specify the dimensions of arrays. The form of the *DIMENSION* statement is
>
> DIMENSION *ad*[,ad]...
>
> where ad is an array descriptor of the form
>
> n(d[,d]...)
>
> where n is the symbolic name of the array and d is a dimension declarator. Symbolic names can be one to six letters or digits, the first of which must be a letter. The minimum and maximum numbers of dimension declarations that can be specified for an array are one and seven, respectively. The form of a dimension declarator is
>
> [lb:]ub
>
> where lb and ub are the lower and upper dimension bounds. A bound may be a constant in the range −65534 to 65535 or the name of an integer variable (but not an array element name). If lb is not specified, it is assumed to be 1. The value of ub must be greater than or equal to lb. If lb is specified, its value may be negative, 0, or positive. As for all statements, the *DIMENSION* statement may be continued over multiple lines.

The first step is to identify the input conditions and, from these, locate the equivalence classes. These are tabulated in Table 4.1. The numbers in the table are unique identifiers of the equivalence classes.

The next step is to write a test case covering one or more valid equivalence classes. For instance, the test case

```
DIMENSION A(2)
```

covers classes 1, 4, 7, 10, 12, 15, 24, 28, 29, and 43.

TABLE 4.1 Equivalence Classes

Input Condition	Valid Equivalence Classes	Invalid Equivalence Classes
Number of array descriptors	one (1), > one (2)	none (3)
Size of array name	1–6 (4)	0 (5), >6 (6)
Array name	has letters (7), has digits (8)	has something else (9)
Array name starts with letter	yes (10)	no (11)
Number of dimensions	1–7 (12)	0 (13), >7 (14)
Upper bound is	constant (15), integer variable (16)	array element name (17), something else (18)
Integer variable name	has letter (19), has digits (20)	has something else (21)
Integer variable starts with letter	yes (22)	no (23)
Constant	–65534–65535 (24)	<–65534 (25), >65535 (26)
Lower bound specified	yes (27), no (28)	
Upper bound to lower bound	greater than (29), equal (30)	less than (31)
Specified lower bound	negative (32), zero (33), > 0 (34)	
Lower bound is	constant (35), integer variable (36)	array element name (37), something else (38)
Lower bound is	one (39)	ub>=1 (40), ub<1 (41)
Multiple lines	yes (42), no (43)	

The next step is to devise one or more test cases covering the remaining valid equivalence classes. One test case of the form

```
DIMENSION A 12345 (I,9,J4XXXX,65535,1,KLM,
X,1000, BBB(-65534:100,0:1000,10:10, I:65535)
```

covers the remaining classes. The invalid input equivalence classes, and a test case representing each, are:

```
  (3):  DIMENSION
  (5):  DIMENSION (10)
  (6):  DIMENSION A234567(2)
  (9):  DIMENSION A.1(2)
 (11):  DIMENSION 1A(10)
 (13):  DIMENSION B
 (14):  DIMENSION B(4,4,4,4,4,4,4,4)
 (17):  DIMENSION B(4,A(2))
 (18):  DIMENSION B(4,,7)
 (21):  DIMENSION C(I.,10)
 (23):  DIMENSION C(10,1J)
 (25):  DIMENSION D(-65535:1)
 (26):  DIMENSION D(65536)
 (31):  DIMENSION D(4:3)
 (37):  DIMENSION D(A(2):4)
 (38):  D(.:4)
 (43):  DIMENSION D(0)
```

Hence, the equivalence classes have been covered by 17 test cases. You may want to consider how these test cases would compare to a set of test cases derived in an ad hoc manner.

Although equivalence partitioning is vastly superior to a random selection of test cases, it still has deficiencies. It overlooks certain types of high-yield test cases, for example. The next two methodologies, boundary value analysis and cause-effect graphing, cover many of these deficiencies.

Boundary Value Analysis

Experience shows that test cases that explore *boundary conditions* have a higher payoff than test cases that do not. Boundary conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes. Boundary value analysis differs from equivalence partitioning in two respects:

1. Rather than selecting any element in an equivalence class as being representative, boundary value analysis requires that one or more elements be selected such that each edge of the equivalence class is the subject of a test.
2. Rather than just focusing attention on the input conditions (input space), test cases are also derived by considering the *result space* (output equivalence classes).

It is difficult to present a "cookbook" for boundary value analysis, since it requires a degree of creativity and a certain amount of specialization toward the problem at hand. (Hence, like many other aspects of testing, it is more a state of mind than anything else.) However, a few general guidelines are in order:

1. If an input condition specifies a range of values, write test cases for the ends of the range, and invalid-input test cases for situations just beyond the ends. For instance, if the valid domain of an input value is –1.0 to 1.0, write test cases for the situations –1.0, 1.0, –1.001, and 1.001.
2. If an input condition specifies a number of values, write test cases for the minimum and maximum number of values and one beneath and beyond these values. For instance, if an input file can contain 1–255 records, write test cases for 0, 1, 255, and 256 records.
3. Use guideline 1 for each output condition. For instance, if a payroll program computes the monthly FICA deduction, and if the minimum is $0.00 and the maximum is $1,165.25, write test cases that cause $0.00 and $1,165.25 to be deducted. Also, see whether it is possible to invent test cases that might cause a negative deduction or a deduction of more than $1,165.25.

   Note that it is important to examine the boundaries of the result space because it is not always the case that the boundaries of the input domains represent the same set of circumstances as the boundaries of the output ranges (e.g., consider a sine subroutine). Also, it is not always possible to generate a result outside of the output range; nonetheless, it is worth considering the possibility.
4. Use guideline 2 for each output condition. If an information retrieval system displays the most relevant abstracts based on an input request, but never more than four abstracts, write test cases such that the program displays zero, one, and four abstracts, and write a test case that might cause the program to erroneously display five abstracts.
5. If the input or output of a program is an ordered set (a sequential file, for example, or a linear list or a table), focus attention on the first and last elements of the set.
6. In addition, use your ingenuity to search for other boundary conditions.

The triangle analysis program of Chapter 1 can illustrate the need for boundary value analysis. For the input values to represent a triangle, they must be integers greater than 0 where the sum of any two is greater than the third. If you were defining equivalent partitions, you might define one where this condition is met and another where the sum of two of the integers is not greater than the third. Hence, two possible test cases might be 3–4–5 and 1–2–4. However, we have missed a likely error. That is, if an expression in the program were coded as `A+B>=C` instead of `A+B>C`, the program would erroneously tell us that 1–2–3 represents a valid scalene triangle. Hence, the important difference between boundary value analysis and equivalence partitioning is that boundary value analysis explores situations *on and around the edges of the equivalence partitions*.

As an example of a boundary value analysis, consider the following program specification:

> MTEST is a program that grades multiple-choice examinations. The input is a data file named OCR, with multiple records that are 80 characters long. Per the file specification, the first record is a title used as a title on each output report. The next set of records describes the correct answers on the exam. These records contain a "2" as the last character in column 80. In the first record of this set, the number of questions is listed in columns 1–3 (a value of 1–999). Columns 10–59 contain the correct answers for questions 1–50 (any character is valid as an answer). Subsequent records contain, in columns 10–59, the correct answers for questions 51–100, 101–150, and so on.
>
> The third set of records describes the answers of each student; each of these records contains a "3" in column 80. For each student, the first record contains the student's name or number in columns 1–9 (any characters); columns 10–59 contain the student's answers for questions 1–50. If the test has more than 50 questions, subsequent records for the student contain answers 51–100, 101–150, and so on, in columns 10–59. The maximum number of students is 200. The input data are illustrated in Figure 4.4. The four output records are:
>
> 1. A report, sorted by student identifier, showing each student's grade (percentage of answers correct) and rank.
> 2. A similar report, but sorted by grade.

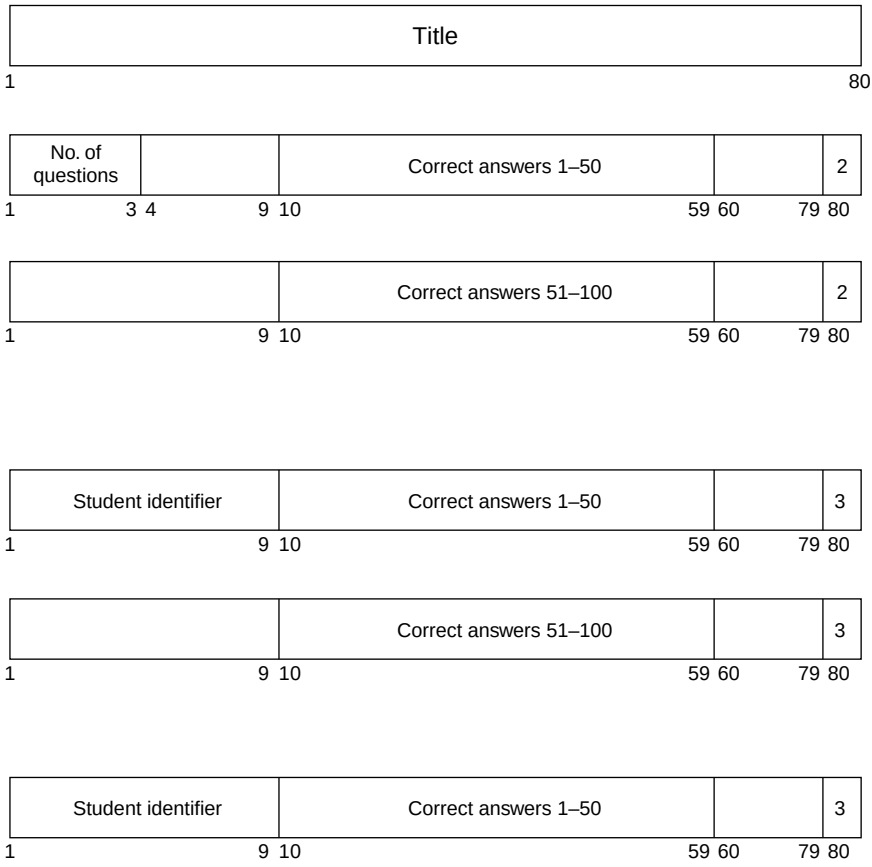

FIGURE 4.4 Input to the MTEST Program.

3. A report indicating the mean, median, and standard deviation of the grades.
4. A report, ordered by question number, showing the percentage of students answering each question correctly.

We can begin by methodically reading the specification, looking for input conditions. The first boundary input condition is an empty input file. The second input condition is the title record; boundary conditions are a missing title record and the shortest and longest possible titles. The next input conditions are the presence of correct-answer records and the number-of-questions field on the first answer record. The equivalence class

for the number of questions is not 1–999, because something special happens at each multiple of 50 (i.e., multiple records are needed). A reasonable partitioning of this into equivalence classes is 1–50 and 51–999. Hence, we need test cases where the number-of-questions field is set to 0, 1, 50, 51, and 999. This covers most of the boundary conditions for the number of correct-answer records; however, three more interesting situations are the absence of answer records and having one too many and one too few answer records (e.g., the number of questions is 60, but there are three answer records in one case and one answer record in the other case). The unique test cases identified so far are:

1. Empty input file
2. Missing title record
3. 1-character title
4. 80-character title
5. 1-question exam
6. 50-question exam
7. 51-question exam
8. 999-question exam
9. 0-question exam
10. Number-of-questions field with nonnumeric value
11. No correct-answer records after title record
12. One too many correct-answer records
13. One too few correct-answer records

The next input conditions are related to the students' answers. The boundary value test cases here appear to be:

14. 0 students
15. 1 student
16. 200 students
17. 201 students
18. A student has one answer record, but there are two correct-answer records.
19. The above student is the first student in the file.
20. The above student is the last student in the file.
21. A student has two answer records, but there is just one correct-answer record.

22. The above student is the first student in the file.
23. The above student is the last student in the file.

You also can derive a useful set of test cases by examining the output boundaries, although some of the output boundaries (e.g., empty report 1) are covered by the existing test cases. The boundary conditions of reports 1 and 2 are:

0 students (same as test 14)
1 student (same as test 15)
200 students (same as test 16)

24. All students receive the same grade.
25. All students receive a different grade.
26. Some, but not all, students receive the same grade (to see if ranks are computed correctly).
27. A student receives a grade of 0.
28. A student receives a grade of 10.
29. A student has the lowest possible identifier value (to check the sort).
30. A student has the highest possible identifier value.
31. The number of students is such that the report is just large enough to fit on one page (to see if an extraneous page is printed).
32. The number of students is such that all students but one fit on one page.

The boundary conditions from report 3 (mean, median, and standard deviation) are:

33. The mean is at its maximum (all students have a perfect score).
34. The mean is 0 (all students receive a grade of 0).
35. The standard deviation is at its maximum (one student receives a 0 and the other receives a 100).
36. The standard deviation is 0 (all students receive the same grade).

Tests 33 and 34 also cover the boundaries of the median. Another useful test case is the situation where there are 0 students (looking for a division by 0 in computing the mean), but this is identical to test case 14.

An examination of report 4 yields the following boundary value tests:

37. All students answer question 1 correctly.
38. All students answer question 1 incorrectly.
39. All students answer the last question correctly.
40. All students answer the last question incorrectly.
41. The number of questions is such that the report is just large enough to fit on one page.
42. The number of questions is such that all questions but one fit on one page.

An experienced programmer would probably agree at this point that many of these 42 test cases represent common errors that might have been made in developing this program, yet most of these errors probably would go undetected if a random or ad hoc test-case generation method were used. Boundary value analysis, if practiced correctly, is one of the most useful test-case design methods. However, it often is used ineffectively because the technique, on the surface, sounds simple. You should understand that boundary conditions may be very subtle and, hence, identification of them requires a lot of thought.

Cause-Effect Graphing

One weakness of boundary value analysis and equivalence partitioning is that they do not explore *combinations* of input circumstances. For instance, perhaps the MTEST program of the previous section fails when the product of the number of questions and the number of students exceeds some limit (the program runs out of memory, for example). Boundary value testing would not necessarily detect such an error.

The testing of input combinations is not a simple task because even if you equivalence-partition the input conditions, the number of combinations usually is astronomical. If you have no systematic way of selecting a subset of input conditions, you'll probably select an arbitrary subset of conditions, which could lead to an ineffective test.

Cause-effect graphing aids in selecting, in a systematic way, a high-yield set of test cases. It has a beneficial side effect in pointing out incompleteness and ambiguities in the specification.

A cause-effect graph is a formal language into which a natural-language specification is translated. The graph actually is a digital logic circuit (a combinatorial logic network), but instead of standard electronics notation, a somewhat simpler notation is used. No knowledge of electronics is necessary other than an understanding of Boolean logic (i.e., of the logic operators *and, or,* and *not*).

The following process is used to derive test cases:

1. The specification is divided into workable pieces. This is necessary because cause-effect graphing becomes unwieldy when used on large specifications. For instance, when testing an e-commerce system, a workable piece might be the specification for choosing and verifying a single item placed in a shopping cart. When testing a Web page design, you might test a single menu tree or even a less complex navigation sequence.
2. The causes and effects in the specification are identified. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition or a system transformation (a lingering effect that an input has on the state of the program or system). For instance, if a transaction causes a file or database record to be updated, the alteration is a system transformation; a confirmation message would be an output condition.

   You identify causes and effects by reading the specification word by word and underlining words or phrases that describe causes and effects. Once identified, each cause and effect is assigned a unique number.
3. The semantic content of the specification is analyzed and transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph.
4. The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.
5. By methodically tracing state conditions in the graph, you convert the graph into a limited-entry decision table. Each column in the table represents a test case.
6. The columns in the decision table are converted into test cases.

The basic notation for the graph is shown in Figure 4.5. Think of each node as having the value 0 or 1; 0 represents the "absent" state and 1 represents the "present" state.

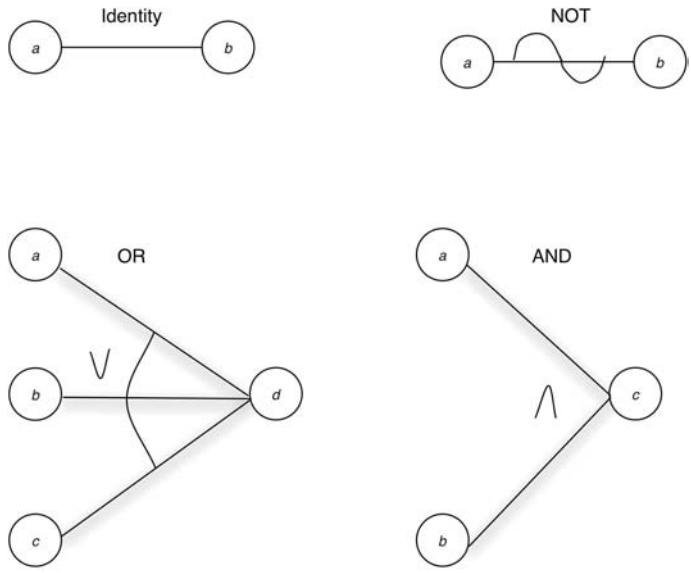

FIGURE 4.5 Basic Cause-Effect Graph Symbols.

- The *identity* function states that if *a* is 1, *b* is 1; else *b* is 0.
- The *not* function states that if *a* is 1, *b* is 0, else *b* is 1.
- The *or* function states that if *a* or *b* or *c* is 1, *d* is 1; else *d* is 0.
- The *and* function states that if both *a* and *b* are 1, *c* is 1; else *c* is 0.

The latter two functions (*or* and *and*) are allowed to have any number of inputs.

To illustrate a small graph, consider the following specification:

> The character in column 1 must be an "A" or a "B." The character in column 2 must be a digit. In this situation, the file update is made. If the first character is incorrect, message `X12` is issued. If the second character is not a digit, message `X13` is issued.

The causes are:

1—character in column 1 is "A"
2—character in column 1 is "B"
3—character in column 2 is a digit

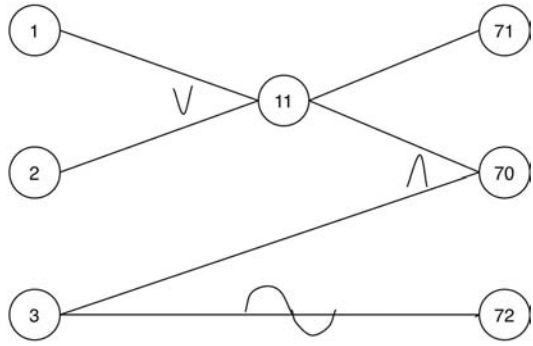

FIGURE 4.6 Sample Cause-Effect Graph.

and the effects are:

70—update made
71—message X12 is issued
72—message X13 is issued

The cause-effect graph is shown in Figure 4.6. Notice the intermediate node 11 that was created. You should confirm that the graph represents the specification by setting all possible states of the causes and verifying that the effects are set to the correct values. For readers familiar with logic diagrams, Figure 4.7 is the equivalent logic circuit.

Although the graph in Figure 4.6 represents the specification, it does contain an impossible combination of causes—it is impossible for both causes 1 and 2 to be set to 1 simultaneously. In most programs, certain combinations of causes are impossible because of syntactic or environmental considerations (a character cannot be an "A" and a "B" simultaneously).

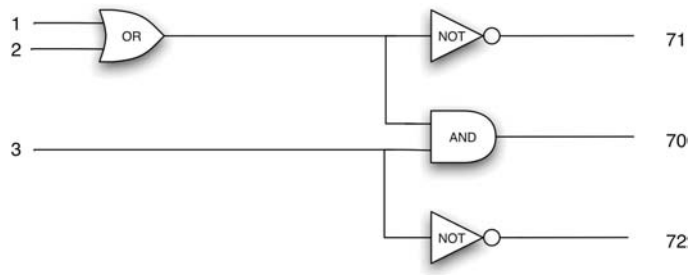

FIGURE 4.7 Logic Diagram Equivalent to Figure 4.6.

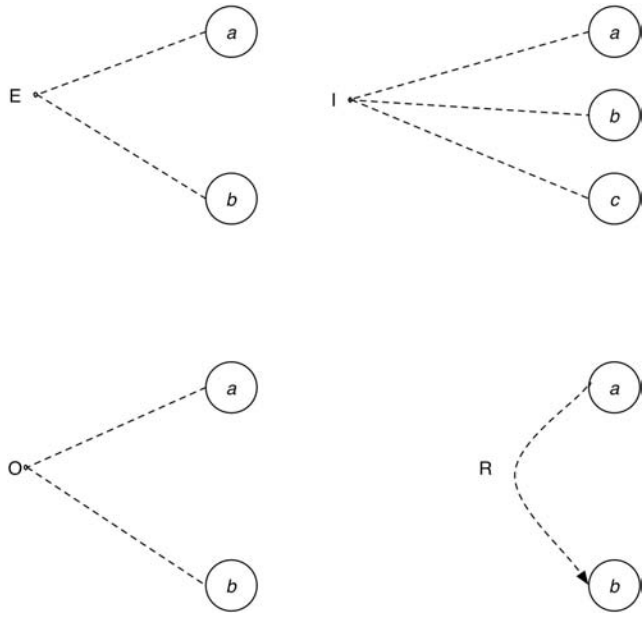

FIGURE 4.8 Constraint Symbols.

To account for these, the notation in Figure 4.8 is used. The E constraint states that it must always be true that, at most, one of *a* and *b* can be 1 (*a* and *b* cannot be 1 simultaneously). The I constraint states that at least one of *a, b,* and *c* must always be 1 (*a, b,* and *c* cannot be 0 simultaneously). The O constraint states that one, and only one, of *a* and *b* must be 1. The R constraint states that for *a* to be 1, *b* must be 1 (i.e., it is impossible for *a* to be 1 and *b* to be 0).

There frequently is a need for a constraint among effects. The `M` constraint in Figure 4.9 states that if effect *a* is 1, effect *b* is forced to 0.

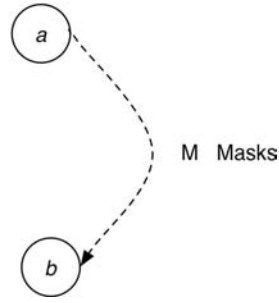

FIGURE 4.9 Symbol for "Masks" Constraint.

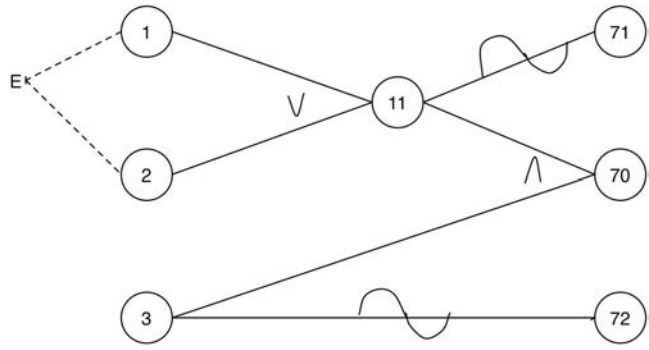

FIGURE 4.10 Sample Cause-Effect Graph with "Exclusive" Constraint.

Returning to the preceding simple example, we see that it is physically impossible for causes 1 and 2 to be present simultaneously, but it is possible for neither to be present. Hence, they are linked with the E constraint, as shown in Figure 4.10.

To illustrate how cause-effect graphing is used to derive test cases, we use the following specification for a debugging command in an interactive system.

> The *DISPLAY* command is used to view from a terminal window the contents of memory locations. The command syntax is shown in Figure 4.11. Brackets represent alternative optional operands. Capital letters represent operand keywords. Lowercase letters represent operand values (actual values are to be substituted). Underlined operands represent the default values (i.e., the value used when the operand is omitted).

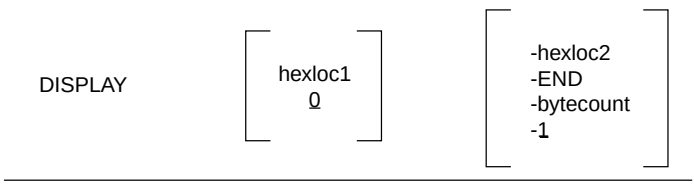

FIGURE 4.11 Syntax of the DISPLAY Command.

The first operand (*hexloc1*) specifies the address of the first byte whose contents are to be displayed. The address may be one to six hexadecimal digits (0–9, A–F) in length. If it is not specified, the address 0 is assumed. The address must be within the actual memory range of the machine.

The second operand specifies the amount of memory to be displayed. If *hexloc2* is specified, it defines the address of the last byte in the range of locations to be displayed. It may be one to six hexadecimal digits in length. The address must be greater than or equal to the starting address (*hexloc1*). Also, *hexloc2* must be within the actual memory range of the machine. If `END` is specified, memory is displayed up through the last actual byte in the machine. If `bytecount` is specified, it defines the number of bytes of memory to be displayed (starting with the location specified in *hexloc1*). The operand *bytecount* is a hexadecimal integer (one to six digits). The sum of `bytecount` and *hexloc1* must not exceed the actual memory size plus 1, and *bytecount* must have a value of at least 1.

When memory contents are displayed, the output format on the screen is one or more lines of the format

```
xxxxxx = word1 word2 word3 word4
```

where `xxxxxx` is the hexadecimal address of `word1`. An integral number of words (four-byte sequences, where the address of the first byte in the word is a multiple of 4) is always displayed, regardless of the value of *hexloc1* or the amount of memory to be displayed. All output lines will always contain four words (16 bytes). The first byte of the displayed range will fall within the first word.

The error messages that can be produced are

M1 is invalid command syntax.
M2 memory requested is beyond actual memory limit.
M3 memory requested is a zero or negative range.

As examples:

```
DISPLAY
```

displays the first four words in memory (default starting address of 0, default byte count of 1);

```
DISPLAY  77F
```

displays the word containing the byte at address 77F, and the three subsequent words;

```
DISPLAY  77F-407A
```

displays the words containing the bytes in the address range 775–407A;

```
DISPLAY  77F.6
```

displays the words containing the six bytes starting at location 77F; and

```
DISPLAY  50FF-END
```

displays the words containing the bytes in the address range 50FF to the end of memory.

The first step is a careful analysis of the specification to identify the causes and effects. The causes are as follows:

1. First operand is present.
2. The *hexloc1* operand contains only hexadecimal digits.
3. The *hexloc1* operand contains one to six characters.
4. The *hexloc1* operand is within the actual memory range of the machine.
5. Second operand is *END*.
6. Second operand is *hexloc*.
7. Second operand is *bytecount*.
8. Second operand is omitted.
9. The *hexloc2* operand contains only hexadecimal digits.
10. The *hexloc2* operand contains one to six characters.
11. The *hexloc2* operand is within the actual memory range of the machine.
12. The *hexloc2* operand is greater than or equal to the *hexloc1* operand.
13. The *bytecount* operand contains only hexadecimal digits.
14. The *bytecount* operand contains one to six characters.

15. *bytecount* + *hexloc1* <= memory size + 1.
16. *bytecount* >= 1.
17. Specified range is large enough to require multiple output lines.
18. Start of range does not fall on a word boundary.

Each cause has been given an arbitrary unique number. Notice that four causes (5 through 8) are necessary for the second operand because the second operand could be (1) ***END***, (2) *hexloc2,* (3) *byte-count,* (4) absent, and (5) none of the above. The effects are as follows:

91. Message `M1` is displayed.
92. Message `M2` is displayed.
93. Message `M3` is displayed.
94. Memory is displayed on one line.
95. Memory is displayed on multiple lines.
96. First byte of displayed range falls on a word boundary.
97. First byte of displayed range does not fall on a word boundary.

The next step is the development of the graph. The cause nodes are listed vertically on the left side of the sheet of paper; the effect nodes are listed vertically on the right side. The semantic content of the specification is carefully analyzed to interconnect the causes and effects (i.e., to show under what conditions an effect is present).

Figure 4.12 shows an initial version of the graph. Intermediate node 32 represents a syntactically valid first operand; node 35 represents a syntactically valid second operand. Node 36 represents a syntactically valid command. If node 36 is 1, effect 91 (the error message) does not appear. If node 36 is 0, effect 91 is present.

The full graph is shown in Figure 4.13. You should explore it carefully to convince yourself that it accurately reflects the specification.

If Figure 4.13 were used to derive the test cases, many impossible-to-create test cases would be derived. The reason is that certain combinations of causes are impossible because of syntactic constraints. For instance, causes 2 and 3 cannot be present unless cause 1 is present. Cause 4 cannot be present unless both causes 2 and 3 are present. Figure 4.14 contains the complete graph with the constraint conditions. Notice that, at most, one of the causes 5, 6, 7, and 8 can be present. All other cause constraints are the *requires* condition. Notice that cause 17 (multiple output lines) requires the *not* of cause 8

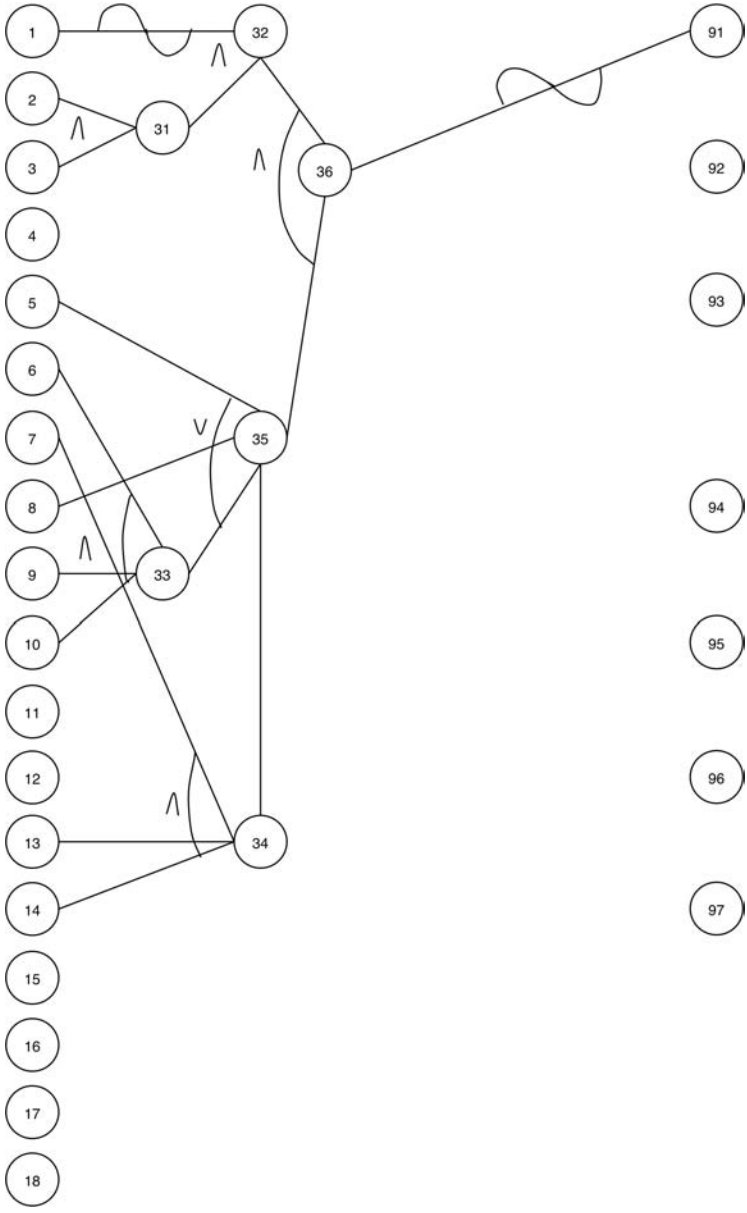

FIGURE 4.12 Beginning of the Graph for the DISPLAY Command.

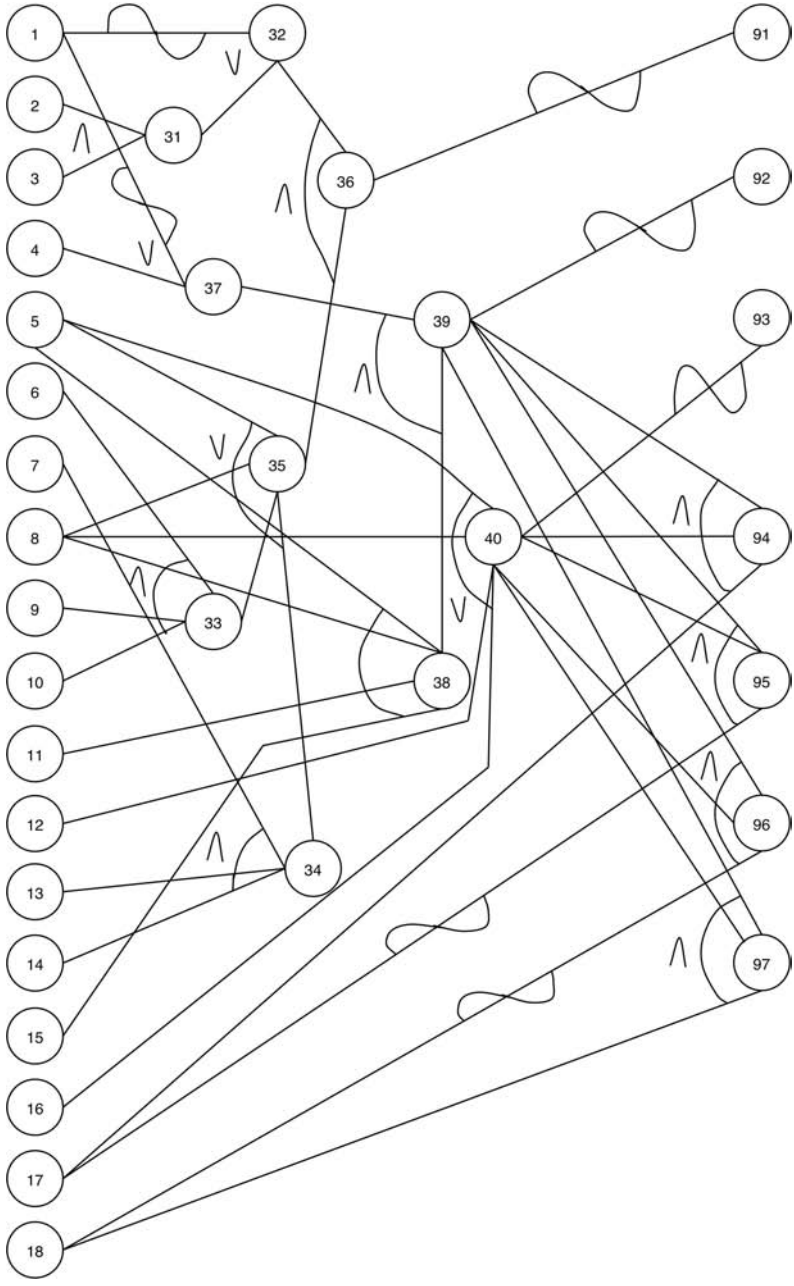

FIGURE 4.13 Full Cause-Effect Graph without Constraints.

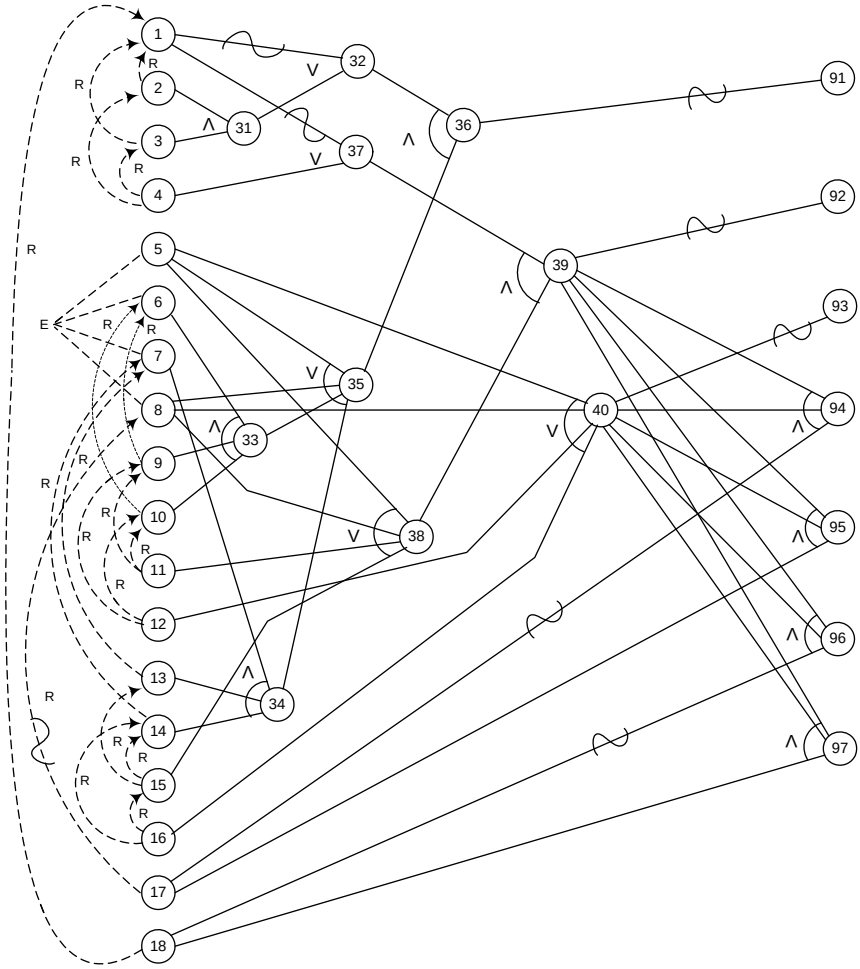

FIGURE 4.14 Complete Cause-Effect Graph of the DISPLAY Command.

(second operand is omitted); cause 17 can be present only when cause 8 is absent. Again, you should explore the constraint conditions carefully.

The next step is the generation of a limited-entry decision table. For readers familiar with decision tables, the causes are the conditions and the effects are the actions. The procedure used is as follows:

1. Select an effect to be the present (1) state.
2. Tracing back through the graph, find all combinations of causes (subject to the constraints) that will set this effect to 1.
3. Create a column in the decision table for each combination of causes.

4. For each combination, determine the states of all other effects and place these in each column.

In performing step 2, the considerations are as follows:

1. When tracing back through an *or* node whose output should be 1, never set more than one input to the *or* to 1 simultaneously. This is called *path sensitizing*. Its objective is to prevent the failure to detect certain errors because of one cause masking another cause.
2. When tracing back through an *and* node whose output should be 0, all combinations of inputs leading to 0 output must, of course, be enumerated. However, if you are exploring the situation where one input is 0 and one or more of the others are 1, it is not necessary to enumerate all conditions under which the other inputs can be 1.
3. When tracing back through an *and* node whose output should be 0, only one condition where all inputs are zero need be enumerated. (If the *and* is in the middle of the graph such that its inputs come from other intermediate nodes, there may be an excessively large number of situations under which all of its inputs are 0.)

These complicated considerations are summarized in Figure 4.15, and Figure 4.16 is used as an example.

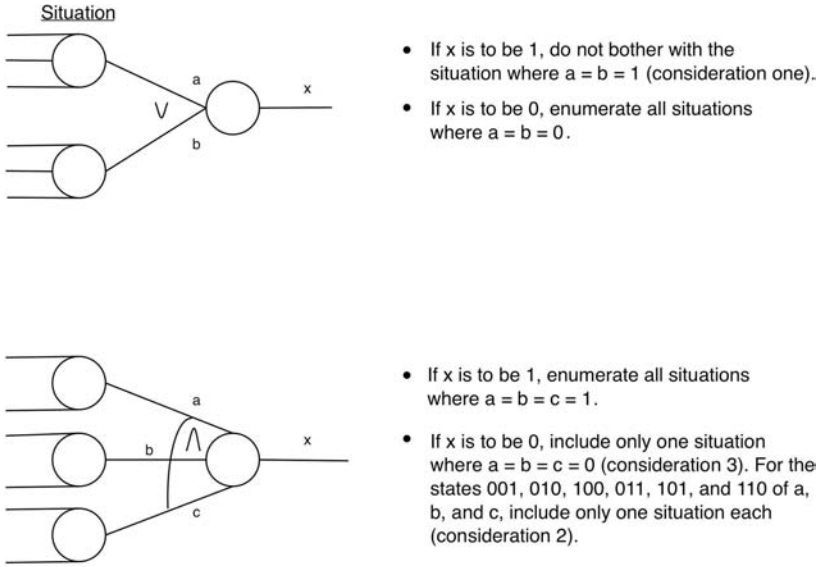

FIGURE 4.15 Considerations Used When Tracing the Graph.

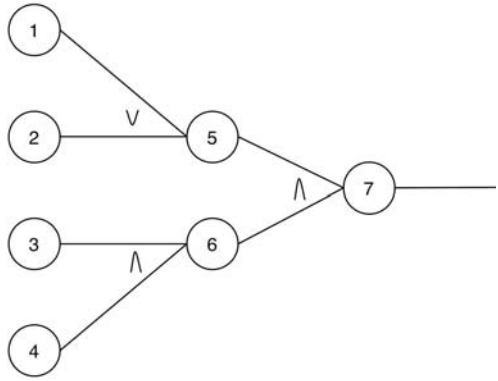

FIGURE 4.16 Sample Graph to Illustrate the Tracing Considerations.

Assume that we want to locate all input conditions that cause the output state to be 0. Consideration 3 states that we should list only one circumstance where nodes 5 and 6 are 0. Consideration 2 states that for the state where node 5 is 1 and node 6 is 0, we should list only one circumstance where node 5 is 1, rather than enumerating all possible ways that node 5 can be 1. Likewise, for the state where node 5 is 0 and node 6 is 1, we should list only one circumstance where node 6 is 1 (although there is only one in this example). Consideration 1 states that where node 5 should be set to 1, we should not set nodes 1 and 2 to 1 simultaneously. Hence, we would arrive at five states of nodes 1 through 4; for example, the values:

0	0	0	0	(5=0, 6=0)
1	0	0	0	(5=1, 6=0)
1	0	0	1	(5=1, 6=0)
1	0	1	0	(5=1, 6=0)
0	0	1	1	(5=0, 6=1)

rather than the 13 possible states of nodes 1 through 4 that lead to a 0 output state.

These considerations may appear to be capricious, but they have an important purpose: to lessen the combined effects of the graph. They eliminate situations that tend to be low-yield test cases. If low-yield test cases are not eliminated, a large cause-effect graph will produce an astronomical number of test cases. If the number of test cases is too large to be practical,

you will select some subset, but there is no guarantee that the low-yield test cases will be the ones eliminated. Hence, it is better to eliminate them during the analysis of the graph.

We will now convert the cause-effect graph in Figure 4.14 into the decision table. Effect 91 will be selected first. Effect 91 is present if node 36 is 0. Node 36 is 0 if nodes 32 and 35 are 0,0; 0,1; or 1,0; and considerations 2 and 3 apply here. By tracing back to the causes, and considering the constraints among causes, you can find the combinations of causes that lead to effect 91 being present, although doing so is a laborious process.

The resultant decision table, under the condition that effect 91 is present, is shown in Figure 4.17 (columns 1 through 11). Columns (tests) 1 through 3 represent the conditions where node 32 is 0 and node 35 is 1. Columns 4 through 10 represent the conditions where node 32 is 1 and node 35 is 0. Using consideration 3, only one situation (column 11) out of a possible 21 situations where nodes 32 and 35 are 0 is identified. Blanks in the table represent "don't care" situations (i.e., the state of the cause is irrelevant) or indicate that the state of a cause is obvious because of the states of other dependent causes (e.g., in column 1, we know that causes 5, 7, and 8 must be 0 because they exist in an "at most one" situation with cause 6).

Columns 12 through 15 represent the situations where effect 92 is present. Columns 16 and 17 represent the situations where effect 93 is present. Figure 4.18 represents the remainder of the decision table.

The last step is to convert the decision table into 38 test cases. A set of 38 test cases is listed here. The number or numbers beside each test case designate the effects that are expected to be present. Assume that the last location in memory on the machine being used is 7FFF.

1	`DISPLAY 234AF74–123`	(91)
2	`DISPLAY 2ZX4–3000`	(91)
3	`DISPLAY HHHHHHHH-2000`	(91)
4	`DISPLAY 200 200`	(91)
5	`DISPLAY 0–22222222`	(91)
6	`DISPLAY 1–2X`	(91)
7	`DISPLAY 2-ABCDEFGHI`	(91)
8	`DISPLAY 3.1111111`	(91)
9	`DISPLAY 44.$42`	(91)
10	`DISPLAY 100.$$$$$$$`	(91)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	0	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1
4												1	1	0	0	1	1
5				0										1			
6	1	1	1	0	1	1	1				1	1			1	1	
7				0				1	1	1			1				1
8				0													
9	1	1	1		1	0	0				0	1			1	1	
10	1	1	1		0	1	0				1	1			1	1	
11												0			0	1	
12																0	
13								1	0	0			1				1
14								0	1	0			1				1
15													0				
16																	0
17																	
18																	
91	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
94	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
96	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
97	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FIGURE 4.17 First Half of the Resultant Decision Table.

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
1	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	0	0	0	1	1	1
2	1	1	1	1					1	1	1	1	1	1	1				1	1	1
3	1	1	1	1					1	1	1	1	1	1	1				1	1	1
4	1	1	1	1					1	1	1	1	1	1	1				1	1	1
5	1				1				1				1			1			1		
6			1				1				1			1			1			1	
7				1				1				1			1			1			1
8		1				1				1											
9			1				1				1			1			1			1	
10			1				1				1			1			1			1	
11			1				1				1			1			1			1	
12			1				1				1			1			1			1	
13				1				1				1			1			1			1
14				1				1				1			1			1			1
15				1				1				1			1			1			1
16				1				1				1			1			1			1
17	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
18	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
91	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
92	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
93	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
94	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
95	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
96	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	1
97	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

FIGURE 4.18 Second Half of the Resultant Decision Table.

```
11      DISPLAY 10000000-M          (91)
12      DISPLAY FF-8000             (92)
13      DISPLAY FFF.7001            (92)
14      DISPLAY 8000-END            (92)
15      DISPLAY 8000–8001           (92)
16      DISPLAY AA-A9               (93)
17      DISPLAY 7000.0              (93)
18      DISPLAY 7FF9-END            (94, 97)
```

19	`DISPLAY 1`	(94, 97)
20	`DISPLAY 21–29`	(94, 97)
21	`DISPLAY 4021.A`	(94, 97)
22	`DISPLAY -END`	(94, 96)
23	`DISPLAY`	(94, 96)
24	`DISPLAY -F`	(94, 96)
25	`DISPLAY .E`	(94, 96)
26	`DISPLAY 7FF8-END`	(94, 96)
27	`DISPLAY 6000`	(94, 96)
28	`DISPLAY A0-A4`	(94, 96)
29	`DISPLAY 20.8`	(94, 96)
30	`DISPLAY 7001-END`	(95, 97)
31	`DISPLAY 5–15`	(95, 97)
32w	`DISPLAY 4FF.100`	(95, 97)
33	`DISPLAY -END`	(95, 96)
34	`DISPLAY -20`	(95, 96)
35	`DISPLAY .11`	(95, 96)
36	`DISPLAY 7000-END`	(95, 96)
37	`DISPLAY 4–14`	(95, 96)
38	`DISPLAY 500.11`	(95, 96)

Note that where two or more different test cases invoked, for the most part, the same set of causes, different values for the causes were selected to slightly improve the yield of the test cases. Also note that, because of the actual storage size, test case 22 is impossible (it will yield effect 95 instead of 94, as noted in test case 33). Hence, 37 test cases have been identified.

Remarks Cause-effect graphing is a systematic method of generating test cases representing combinations of conditions. The alternative would be to make an ad hoc selection of combinations; but in doing so, it is likely that you would overlook many of the "interesting" test cases identified by the cause-effect graph.

Since cause-effect graphing requires the translation of a specification into a Boolean logic network, it gives you a different perspective on, and additional insight into, the specification. In fact, the development of a cause-

effect graph is a good way to uncover ambiguities and incompleteness in specifications. For instance, the astute reader may have noticed that this process has uncovered a problem in the specification of the *DISPLAY* command. The specification states that all output lines contain four words. This cannot be true in all cases; it cannot occur for test cases 18 and 26 because the starting address is less than 16 bytes away from the end of memory.

Although cause-effect graphing does produce a set of useful test cases, it normally does not produce *all* of the useful test cases that might be identified. For instance, in the example we said nothing about verifying that the displayed memory values are identical to the values in memory and determining whether the program can display every possible value in a memory location. Also, the cause-effect graph does not adequately explore boundary conditions. Of course, you could attempt to cover boundary conditions during the process. For instance, instead of identifying the single cause

```
hexloc2>=hexloc1
```

you could identify two causes:

hexloc2 = hexloc1
hexloc2 > hexloc1

The problem in doing this, however, is that it complicates the graph tremendously and leads to an excessively large number of test cases. For this reason it is best to consider a separate boundary value analysis. For instance, the following boundary conditions can be identified for the *DISPLAY* specification:

1. *hexloc1* has one digit
2. *hexloc1* has six digits
3. *hexloc1* has seven digits
4. *hexloc1* = 0
5. *hexloc1* = 7FFF
6. *hexloc1* = 8000
7. *hexloc2* has one digit
8. *hexloc2* has six digits
9. *hexloc2* has seven digits
10. *hexloc2* = 0

11. *hexloc2* = 7FFF
12. *hexloc2* = 8000
13. *hexloc2* = *hexloc*
14. *hexloc2* = *hexloc1* + 1
15. *hexloc2* = *hexloc1* − 1
16. *bytecount* has one digit
17. *bytecount* has six digits
18. *bytecount* has seven digits
19. *bytecount* = 1
20. *hexloc1* + *bytecount* = 8000
21. *hexloc1* + *bytecount* = 8001
22. *display* 16 bytes (one line)
23. *display* 17 bytes (two lines)

Note that this does not imply that you would write 60 (37 + 23) test cases. Since the cause-effect graph gives us leeway in selecting specific values for operands, the boundary conditions could be blended into the test cases derived from the cause-effect graph. In this example, by rewriting some of the original 37 test cases, all 23 boundary conditions could be covered without any additional test cases. Thus, we arrive at a small but potent set of test cases that satisfy both objectives.

Note that cause-effect graphing is consistent with several of the testing principles in Chapter 2. Identifying the expected output of each test case is an inherent part of the technique (each column in the decision table indicates the expected effects). Also note that it encourages us to look for unwanted side effects. For instance, column (test) 1 specifies that we should expect effect 91 to be present and that effects 92 through 97 should be absent.

The most difficult aspect of the technique is the conversion of the graph into the decision table. This process is algorithmic, implying that you could automate it by writing a program; several commercial programs exist to help with the conversion.

Error Guessing

It has often been noted that some people seem to be naturally adept at program testing. Without using any particular methodology such as boundary

value analysis of cause-effect graphing, these people seem to have a knack for sniffing out errors.

One explanation for this is that these people are practicing—subconsciously more often than not—a test-case design technique that could be termed *error guessing*. Given a particular program, they surmise—both by intuition and experience—certain probable types of errors and then write test cases to expose those errors.

It is difficult to give a procedure for the error-guessing technique since it is largely an intuitive and ad hoc process. The basic idea is to enumerate a list of possible errors or error-prone situations and then write test cases based on the list. For instance, the presence of the value 0 in a program's input is an error-prone situation. Therefore, you might write test cases for which particular input values have a 0 value and for which particular output values are forced to 0. Also, where a variable number of inputs or outputs can be present (e.g., the number of entries in a list to be searched), the cases of "none" and "one" (e.g., empty list, list containing just one entry) are error-prone situations. Another idea is to identify test cases associated with assumptions that the programmer might have made when reading the specification (i.e., factors that were omitted from the specification, either by accident or because the writer felt them to be obvious).

Since a procedure for error guessing cannot be given, the next-best alternative is to discuss the spirit of the practice, and the best way to do this is by presenting examples. If you are testing a sorting subroutine, the following are situations to explore:

- The input list is empty.
- The input list contains one entry.
- All entries in the input list have the same value.
- The input list is already sorted.

In other words, you enumerate those special cases that may have been overlooked when the program was designed. If you are testing a binary search subroutine, you might try the situations where: (1) there is only one entry in the table being searched; (2) the table size is a power of 2 (e.g., 16); and (3) the table size is one less than and one greater than a power of 2 (e.g., 15 or 17).

Consider the MTEST program in the section on boundary value analysis. The following additional tests come to mind when using the error-guessing technique:

- Does the program accept "blank" as an answer?
- A type-2 (answer) record appears in the set of type-3 (student) records.
- A record without a 2 or 3 in the last column appears as other than the initial (title) record.
- Two students have the same name or number.
- Since a median is computed differently depending on whether there is an odd or an even number of items, test the program for an even number of students and an odd number of students.
- The number-of-questions field has a negative value.

Error-guessing tests that come to mind for the *DISPLAY* command of the previous section are as follows:

`DISPLAY 100-` (partial second operand)
`DISPLAY 100.` (partial second operand)
`DISPLAY 100–10A 42` (extra operand)
`DISPLAY 000–0000FF` (leading zeros)

The Strategy

The test-case design methodologies discussed in this chapter can be combined into an overall strategy. The reason for combining them should be obvious by now: Each contributes a particular set of useful test cases, but none of them by itself contributes a thorough set of test cases. A reasonable strategy is as follows:

1. If the specification contains combinations of input conditions, start with cause-effect graphing.
2. In any event, use boundary value analysis. Remember that this is an analysis of input and output boundaries. The boundary value analysis yields a set of supplemental test conditions, but as noted in the section on cause-effect graphing, many or all of these can be incorporated into the cause-effect tests.

3. Identify the valid and invalid equivalence classes for the input and output, and supplement the test cases identified above, if necessary.
4. Use the error-guessing technique to add additional test cases.
5. Examine the program's logic with regard to the set of test cases. Use the decision coverage, condition coverage, decision/condition coverage, or multiple-condition coverage criterion (the last being the most complete). If the coverage criterion has not been met by the test cases identified in the prior four steps, and if meeting the criterion is not impossible (i.e., certain combinations of conditions may be impossible to create because of the nature of the program), add sufficient test cases to cause the criterion to be satisfied.

Again, the use of this strategy will not guarantee that all errors will be found, but it has been found to represent a reasonable compromise. Also, it represents a considerable amount of hard work, but as we said at the beginning of this chapter, no one has ever claimed that program testing is easy.

Summary

Once you have agreed that aggressive software testing is a worthy addition to your development efforts, the next step is to design test cases that will exercise your application sufficiently to produce satisfactory test results. In most cases, consider a combination of black-box and white-box methodologies to ensure that you have designed rigorous program testing.

Test case design techniques discussed in this chapter include:

- *Logic coverage.* Tests that exercise all decision point outcomes at least once, and ensure that all statements or entry points are executed at least once.
- *Equivalence partitioning.* Defines condition or error classes to help reduce the number of finite tests. Assumes that a test of a representative value within a class also tests all values or conditions within that class.
- *Boundary value analysis.* Tests each edge condition of an equivalence class; also considers output equivalence classes as well as input classes.
- *Cause-effect graphing.* Produces Boolean graphical representations of potential test case results to aid in selecting efficient and complete test cases.

- *Error guessing*. Produces test cases based on intuitive and expert knowledge of test team members to define potential software errors to facilitate efficient test case design.

Extensive, in-depth testing is not easy; nor will the most extensive test case design assure that every error will be uncovered. That said, developers willing to go beyond cursory testing, who will dedicate sufficient time to test case design, analyze carefully the test results, and act decisively on the findings, will be rewarded with functional, reliable software that is reasonably error free.

5 Module (Unit) Testing

Up to this point we have largely ignored the mechanics of testing and the size of the program being tested. However, because large programs (say, of 500 statements or 50-plus classes) require special testing treatment, in this chapter we consider an initial step in structuring the testing of a large program: module testing. Chapters 6 and 7 enumerate the remaining steps.

Module testing (or unit testing) is a process of testing the individual subprograms, subroutines, classes, or procedures in a program. More specifically, rather than initially testing the program as a whole, testing is first focused on the smaller building blocks of the program. The motivations for doing this are threefold. First, module testing is a way of managing the combined elements of testing, since attention is focused initially on smaller units of the program. Second, module testing eases the task of debugging (the process of pinpointing and correcting a discovered error), since, when an error is found, it is known to exist in a particular module. Finally, module testing introduces parallelism into the program testing process by presenting us with the opportunity to test multiple modules simultaneously.

The purpose of module testing is to compare the function of a module to some functional or interface specification defining the module. To reemphasize the goal of all testing processes, the objective here is not to show that the module meets its specification, but that the module contradicts the specification. In this chapter, we address module testing from three points of view:

1. The manner in which test cases are designed.
2. The order in which modules should be tested and integrated.
3. Advice about performing the tests.

Test-Case Design

You need two types of information when designing test cases for a module test: a specification for the module and the module's source code. The specification typically defines the module's input and output parameters and its function.

Module testing is largely white-box oriented. One reason is that as you test larger entities, such as entire programs (which will be the case for subsequent testing processes), white-box testing becomes less feasible. A second reason is that the subsequent testing processes are oriented toward finding different types of errors (e.g., errors not necessarily associated with the program's logic, such as the program failing to meet its users' requirements). Hence, the test-case design procedure for a module test is the following:

> Analyze the module's logic using one or more of the white-box methods, and then supplement these test cases by applying black-box methods to the module's specification.

The test-case design methods we will use were defined in Chapter 4; we will illustrate their use in a module test here through an example.

Assume that we wish to test a module named *BONUS*, and its function is to add $2,000 to the salary of all employees in the department or departments having the largest sales revenue. However, if an eligible employee's current salary is $150,000 or more, or if the employee is a manager, the salary is to be increased by only $1,000.

The inputs to the module are shown in the tables in Figure 5.1. If the module performs its function correctly, it returns an error code of 0. If either the employee or the department table contains no entries, it returns an error code of 1. If it finds no employees in an eligible department, it returns an error code of 2.

The module's source code is shown in Figure 5.2. Input parameters `ESIZE` and `DSIZE` contain the number of entries in the employee and department tables. Note that though the module is written in `PL/1`, the following discussion is largely language independent; the techniques are applicable to programs coded in other languages. Also, because the `PL/1` logic in the module is fairly simple, virtually any reader, even those not familiar with `PL/1`, should be able to understand it.

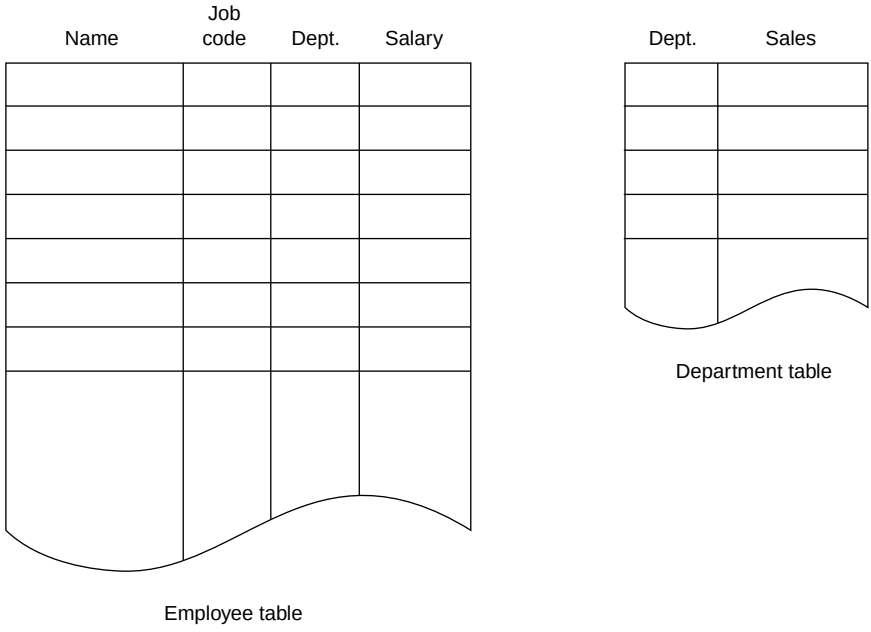

FIGURE 5.1 Input Tables to Module BONUS.

```
BONUS : PROCEDURE(EMPTAB,DEPTTAB,ESIZE,DSIZE,ERRCODE);
DECLARE  1  EMPTAB (*),
            2 NAME CHAR(6),
            2 CODE CHAR(1),
            2 DEPT CHAR(3),
            2 SALARY FIXED DECIMAL(7,2);
DECLARE  1  DEPTTAB (*),
            2 DEPT CHAR(3),
            2 SALES FIXED DECIMAL(8,2);
DECLARE  (ESIZE,DSIZE) FIXED BINARY;
DECLARE  ERRCODE FIXED DECIMAL(1);
DECLARE  MAXSALES FIXED DECIMAL(8,2) INIT(0); /*MAX. SALES IN DEPTTAB*/
DECLARE  (I,J,K) FIXED BINARY;             /*COUNTERS*/
DECLARE  FOUND BIT(1);  /*TRUE IF ELIGIBLE DEPT. HAS EMPLOYEES*/
DECLARE  SINC FIXED DECIMAL(7,2) INIT(200.00);  /*STANDARD INCREMENT*/
DECLARE  LINC FIXED DECIMAL(7,2) INIT(100.00);  /*LOWER INCREMENT*/
DECLARE  LSALARY FIXED DECIMAL(7,2) INIT(15000.00);   /*SALARY BOUNDARY*/
DECLARE  MGR CHAR(1) INIT('M');
```

(continued)

FIGURE 5.2 Module BONUS.

```
1  ERRCODE=0;
2  IF(ESIZE<=0)|(DSIZE<=0)
3     THEN ERRCODE=1;                  /*EMPTAB OR DEPTTAB ARE EMPTY*/
4     ELSE DO;
5        DO I = 1 TO DSIZE;            /*FIND MAXSALES AND MAXDEPTS*/
6           IF(SALES(I)>=MAXSALES) THEN MAXSALES=SALES(I);
7        END;
8        DO J = 1 TO DSIZE;
9           IF(SALES(J)=MAXSALES)         /*ELIGIBLE DEPARTMENT*/
10             THEN DO;
11                FOUND='0'B;
12                DO K = 1 TO ESIZE;
13                   IF(EMPTAB.DEPT(K)=DEPTTAB.DEPT(J))
14                      THEN DO;
15                         FOUND='1'B;
16                         IF(SALARY(K)>=LSALARY)|CODE(K)=MGR)
17                            THEN SALARY(K)=SALARY(K)+LINC;
18                            ELSE SALARY(K)=SALARY(K)+SINC;
19                         END;
20                END;
21                IF(-FOUND) THEN ERRCODE=2;
22             END;
23       END;
24    END;
25 END;
```

FIGURE 5.2 *(continued)*

Sidebar 5.1: PL/1 Background

Readers new to software development may be unfamiliar with PL/1 and think of it is a "dead" language. True, there probably is very little new development using PL/1, but maintenance of existing systems continues, and the PL/1 constructs still are a pretty good way to learn about programming procedures.

PL/1, which stands for Programming Language One, was developed in the 1960s by IBM to provide an English-like development environment for its mainframe class machines, beginning with the IBM System/360. At this time in computer history, many programmers were migrating toward specialty languages such as COBOL, designed for business application development, and Fortran, designed for scientific applications. (See Sidebar 3.1 in Chapter 3 for a little background on these languages.)

One of the main goals for PL/1 designers was a development language that could compete successfully with COBOL and Fortran while providing a development environment that would be easier to learn with a more natural language. All of the early goals for PL/1 likely never were achieved, but those early designers obviously did their homework, because PL/1 has been refined and upgraded over the years and still is in use in some environments today.

By the mid-1990s PL/1 had been extended to other computer platforms, including OS/2, Linux, UNIX, and Windows. New operating system support brought language extensions to provide more flexibility and functionality.

Regardless of which of the logic coverage techniques you use, the first step is to list the conditional decisions in the program. Candidates in this program are all `IF` and `DO` statements. By inspecting the program, we can see that all of the `DO` statements are simple iterations, and each iteration limit will be equal to or greater than the initial value (meaning that each loop body always will execute at least once); and the only way of exiting each loop is via the `DO` statement. Thus, the *`DO`* statements in this program need no special attention, since any test case that causes a *`DO`* statement to execute will eventually cause it to branch in both directions (i.e., enter the loop body and skip the loop body). Therefore, the statements that must be analyzed are:

```
2 IF (ESIZE<=0) | (DSIZE<=0)
6 IF (SALES(I)>=MAXSALES)
9 IF (SALES(J)=MAXSALES)
13 IF (EMPTAB.DEPT(K)=DEPTTAB.DEPT(J))
16 IF (SALARY(K)>=LSALARY) | (CODE(K)=MGR)
21 IF(-FOUND) THEN ERRCODE=2
```

TABLE 5.1 Situations Corresponding to the Decision Outcomes

Decision	True Outcome	False Outcome
2	ESIZE or DSIZE≤0	ESIZE and DSIZE>0
6	Will always occur at least once.	Order DEPTTAB so that a department with lower sales occurs after a department with higher sales.
9	Will always occur at least once.	All departments do not have the same sales.
13	There is an employee in an eligible department.	There is an employee who is not in an eligible department.
16	An eligible employee is either a manager or earns LSALARY or more.	An eligible employee is not a manager and earns less than LSALARY.
21	All eligible departments contain no employees.	An eligible department contains at least one employee.

Given the small number of decisions, we probably should opt for multicondition coverage, but we will examine all the logic coverage criteria (except statement coverage, which always is too limited to be of use) to see their effects.

To satisfy the decision coverage criterion, we need sufficient test cases to invoke both outcomes of each of the six decisions. The required input situations to invoke all decision outcomes are listed in Table 5.1. Since two of the outcomes will always occur, there are 10 situations that need to be forced by test cases. Note that to construct Table 5.1, decision-outcome circumstances had to be traced back through the logic of the program to determine the proper corresponding input circumstances. For instance, decision 16 is not invoked by any employee meeting the conditions; the employee must be in an eligible department.

The 10 situations of interest in Table 5.1 could be invoked by the two test cases shown in Figure 5.3. Note that each test case includes a definition of the expected output, in adherence to the principles discussed in Chapter 2.

Although these two test cases meet the decision coverage criterion, it should be obvious that there could be many types of errors in the module that are not detected by these two test cases. For instance, the test cases do not explore the circumstances where the error code is 0, an employee is a manager, or the department table is empty (DSIZE<=0).

<table>
<tr><th>Test case</th><th>Input</th><th>Expected output</th></tr>
<tr><td>1</td><td>ESIZE = 0
All other inputs are irrelevant</td><td>ERRCODE = 1
ESIZE, DSIZE, EMPTAB, and DEPTTAB are unchanged</td></tr>
<tr><td>2</td><td>ESIZE = DSIZE = 3
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21,000.00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10,000.00</td></tr></table>DEPTTAB
<table><tr><td>D42</td><td>10,000.00</td></tr><tr><td>D32</td><td>8,000.00</td></tr><tr><td>D95</td><td>10,000.00</td></tr></table></td><td>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>JONES</td><td>E</td><td>D42</td><td>21,100.00</td></tr><tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr><tr><td>LORIN</td><td>E</td><td>D42</td><td>10,200.00</td></tr></table></td></tr>
</table>

FIGURE 5.3 Test Cases to Satisfy the Decision-Coverage Criterion.

A more satisfactory test can be obtained by using the condition coverage criterion. Here we need sufficient test cases to invoke both outcomes of each condition in the decisions. The conditions and required input situations to invoke all outcomes are listed in Table 5.2. Since two of the outcomes will always occur, there are 14 situations that must be forced by test cases. Again, these situations can be invoked by only two test cases, as shown in Figure 5.4.

The test cases in Figure 5.4 were designed to illustrate a problem. Since they do invoke all the outcomes in Table 5.2, they satisfy the condition coverage criterion, but they are probably a poorer set of test cases than those in Figure 5.3 in terms of satisfying the decision coverage criterion. The reason is that they do not execute every statement. For example, statement 18 is never executed. Moreover, they do not accomplish much more than the test cases in Figure 5.3. They do not cause the output situation `ERRORCODE=0`. If statement 2 had erroneously set `ESIZE=0` and `DSIZE=0`, this error would go undetected. Of course, an alternative set of test cases might solve these problems, but the fact remains that the two test cases in Figure 5.4 do satisfy the condition coverage criterion.

Using the decision/condition coverage criterion would eliminate the major weakness in the test cases in Figure 5.4. Here we would provide sufficient test cases such that all outcomes of all conditions *and* decisions would be invoked at least once. Making Jones a manager and making Lorin a nonmanager could accomplish this. This would have the result of generating both outcomes of decision 16, thus causing us to execute statement 18.

TABLE 5.2 Situations Corresponding to the Condition Outcomes

Decision	Condition	True Outcome	False Outcome
2	ESIZE≤0	ESIZE≤0	ESIZE>0
2	DSIZE≤0	DSIZE≤0	DSIZE>0
6	SALES(I)≥ MAXSALES	Will always occur at least once.	Order DEPTTAB so that a department with lower sales occurs after a department with higher sales.
9	SALES(J)= MAXSALES	Will always occur at least once.	All departments do not have the same sales.
13	EMPTAB.DEPT (K)= DEPTTAB. DEPT(J)	There is an employee in an eligible department.	There is an employee who is not in an eligible department.
16	SALARY(K)≥ LSALARY	An eligible employee earns LSALARY or more.	An eligible employee earns less than LSALARY.
16	CODE(K)=MGR	An eligible employee is a manager.	An eligible employee is not a manager.
21	–FOUND	An eligible department contains no employees.	An eligible department contains at least one employee.

Test case	Input	Expected output
1	ESIZE = DSIZE = 0 All other inputs are irrelevant	ERRCODE = 1 ESIZE, DSIZE, EMPTAB, and DEPTTAB are unchanged
2	ESIZE = DSIZE = 3 EMPTAB JONES E D42 21,000.00 SMITH E D32 14,000.00 LORIN M D42 10,000.00 DEPTTAB D42 10,000.00 D32 8,000.00 D95 10,000.00	ERRCODE = 2 ESIZE, DSIZE, and DEPTTAB are unchanged EMPTAB JONES E D42 21,000.00 SMITH E D32 14,000.00 LORIN M D42 10,100.00

FIGURE 5.4 Test Cases to Satisfy the Condition Coverage Criterion.

One problem with this, however, is that it is essentially no better than the test cases in Figure 5.3. If the compiler being used stops evaluating an *or* expression as soon as it determines that one operand is *true,* this modification would result in the expression `CODE(K)=MGR` in statement 16 never having a *true* outcome. Hence, if this expression were coded incorrectly, the test cases would not detect the error.

The last criterion to explore is multicondition coverage. This criterion requires sufficient test cases such that all possible combinations of conditions in each decision are invoked at least once. This can be accomplished by working from Table 5.2. Decisions 6, 9, 13, and 21 have two combinations each; decisions 2 and 16 have four combinations each. The methodology to design the test cases is to select one that covers as many of the combinations as possible, select another that covers as many of the remaining combinations as possible, and so on. A set of test cases satisfying the multicondition coverage criterion is shown in Figure 5.5. The set is more

<table>
<tr><th>Test case</th><th>Input</th><th>Expected output</th></tr>
<tr><td>1</td><td>ESIZE = 0 DSIZE = 0
All other inputs are irrelevant</td><td>ERRCODE = 1
ESIZE, DSIZE, EMPTAB, and DEPTTAB are unchanged</td></tr>
<tr><td>2</td><td>ESIZE = 0 DSIZE > 0
All other inputs are irrelevant</td><td>Same as above</td></tr>
<tr><td>3</td><td>ESIZE > 0 DSIZE = 0
All other inputs are irrelevant</td><td>Same as above</td></tr>
<tr><td>4</td><td>ESIZE = 5 DSIZE = 4
EMPTAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21,000.00</td></tr>
<tr><td>WARNS</td><td>M</td><td>D95</td><td>12,000.00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10,000.00</td></tr>
<tr><td>TOY</td><td>E</td><td>D95</td><td>16,000.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr>
</table>
DEPTTAB
<table>
<tr><td>D42</td><td>10,000.00</td></tr>
<tr><td>D32</td><td>8,000.00</td></tr>
<tr><td>D95</td><td>10,000.00</td></tr>
<tr><td>D44</td><td>10,000.00</td></tr>
</table></td><td>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table>
<tr><td>JONES</td><td>M</td><td>D42</td><td>21,100.00</td></tr>
<tr><td>WARNS</td><td>M</td><td>D95</td><td>12,100.00</td></tr>
<tr><td>LORIN</td><td>E</td><td>D42</td><td>10,200.00</td></tr>
<tr><td>TOY</td><td>E</td><td>D95</td><td>16,100.00</td></tr>
<tr><td>SMITH</td><td>E</td><td>D32</td><td>14,000.00</td></tr>
</table></td></tr>
</table>

FIGURE 5.5 Test Cases to Catisfy the Multicondition Coverage Criterion.

comprehensive than the previous sets of test cases, implying that we should have selected this criterion at the beginning.

It is important to realize that module *BONUS* could have such a large number of errors that even the tests satisfying the multicondition coverage criterion would not detect them all. For instance, no test cases generate the situation where ERRORCODE is returned with a value of 0; thus, if statement 1 were missing, the error would go undetected. If LSALARY were erroneously initialized to $150,000.01, the mistake would go unnoticed. If statement 16 stated SALARY(K)>LSALARY instead of SALARY(K) >=LSALARY, this error would not be found. Also, whether a variety of off-by-one errors (such as not handling the last entry in DEPTTAB or EMPTAB correctly) would be detected would depend largely on chance.

Two points should be apparent now: One, the multicondition criterion is superior to the other criteria, and, two, any logic coverage criterion is not good enough to serve as the only means of deriving module tests. Hence, the next step is to supplement the tests in Figure 5.5 with a set of black-box tests. To do so, the interface specifications of *BONUS* are shown in the following:

> *BONUS*, a PL/1 module, receives five parameters, symbolically referred to here as EMPTAB, DEPTTAB, ESIZE, DSIZE, and ERRORCODE. The attributes of these parameters are:

```
DECLARE 1 EMPTAB(*), /*INPUT AND OUTPUT*/
    2 NAME CHARACTER(6),
    2 CODE CHARACTER(1),
    2 DEPT CHARACTER(3),
    2 SALARY FIXED DECIMAL(7,2);
DECLARE 1 DEPTTAB(*), /*INPUT*/
    2 DEPT CHARACTER(3),
    2 SALES FIXED DECIMAL(8,2);
DECLARE (ESIZE, DSIZE) FIXED BINARY; /*INPUT*/
DECLARE ERRCODE FIXED DECIMAL(1); /*OUTPUT*/
```

The module assumes that the transmitted arguments have these attributes. ESIZE and DSIZE indicate the number of entries in EMPTAB and DEPTTAB, respectively. No assumptions should be made about the order of entries in EMPTAB and DEPTTAB. The function of the module is to increment the salary (EMPTAB.SALARY) of those employees in the department or departments having the largest sales amount (DEPTTAB.SALES). If an eligible

employee's current salary is $150,000 or more, or if the employee is a manager (EMPTAB.CODE='M'), the increment is $1,000; if not, the increment for the eligible employee is $2,000. The module assumes that the incremented salary will fit into field EMPTAB.SALARY. If ESIZE and DSIZE are not greater than 0, ERRCODE is set to 1 and no further action is taken. In all other cases, the function is completely performed. However, if a maximum-sales department is found to have no employee, processing continues but ERRCODE will have the value 2; otherwise, it is set to 0.

This specification is not suited to cause-effect graphing (there is not a discernible set of input conditions whose combinations should be explored); thus, boundary value analysis will be used. The input boundaries identified are as follows:

1. EMPTAB has 1 entry.
2. EMPTAB has the maximum number of entries (65,535).
3. EMPTAB has 0 entries.
4. DEPTTAB has 1 entry.
5. DEPTTAB has 65,535 entries.
6. DEPTTAB has 0 entries.
7. A maximum-sales department has 1 employee.
8. A maximum-sales department has 65,535 employees.
9. A maximum-sales department has no employees.
10. All departments in DEPTTAB have the same sales.
11. The maximum-sales department is the first entry in DEPTTAB.
12. The maximum-sales department is the last entry in DEPTTAB.
13. An eligible employee is the first entry in EMPTAB.
14. An eligible employee is the last entry in EMPTAB.
15. An eligible employee is a manager.
16. An eligible employee is not a manager.
17. An eligible employee who is not a manager has a salary of $149,999.99.
18. An eligible employee who is not a manager has a salary of $150,000.
19. An eligible employee who is not a manager has a salary of $150,000.01.

    The output boundaries are as follows:

20. ERRCODE=0
21. ERRCODE=1
22. ERRCODE=2
23. The incremented salary of an eligible employee is $299,999.99.

A further test condition based on the error-guessing technique is as follows:

24. A maximum-sales department with no employees is followed in DEPTTAB with another maximum-sales department having employees.

This is used to determine whether the module erroneously terminates processing of the input when it encounters an ERRCODE=2 situation.

Reviewing these 24 conditions, numbers 2, 5, and 8 seem like impractical test cases. Since they also represent conditions that will never occur (usually a dangerous assumption to make when testing, but seemingly safe here), we exclude them. The next step is to compare the remaining 21 conditions to the current set of test cases (Figure 5.5) to determine which boundary conditions are not already covered. Doing so, we see that conditions 1, 4, 7, 10, 14, 17, 18, 19, 20, 23, and 24 require test cases beyond those in Figure 5.5.

The next step is to design additional test cases to cover the 11 boundary conditions. One approach is to merge these conditions into the existing test cases (i.e., by modifying test case 4 in Figure 5.5), but this is not recommended because doing so could inadvertently upset the complete multicondition coverage of the existing test cases. Hence, the safest approach is to add test cases to those of Figure 5.5. In doing this, the goal is to design the smallest number of test cases necessary to cover the boundary conditions. The three test cases in Figure 5.6 accomplish this. Test case 5 covers conditions 7, 10, 14, 17, 18, 19, and 20; test case 6 covers conditions 1, 4, and 23; and test case 7 covers condition 24.

The premise here is that the logic coverage, or white-box, test cases in Figure 5.6 form a reasonable module test for procedure *BONUS*.

Incremental Testing

In performing the process of module testing, there are two key considerations: the design of an effective set of test cases, which was discussed in the previous section, and the manner in which the modules are combined to form a working program. The second consideration is important because it has these implications:

- The form in which module test cases are written
- The types of test tools that might be used

<table>
<tr><th>Test case</th><th>Input</th><th>Expected output</th></tr>
<tr><td>5</td><td>ESIZE = 3 DSIZE = 2
EMPTAB
<table><tr><td>ALLY</td><td>E</td><td>D36</td><td>14,999.99</td></tr><tr><td>BEST</td><td>E</td><td>D33</td><td>15,000.00</td></tr><tr><td>CELTO</td><td>E</td><td>D33</td><td>15,000.01</td></tr></table>DEPTTAB
<table><tr><td>D33</td><td>55,400.01</td></tr><tr><td>D36</td><td>55,400.01</td></tr></table></td><td>ERRCODE = 0
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>ALLY</td><td>E</td><td>D36</td><td>15,199.99</td></tr><tr><td>BEST</td><td>E</td><td>D33</td><td>15,100.00</td></tr><tr><td>CELTO</td><td>E</td><td>D33</td><td>15,100.01</td></tr></table></td></tr>
<tr><td>6</td><td>ESIZE = 1 DSIZE = 1
EMPTAB
<table><tr><td>CHIEF</td><td>M</td><td>D99</td><td>99,899.99</td></tr></table>DEPTTAB
<table><tr><td>D99</td><td>99,000.00</td></tr></table></td><td>ERRCODE = 0
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>CHIEF</td><td>M</td><td>D99</td><td>99,999.99</td></tr></table></td></tr>
<tr><td>7</td><td>ESIZE = 2 DSIZE = 2
EMPTAB
<table><tr><td>DOLE</td><td>E</td><td>D67</td><td>10,000.00</td></tr><tr><td>FORD</td><td>E</td><td>D22</td><td>33,333.33</td></tr></table>DEPTTAB
<table><tr><td>D66</td><td>20,000.00</td></tr><tr><td>D67</td><td>20,000.00</td></tr></table></td><td>ERRCODE = 2
ESIZE, DSIZE, and DEPTTAB are unchanged
EMPTAB
<table><tr><td>DOLE</td><td>E</td><td>D67</td><td>10,000.00</td></tr><tr><td>FORD</td><td>E</td><td>D22</td><td>33,333.33</td></tr></table></td></tr>
</table>

FIGURE 5.6 Supplemental Boundary Value Analysis Test Cases for BONUS.

- The order in which modules are coded and tested
- The cost of generating test cases
- The cost of debugging (locating and repairing detected errors)

In short, then, it is a consideration of substantial importance. In this section, we discuss two approaches, incremental and nonincremental testing; in the next, we explore two incremental approaches, top-down and bottom-up development or testing.

The question pondered here is the following: Should you test a program by testing each module independently and then combining the modules to

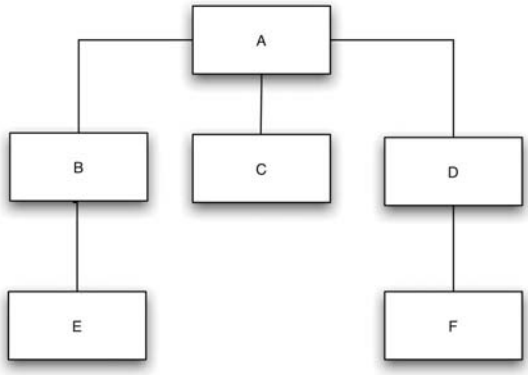

FIGURE 5.7 Sample Six-Module Program.

form the program, or should you combine the next module to be tested with the set of previously tested modules before it is tested? The first approach is called *nonincremental*, or "big-bang," testing or integration; the second approach is known as *incremental* testing or integration.

The program in Figure 5.7 is used as an example. The rectangles represent the six modules (subroutines or procedures) in the program. The lines connecting the modules represent the control hierarchy of the program; that is, module *A* calls modules *B*, *C*, and *D*; module *B* calls module *E*; and so on. Nonincremental testing, the traditional approach, is performed in the following manner. First, a module test is performed on each of the six modules, testing each module as a stand-alone entity. The modules might be tested at the same time or in succession, depending on the environment (e.g., interactive versus batch-processing computing facilities) and the number of people involved. Finally, the modules are combined or integrated (e.g., "link edited") to form the program.

The testing of each module requires a special *driver* module and one or more *stub modules*. For instance, to test module *B*, test cases are first designed and then fed to module *B* by passing it input arguments from a driver module, a small module that must be coded to "drive," or transmit, test cases through the module under test. (Alternatively, a test tool could be used.) The driver module must also display, to the tester, the results produced by *B*. In addition, since module *B* calls module *E*, something must be present to receive control when *B* calls *E*. A stub module, a special module given the name "*E*" that must be coded to simulate the function of module *E*, accomplishes this.

When the module testing of all six modules has been completed, the modules are combined to form the program.

The alternative approach is incremental testing. Rather than testing each module in isolation, the next module to be tested is first combined with the set of modules that have been tested already.

It is premature to give a procedure for incrementally testing the program in Figure 5.7, because there is a large number of possible incremental approaches. A key issue is whether we should begin at the top or bottom of the program. However, since we discuss this issue in the next section, let us assume for the moment that we are beginning from the bottom.

The first step is to test modules *E*, *C*, and *F*, either in parallel (by three people) or serially. Notice that we must prepare a driver for each module, but not a stub. The next step is to test *B* and *D*; but rather than testing them in isolation, they are combined with modules *E* and *F*, respectively. In other words, to test module *B*, a driver is written, incorporating the test cases, and the pair *B*-*E* is tested. The incremental process, adding the next module to the set or subset of previously tested modules, is continued until the last module (module *A* in this case) is tested. Note that this procedure could have alternatively progressed from the top to the bottom.

Several observations should be apparent at this point:

1. Nonincremental testing requires more work. For the program in Figure 5.7, five drivers and five stubs must be prepared (assuming we do not need a driver module for the top module). The bottom-up incremental test would require five drivers but no stubs. A top-down incremental test would require five stubs but no drivers. Less work is required because previously tested modules are used instead of the driver modules (if you start from the top) or stub modules (if you start from the bottom) needed in the nonincremental approach.
2. Programming errors related to mismatching interfaces or incorrect assumptions among modules will be detected earlier when incremental testing is used. The reason is that combinations of modules are tested together at an early point in time. However, when nonincremental testing is used, modules do not "see one another" until the end of the process.
3. As a result, debugging should be easier if incremental testing is used. If we assume that errors related to intermodule interfaces and assumptions do exist (a good assumption, from experience), then, if

nonincremental testing has been used, the errors will not surface until the entire program has been combined. At this time, we may have difficulty pinpointing the error, since it could be anywhere within the program. Conversely, if incremental testing is used, an error of this type should be easier to pinpoint, because it is likely that the error is associated with the most recently added module.

4. Incremental testing might result in more thorough testing. If you are testing module *B*, either module *E* or *A* (depending on whether you started from the bottom or the top) is executed as a result. Although *E* or *A* should have been thoroughly tested previously, perhaps executing it as a result of *B*'s module test will invoke a new condition, perhaps one that represents a deficiency in the original test of *E* or *A*. On the other hand, if nonincremental testing is used, the testing of *B* will affect only module *B*. In other words, incremental testing substitutes previously tested modules for the stubs or drivers needed in the nonincremental test. As a result, the actual modules receive more exposure by the completion of the last module test.
5. The nonincremental approach appears to use less machine time. If module *A* of Figure 5.7 is being tested using the bottom-up approach, modules *B*, *C*, *D*, *E*, and *F* probably execute during the execution of *A*. In a nonincremental test of *A*, only stubs for *B*, *C*, and *E* are executed. The same is true for a top-down incremental test. If module *F* is being tested, modules *A*, *B*, *C*, *D*, and *E* may be executed during the test of *F*; in the nonincremental test of *F*, only the driver for *F*, plus *F* itself, executes. Hence, the number of machine instructions executed during a test run using the incremental approach is apparently greater than that for the nonincremental approach. Offsetting this is the fact that the nonincremental test requires more drivers and stubs than the incremental test; machine time is needed to develop the drivers and stubs.
6. At the beginning of the module testing phase, there is more opportunity for parallel activities when nonincremental testing is used (that is, all the modules can be tested simultaneously). This might be of significance in a large project (many modules and people), since the head count of a project is usually at its peak at the start of the module test phase.

In summary, observations 1 through 4 are advantages of incremental testing, while observations 5 and 6 are disadvantages. Given current trends

in the computing industry (hardware costs have been decreasing, and seem destined to continue to do so, while hardware capability increases, and labor costs and the consequences of software errors are increasing), and given the fact that the earlier an error is found, the lower the cost of repairing it, you can see that observations 1 through 4 are growing in importance, whereas observation 5 is becoming less important. Observation 6 seems to be a weak disadvantage, if one at all. This leads to the conclusion that incremental testing is superior.

Top-Down versus Bottom-Up Testing

Given the conclusion of the previous section—that incremental testing is superior to nonincremental testing—we next explore two incremental strategies: top-down and bottom-up testing. Before getting into them, however, we should clarify several misconceptions. First, the terms *top-down testing*, *top-down development*, and *top-down design* often are used as synonyms. Top-down testing and top-down development *are* synonyms (they represent a strategy of ordering the coding and testing of modules), but top-down design is something quite different and independent. A program that was designed in top-down fashion can be incrementally tested in either a top-down or a bottom-up fashion.

Second, bottom-up testing (or bottom-up development) is often mistakenly equated with nonincremental testing. The reason is that bottom-up testing begins in a manner that is identical to a nonincremental test (i.e., when the bottom, or terminal, modules are tested), but as we saw in the previous section, bottom-up testing is an incremental strategy. Finally, since both strategies are incremental, we won't repeat here the advantages of incremental testing; we will discuss only the differences between top-down and bottom-up testing.

Top-Down Testing

The top-down strategy starts with the top, or initial, module in the program. After this, there is no single "right" procedure for selecting the next module to be incrementally tested; the only rule is that to be eligible to be the next module, at least one of the module's subordinate (calling) modules must have been tested previously.

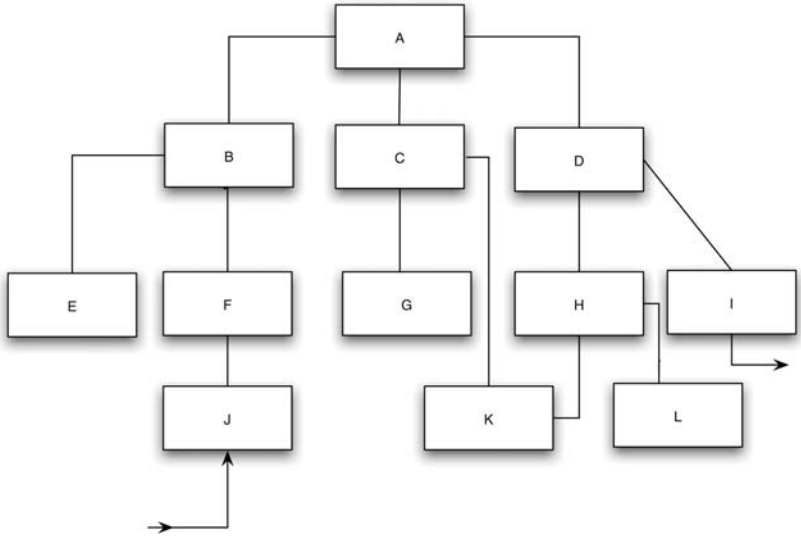

FIGURE 5.8 Sample 12-Module Program.

Figure 5.8 is used to illustrate this strategy. *A* through *L* are the 12 modules in the program. Assume that module *J* contains the program's I/O read operations and module *I* contains the write operations.

The first step is to test module *A*. To accomplish this, stub modules representing *B*, *C*, and *D* must be written. Unfortunately, the production of stub modules is often misunderstood; as evidence, you may often see such statements as "a stub module need only write a message stating 'we got this far' "; and, "in many cases, the dummy module (stub) simply exits—without doing any work at all." In most situations, these statements are false. Since module *A* calls module *B*, *A* is expecting *B* to perform some work; this work most likely is some result (output arguments) returned to *A*. If the stub simply returns control or writes an error message without returning a meaningful result, module *A* will fail, not because of an error in *A*, but because of a failure of the stub to simulate the corresponding module. Moreover, returning a "wired-in" output from a stub module is often insufficient. For instance, consider the task of writing a stub representing a square-root routine, a database table-search routine, an "obtain corresponding master-file record" routine, or the like. If the stub returns a fixed wired-in output, but doesn't have the particular value expected by the calling module during this invocation, the calling module may fail or produce a confusing result. Hence, the production of stubs is not a trivial task.

Another consideration is the form in which test cases are presented to the program, an important consideration that is not even mentioned in most discussions of top-down testing. In our example, the question is: How do you feed test cases to module *A*? The top module in typical programs neither receives input arguments nor performs input/output operations, so the answer is not immediately obvious. The answer is that the test data are fed to the module (module *A* in this situation) from one or more of its stubs. To illustrate, assume that the functions of *B*, *C*, and *D* are as follows:

B—Obtain summary of transaction file.
C—Determine whether weekly status meets quota.
D—Produce weekly summary report.

A test case for *A*, then, is a transaction summary returned from stub *B*. Stub *D* might contain statements to write its input data to a printer, allowing the results of each test to be examined.

In this program, another problem exists. Presumably, module *A* calls module *B* only once; therefore the problem is how to feed more than one test case to *A*. One solution is to develop multiple versions of stub *B*, each with a different wired-in set of test data to be returned to *A*. To execute the test cases, the program is executed multiple times, each time with a different version of stub *B*. Another alternative is to place test data on external files and have stub *B* read the test data and return them to *A*. In either case, keeping in mind the previous discussion, you should see that the development of stub modules is more difficult than it is often made out to be. Furthermore, it often is necessary, because of the characteristics of the program, to represent a test case across multiple stubs beneath the module under test (i.e., where the module receives data to be acted upon by calling multiple modules).

After *A* has been tested, an actual module replaces one of the stubs, and the stubs required by that module are added. For instance, Figure 5.9 might represent the next version of the program.

After testing the top module, numerous sequences are possible. For instance, if we are performing all the testing sequences, four examples of the many possible sequences of modules are:

1. A B C D E F G H I J K L
2. A B E F J C G K D H L I
3. A D H I K L C G B F J E
4. A B F J D I E C G K H L

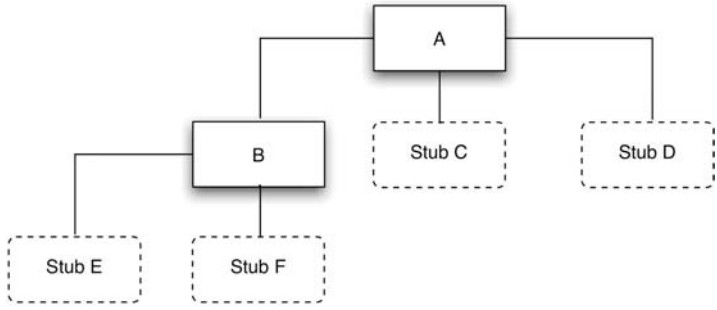

FIGURE 5.9 Second Step in the Top-Down Test.

If parallel testing occurs, other alternatives are possible. For instance, after module *A* has been tested, one programmer could take module *A* and test the combination A-B; another programmer could test A-C; and a third could test A-D. In general, there is no best sequence, but here are two guidelines to consider:

1. If there are critical sections of the program (perhaps module *G*), design the sequence such that these sections are added as early as possible. A "critical section" might be a complex module, a module with a new algorithm, or a module suspected to be error prone.
2. Design the sequence such that the I/O modules are added as early as possible.

The motivation for the first should be obvious, but the motivation for the second deserves further discussion. Recall that a problem with stubs is that some of them must contain the test cases, and others must write their input to a printer or display. However, as soon as the module accepting the program's input is added, the representation of test cases is considerably simplified; their form is identical to the input accepted by the final program (e.g., from a transaction file or a terminal). Likewise, once the module performing the program's output function is added, the placement of code in stub modules to write results of test cases might no longer be necessary. Thus, if modules *J* and *I* are the I/O modules, and if module *G* performs some critical function, the incremental sequence might be

A B F J D I C G E K H L

and the form of the program after the sixth increment would be that shown in Figure 5.10.

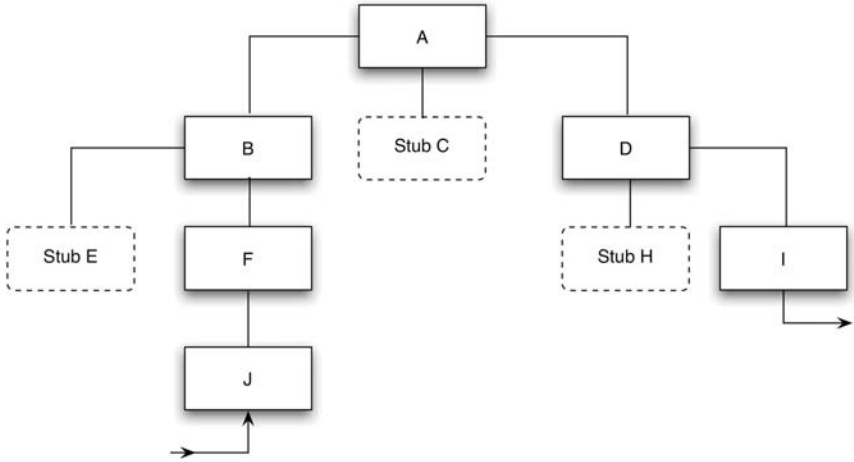

FIGURE 5.10 Intermediate State in the Top-Down Test.

Once the intermediate state in Figure 5.10 has been reached, the representation of test cases and the inspection of results are simplified. It has another advantage, in that you have a working skeletal version of the program, that is, a version that performs actual input and output operations. However, stubs are still simulating some of the "insides." This early skeletal version:

- Allows you to find human-factor errors and problems.
- Makes it possible to demonstrate the program to the eventual user.
- Serves as evidence that the overall design of the program is sound.
- Serves as a morale booster.

These points represent the major advantage of the top-down strategy.

On the other hand, the top-down approach has some serious shortcomings. Assume that our current state of testing is that of Figure 5.10 and that our next step is to replace stub *H* with module *H*. What we should do at this point (or earlier) is use the methods described earlier in this chapter to design a set of test cases for *H*. Note, however, that the test cases are in the form of actual program inputs to module *J*. This presents several problems. First, because of the intervening modules between *J* and *H* (*F*, *B*, *A*, and *D*), we might find it impossible to represent certain test cases to module *J* that test every predefined situation in *H*. For instance, if *H* is the *BONUS* module of Figure 5.2, it might be impossible, because of the nature of intervening module *D*, to create some of the seven test cases of Figures 5.5 and 5.6.

Second, because of the "distance" between *H* and the point at which the test data enter the program, even if it were possible to test every situation, determining which data to feed to *J* to test these situations in *H* is often a difficult mental task.

Third, because the displayed output of a test might come from a module that is a large distance away from the module being tested, correlating the displayed output to what went on in the module may be difficult or impossible. Consider adding module *E* to Figure 5.10. The results of each test case are determined by examining the output written by module *I*, but because of the intervening modules, it may be difficult to deduce the actual output of *E* (that is, the data returned to *B*).

The top-down strategy, depending on how it is approached, may have two further problems. People occasionally feel that the strategy can be overlapped with the program's design phase. For instance, if you are in the process of designing the program in Figure 5.8, you might believe that after the first two levels are designed, modules *A* through *D* can be coded and tested while the design of the lower levels progresses. As we have emphasized elsewhere, this is usually an unwise decision. Program design is an iterative process, meaning that when we are designing the lower levels of a program's structure, we may discover desirable changes or improvements to the upper levels. If the upper levels have already been coded and tested, the desirable improvements will most likely be discarded, an unwise decision in the long run.

A final problem that often arises in practice is failing to completely test a module before proceeding to another module. This occurs for two reasons: because of the difficulty of embedding test data in stub modules, and because the upper levels of a program usually provide resources to lower levels. In Figure 5.8 we saw that testing module *A* might require multiple versions of the stub for module *B*. In practice, there is a tendency to say, "Because this represents a lot of work, I won't execute all of *A*s test cases now. I'll wait until I place module *J* in the program, at which time the representation of test cases will be easier, and remember at this point to finish testing module *A*." Of course, the problem here is that we may forget to test the remainder of module *A* at this later point in time. Also, because upper levels often provide resources for use by lower levels (e.g., opening of files), it is difficult sometimes to determine whether the resources have been provided correctly (e.g., whether a file has been opened with the proper attributes) until the lower modules that use them are tested.

Bottom-Up Testing

The next step is to examine the bottom-up incremental testing strategy. For the most part, bottom-up testing is the opposite of top-down testing; thus, the advantages of top-down testing become the disadvantages of bottom-up testing, and the disadvantages of top-down testing become the advantages of bottom-up testing. Because of this, the discussion of bottom-up testing is shorter.

The bottom-up strategy begins with the terminal modules in the program (the modules that do not call other modules). After these modules have been tested, again there is no best procedure for selecting the next module to be incrementally tested; the only rule is that to be eligible to be the next module, all of the module's subordinate modules (the modules it calls) must have been tested previously.

Returning to Figure 5.8, the first step is to test some or all of modules *E*, *J*, *G*, *K*, *L*, and *I*, either serially or in parallel. To do so, each module needs a special driver module: a module that contains wired-in test inputs, calls the module being tested, and displays the outputs (or compares the actual outputs with the expected outputs). Unlike the situation with stubs, multiple versions of a driver are not needed, since the driver module can iteratively call the module being tested. In most cases, driver modules are easier to produce than stub modules.

As was the case earlier, a factor influencing the sequence of testing is the critical nature of the modules. If we decide that modules *D* and *F* are most critical, an intermediate state of the bottom-up incremental test might be that of Figure 5.11. The next steps might be to test *E* and then test *B*, combining *B* with the previously tested modules *E*, *F*, and *J*.

A drawback of the bottom-up strategy is that there is no concept of an early skeletal program. In fact, the working program does not exist until the last module (module *A*) is added, and this working program is the complete program. Although the I/O functions can be tested before the whole program has been integrated (the I/O modules are being used in Figure 5.11), the advantages of the early skeletal program are not present.

The problems associated with the impossibility, or difficulty, of creating all test situations in the top-down approach do not exist here. If you think of a driver module as a test probe, the probe is being placed directly on the module being tested; there are no intervening modules to worry about. Examining other problems associated with the top-down approach, you

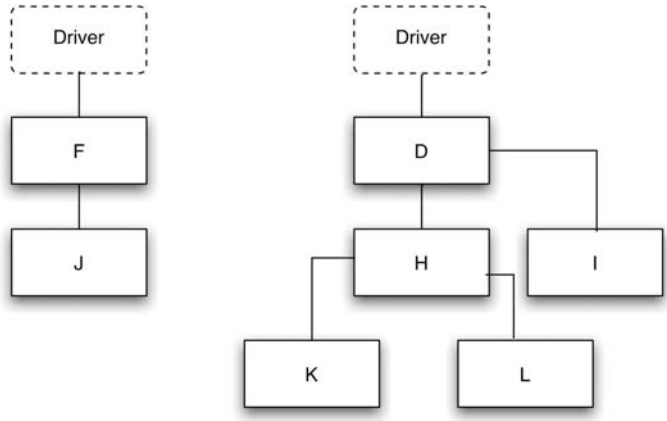

FIGURE 5.11 Intermediate State in the Bottom-Up Test.

can't make the unwise decision to overlap design and testing, since the bottom-up test cannot begin until the bottom of the program has been designed. Also, the problem of not completing the test of a module before starting another, because of the difficulty of encoding test data in versions of a stub, does not exist when using bottom-up testing.

A Comparison

It would be convenient if the top-down versus bottom-up issue were as clear-cut as the incremental versus nonincremental issue, but unfortunately it is not. Table 5.3 summarizes the relative advantages and disadvantages of the two approaches (excluding the previously discussed advantages shared by both—those of incremental testing). The first advantage of each approach might appear to be the deciding factor, but there is no evidence showing that major flaws occur more often at the top or bottom levels of the typical program. The safest way to make a decision is to weigh the factors in Table 5.3 with respect to the particular program being tested. Lacking such a program here, the serious consequences of the fourth disadvantage—of top-down testing and the availability of test tools that eliminate the need for drivers but not stubs—seems to give the bottom-up strategy the edge.

Furthermore, it may be apparent that top-down and bottom-up testing are not the only possible incremental strategies.

TABLE 5.3 Comparison of Top-Down and Bottom-Up Testing

Top-Down Testing	
Advantages	**Disadvantages**
1. Advantageous when major flaws occur toward the top of the program. 2. Once the I/O functions are added, representation of cases is easier. 3. Early skeletal program allows demonstrations and boosts morale.	1. Stub modules must be produced. 2. Stub modules are often more complicated than they first appear to be. 3. Before the I/O functions are added, the representation of test cases in stubs can be difficult. 4. Test conditions may be impossible, or very difficult, to create. 5. Observation of test output is more difficult. 6. Leads to the conclusion that design and testing can be overlapped. 7. Defers the completion of testing certain modules.
Bottom-Up Testing	
Advantages	**Disadvantages**
1. Advantageous when major flaws occur toward the bottom of the program. 2. Test conditions are easier to create. 3. Observation of test results is easier.	1. Driver modules must be produced. 2. The program as an entity does not exist until the last module is added.

Performing the Test

The remaining part of the module test is the act of actually carrying out the test. A set of hints and guidelines for doing this is included here.

When a test case produces a situation where the module's actual results do not match the expected results, there are two possible explanations: either the module contains an error, or the expected results are incorrect (the test case is incorrect). To minimize this confusion, the set of test cases

should be reviewed or inspected before the test is performed (that is, the test cases should be tested).

The use of automated test tools can minimize part of the drudgery of the testing process. For instance, test tools exist that eliminate the need for driver modules. Flow-analysis tools enumerate the paths through a program, find statements that can never be executed ("unreachable" code), and identify instances where a variable is used before it is assigned a value.

As was the practice earlier in this chapter, remember that a definition of the expected result is a necessary part of a test case. When executing a test, remember to look for side effects (instances where a module does something it is not supposed to do). In general, these situations are difficult to detect, but some of them may be found by checking, after execution of the test case, the inputs to the module that are not supposed to be altered. For instance, test case 7 in Figure 5.6 states that as part of the expected result, `ESIZE`, `DSIZE`, and `DEPTTAB` should be unchanged. When running this test case, not only should the output be examined for the correct result, but `ESIZE`, `DSIZE`, and `DEPTTAB` should be examined to determine whether they were erroneously altered.

The psychological problems associated with a person attempting to test his or her own programs apply as well to module testing. Rather than testing their own modules, programmers might swap them; more specifically, the programmer of the calling module is always a good candidate to test the called module. Note that this applies only to testing; the debugging of a module always should be performed by the original programmer.

Avoid throwaway test cases; represent them in such a form that they can be reused in the future. Recall the counterintuitive phenomenon in Figure 2.2. If an abnormally high number of errors is found in a subset of the modules, it is likely that these modules contain even more, as yet undetected, errors. Such modules should be subjected to further module testing, and possibly an additional code walkthrough or inspection. Finally, remember that the purpose of a module test is not to demonstrate that the module functions correctly, but to demonstrate the presence of errors in the module.

Summary

In this chapter we introduced you to some of the mechanics of testing, especially as it relates to large programs. This is a process of testing individual program components—subroutines, subprograms, classes, and

procedures. In module testing you compare software functionality with the specification that defines its intended function. Module or unit testing can be an important part of a developer's toolbox to help achieve a reliable application, especially with object-oriented languages such as Java and C#. The goal in module testing is the same as for any other type of software testing: attempt to show how the program contradicts the specification. In addition to the software specification, you will need each module's source code to effect a module test.

Module testing is largely white-box testing. (See Chapter 4 for more information on white-box procedures and designing test cases for testing.) A thorough module test design will include incremental strategies such as top-down as well as bottom-up techniques.

It is helpful, when preparing for a module test, to review the psychological and economic principles laid out in Chapter 2.

One more point: Module testing software is only the beginning of an exhaustive testing procedure. You will need to move on to higher-order testing, which we address in Chapter 6, and user testing, covered in Chapter 7.

6 Higher-Order Testing

When you finish module-testing a program, you have really only just begun the testing process. This is especially true of large or complex programs. Consider this important concept:

> **A software error occurs when the program does not do what its end user reasonably expects it to do.**

Applying this definition, even if you could perform an absolutely perfect module test, you still couldn't guarantee that you have found all software errors. To complete testing, then, some form of further testing is necessary. We call this new form *higher-order* testing.

Software development is largely a process of communicating information about the eventual program and translating this information from one form to another. In essence, it is moving from the conceptual to the concrete. For that reason, the vast majority of software errors can be attributed to breakdowns, mistakes, and "noise" during the communication and translation of information.

This view of software development is illustrated in Figure 6.1, a model of the development cycle for a software product. The flow of the process can be summarized in seven steps:

1. Translate the program user's needs into a set of written requirements. These are the goals for the product.

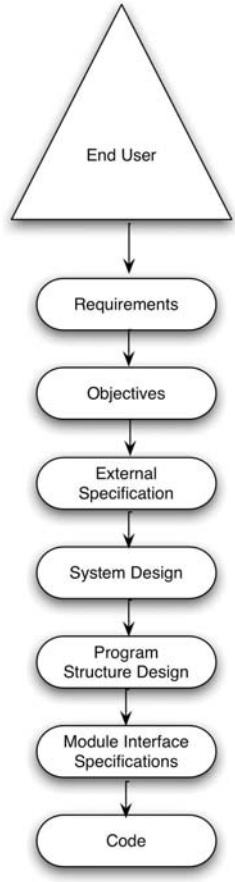

FIGURE 6.1 The Software Development Process.

2. Translate the requirements into specific objectives by assessing feasibility, time, and cost, resolving conflicting requirements, and establishing priorities and trade-offs.
3. Translate the objectives into a precise product specification, viewing the product as a black box and considering only its interfaces and interactions with the end user. This description is called the *external specification*.
4. If the product is a system such as an operating system, flight-control system, database system, or employee personnel management system, rather than an application (e.g., compiler, payroll program, word processor), the next process is system design. This step

partitions the system into individual programs, components, or subsystems, and defines their interfaces.
5. Design the structure of the program or programs by specifying the function of each module, the hierarchical structure of the modules, and the interfaces between modules.
6. Develop a precise specification that defines the interface to, and function of, each module.
7. Translate, through one or more substeps, the module interface specification into the source code algorithm of each module.

Here's another way of looking at these forms of documentation:

- *Requirements* specify why the program is needed.
- *Objectives* specify what the program should do and how well the program should do it.
- *External specifications* define the exact representation of the program to users.
- *Documentation* associated with the subsequent processes specifies, in increasing levels of detail, how the program is constructed.

Given the premise that the seven steps of the development cycle involve communication, comprehension, and translation of information, and the premise that most software errors stem from breakdowns in information handling, there are three complementary approaches to prevent and/or detect these errors.

First, we can introduce more precision into the development process to prevent many of the errors. Second, we can introduce, at the end of each process, a separate verification step to locate as many errors as possible before proceeding to the next process. This approach is illustrated in Figure 6.2. For instance, the external specification is verified by comparing it to the output of the prior stage (the statement of objectives) and feeding back any discovered mistakes to the external specification process. (Use the code inspection and walkthrough methods discussed in Chapter 3 in the verification step at the end of the seventh process.)

The third approach is to orient distinct testing processes toward distinct development processes. That is, focus each testing process on a particular translation step—thus on a particular class of errors. This approach is illustrated in Figure 6.3.

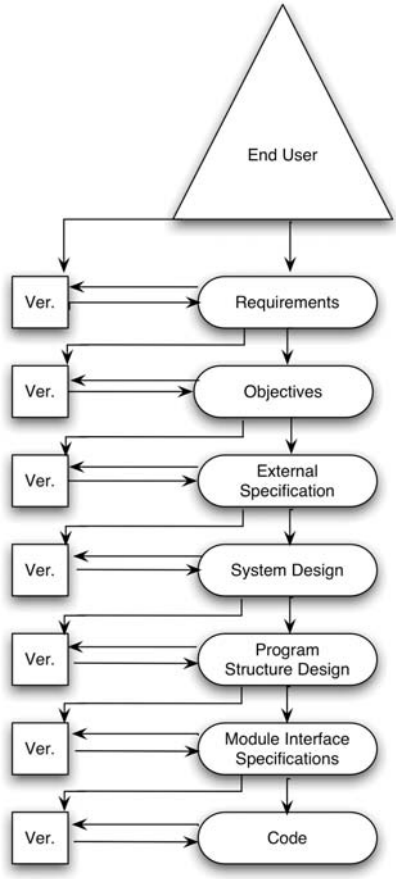

FIGURE 6.2 The Development Process with Intermediate Verification Steps.

The testing cycle is structured to model the development cycle. In other words, you should be able to establish a one-to-one correspondence between development and testing processes. For instance:

- The purpose of a *module test* is to find discrepancies between the program's modules and their interface specifications.
- The purpose of a *function test* is to show that a program does not match its external specifications.
- The purpose of a *system test* is to show that the product is inconsistent with its original objectives.

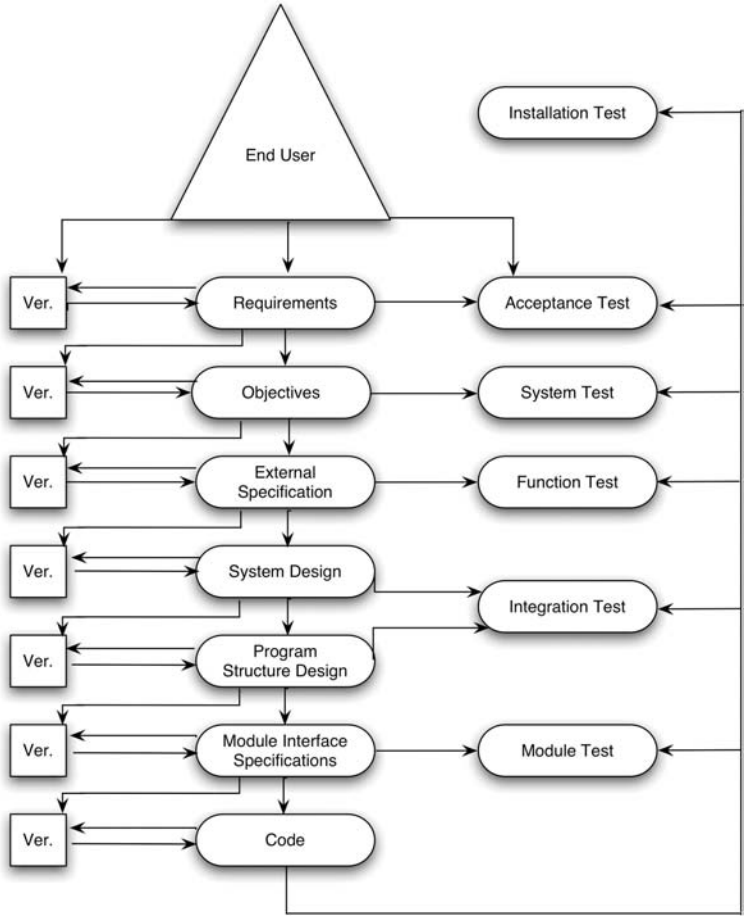

FIGURE 6.3 The Correspondence Between Development and Testing Processes.

Notice how we have structured these statements: "find discrepancies," "does not match," "is inconsistent." Remember that the goal of software testing is to find problems (because we *know* there will be problems!). If you set out to prove that some form of inputs work *properly*, or assume that the program is *true* to its specification and objectives, your testing will be incomplete. Only by setting out to prove that some form of inputs work *improperly*, and assume that the program is *untrue* to its specification and objectives, will your testing be complete. This is an important concept we iterate throughout this book.

The advantages of this structure are that it avoids unproductive redundant testing and prevents you from overlooking large classes of errors. For instance, rather than simply labeling system testing as "the testing of the whole system" and possibly repeating earlier tests, system testing is oriented toward a distinct class of errors (those made during the translation of the objectives to the external specification) and measured with respect to a distinct type of documentation in the development process.

The higher-order testing methods shown in Figure 6.3 are most applicable to software products (programs written as a result of a contract or intended for wide usage, as opposed to experimental programs or those written for use only by the program's author). Programs not written as products often do not have formal requirements and objectives; for such programs, the function test might be the only higher-order test. Also, the need for higher-order testing increases along with the size of the program. The reason is that the ratio of design errors (errors made in the earlier development processes) to coding errors is considerably higher in large programs than in small programs.

Note that the sequence of testing processes in Figure 6.3 does not necessarily imply a time sequence. For instance, since system testing is *not* defined as "the kind of testing you do after function testing," but instead as a distinct type of testing focused on a distinct class of errors, it could very well be partially overlapped in time with other testing processes.

In this chapter, we discuss the processes of function, system, acceptance, and installation testing. We omit integration testing because it is often not regarded as a separate testing step; and, when incremental module testing is used, it is an implicit part of the module test.

We will keep the discussions of these testing processes brief, general, and, for the most part, without examples because specific techniques used in these higher-order tests are highly dependent on the specific program being tested. For instance, the characteristics of a system test (the types of test cases, the manner in which test cases are designed, the test tools used) for an operating system will differ considerably from a system test of a compiler, a program controlling a nuclear reactor, or a database application program.

In the last few sections in this chapter we address planning and organizational issues, along with the important question of determining when to stop testing.

Function Testing

As indicated in Figure 6.3, function testing is a process of attempting to find discrepancies between the program and the external specification. An external specification is a precise description of the program's behavior from the end-user point of view.

Except when used on small programs, function testing is normally a black-box activity. That is, you rely on the earlier module-testing process to achieve the desired white-box logic coverage criteria.

To perform a function test, you analyze the specification to derive a set of test cases. The equivalence partitioning, boundary value analysis, cause-effect graphing, and error-guessing methods described in Chapter 4 are especially pertinent to function testing. In fact, the examples in Chapter 4 are examples of function tests. The descriptions of the Fortran `DIMENSION` statement, the examination scoring program, and the `DISPLAY` command actually are examples of external specifications. They are not, however, completely realistic examples; for instance, an actual external specification for the scoring program would include a precise description of the format of the reports. (Note: Since we discussed function testing in Chapter 4, we present no examples of function tests in this section.)

Many of the guidelines we provided in Chapter 2 also are particularly pertinent to function testing. In particular, keep track of which functions have exhibited the greatest number of errors; this information is valuable because it tells you that these functions probably also contain the preponderance of as-yet undetected errors. Also, remember to focus a sufficient amount of attention on invalid and unexpected input conditions. (Recall that the definition of the expected result is a vital part of a test case.) Finally, as always, keep in mind that the purpose of the function test is to *expose errors and discrepancies* with the specification, not to demonstrate that the program matches its external specification.

System Testing

System testing is the most misunderstood and most difficult testing process. System testing is *not* a process of testing the functions of the complete system or program, because this would be redundant with the process of function testing. Rather, as shown in Figure 6.3, system testing has a

particular purpose: to compare the system or program to its original objectives. Given this purpose, consider these two implications:

1. System testing is not limited to systems. If the product is a program, system testing is the process of attempting to demonstrate how the program, as a whole, fails to meet its objectives.
2. System testing, by definition, is impossible if there is no set of written, measurable objectives for the product.

In looking for discrepancies between the program and its objectives, focus on translation errors made during the process of designing the external specification. This makes the system test a vital test process, because in terms of the product, the number of errors made, and the severity of those errors, this step in the development cycle usually is the most error prone.

It also implies that, unlike the function test, the external specification cannot be used as the basis for deriving the system test cases, since this would subvert the purpose of the system test. On the other hand, the objectives document cannot be used by itself to formulate test cases, since it does not, by definition, contain precise descriptions of the program's external interfaces. We solve this dilemma by using the program's user documentation or publications—design the system test by analyzing the objectives; formulate test cases by analyzing the user documentation. This has the useful side effect of comparing the program to its objectives and to the user documentation, as well as comparing the user documentation to the objectives, as shown in Figure 6.4.

Figure 6.4 illustrates why system testing is the most difficult testing process. The leftmost arrow in the figure, comparing the program to its objectives, is the central purpose of the system test, but there are no known test-case design methodologies. The reason for this is that objectives state what a program should do and how well the program should do it, but they do not state the representation of the program's functions. For instance, the objectives for the *DISPLAY* command specified in Chapter 4 might have read as follows:

> A command will be provided to view, from a terminal, the contents of main storage locations. Its syntax should be consistent with the syntax of all other system commands. The user should be able to specify

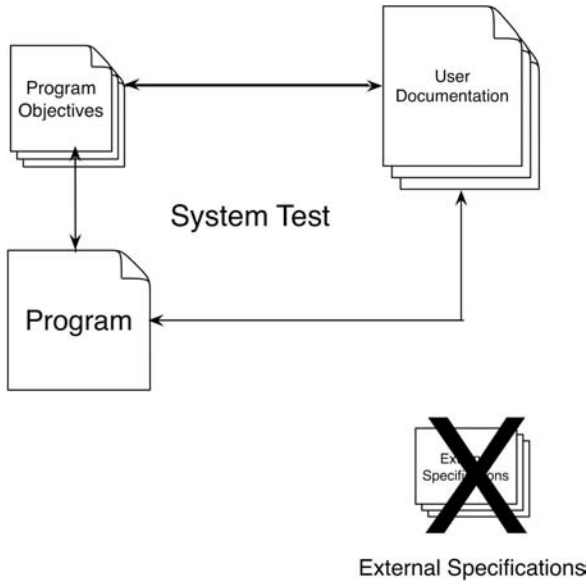

FIGURE 6.4 The System Test.

a range of locations, via an address range or an address and a count. Sensible defaults should be provided for command operands.

Output should be displayed as multiple lines of multiple words (in hexadecimal), with spacing between the words. Each line should contain the address of the first word of that line. The command is a "trivial" command, meaning that under reasonable system loads, it should begin displaying output within two seconds, and there should be no observable delay time between output lines. A programming error in the command processor should, at the worst, cause the command to fail; the system and the user's session must not be affected. The command processor should have no more than one user-detected error after the system is put into production.

Given the statement of objectives, there is no identifiable methodology that would yield a set of test cases, other than the vague but useful guideline of writing test cases to attempt to show that the program is inconsistent with each sentence in the objectives statement. Hence, a different approach to test-case design is taken here: Rather than describing a methodology, distinct categories of system test cases are discussed. Because of the absence of a methodology, system testing requires a substantial

amount of creativity; in fact, the design of good system test cases requires more creativity, intelligence, and experience than are required to design the system or program itself.

Table 6.1 lists 15 categories of test cases, along with a brief description. We discuss the categories in turn here. We don't claim that all 15 categories apply to every program, but to avoid overlooking something, we recommend that you explore all of them when designing test cases.

TABLE 6.1 15 Categories of Test Cases

Category	Description
Facility	Ensure that the functionality in the objectives is implemented.
Volume	Subject the program to abnormally large volumes of data to process.
Stress	Subject the program to abnormally large loads, generally concurrent processing.
Usability	Determine how well the end user can interact with the program.
Security	Try to subvert the program's security measures.
Performance	Determine whether the program meets response and throughput requirements.
Storage	Ensure the program correctly manages its storage needs, both system and physical.
Configuration	Check that the program performs adequately on the recommended configurations.
Compatibility/ Conversion	Determine whether new versions of the program are compatible with previous releases.
Installation	Ensure the installation methods work on all supported platforms.
Reliability	Determine whether the program meets reliability specifications such as uptime and MTBF.
Recovery	Test whether the system's recovery facilities work as designed.
Serviceability/ Maintenance	Determine whether the application correctly provides mechanisms to yield data on events requiring technical support.
Documentation	Validate the accuracy of all user documentation.
Procedure	Determine the accuracy of special procedures required to use or maintain the program.

Facility Testing

The most obvious type of system testing is to determine whether each facility (or function; but the word "function" is not used here to avoid confusing this with function testing) mentioned in the objectives was actually implemented. The procedure is to scan the objectives sentence by sentence, and when a sentence specifies a *what* (e.g., "syntax should be consistent . . . ," "user should be able to specify a range of locations . . ."), determine that the program satisfies the "what." This type of testing often can be performed without a computer; a mental comparison of the objectives with the user documentation is sometimes sufficient. Nonetheless, a checklist is helpful to ensure that you mentally verify the same objectives the next time you perform the test.

Volume Testing

A second type of system testing is to subject the program to heavy volumes of data. For instance, a compiler could be fed an absurdly large source program to compile. A linkage editor might be fed a program containing thousands of modules. An electronic circuit simulator could be given a circuit containing millions of components. An operating system's job queue might be filled to capacity. If a program is supposed to handle files spanning multiple volumes, enough data is created to cause the program to switch from one volume to another. In other words, the purpose of volume testing is to show that the program cannot handle the volume of data specified in its objectives.

Obviously, volume testing can require significant resources, therefore, in terms of machine and people time, you shouldn't go overboard. Still, every program must be exposed to at least a few volume tests.

Stress Testing

Stress testing subjects the program to heavy loads, or stresses. This should not be confused with volume testing; a heavy stress is a *peak volume of data, or activity, encountered over a short span of time.* An analogy would be evaluating a typist: A volume test would determine whether the typist could cope with a draft of a large report; a stress test would determine whether the typist could type at a rate of 50 words per minute.

Because stress testing involves an element of time, it is not applicable to many programs—for example, a compiler or a batch-processing payroll program. It is applicable, however, to programs that operate under varying loads, or interactive, real-time, and process control programs. If an air traffic control system is supposed to keep track of up to 200 planes in its sector, you could stress-test it by simulating the presence of 200 planes. Since there is nothing to physically keep a 201st plane from entering the sector, a further stress test would explore the system's reaction to this unexpected plane. An additional stress test might simulate the *simultaneous* entry of a large number of planes into the sector.

If an operating system is supposed to support a maximum of 15 concurrent jobs, the system could be stressed by attempting to run 15 jobs simultaneously. You could stress a pilot training aircraft simulator by determining the system's reaction to a trainee who forces the rudder left, pulls back on the throttle, lowers the flaps, lifts the nose, lowers the landing gear, turns on the landing lights, and banks left, all at the same time. (Such test cases might require a four-handed pilot or, realistically, two test specialists in the cockpit.) You might stress-test a process control system by causing all of the monitored processes to generate signals simultaneously, or a telephone switching system by routing to it a large number of simultaneous phone calls.

Web-based applications are common subjects of stress testing. Here, you want to ensure that your application, and hardware, can handle a target volume of concurrent users. You could argue that you may have millions of people accessing the site at one time, but that is not realistic. You need to define your audience then design a stress test to represent the maximum number of users you think will use your site. (Chapter 10 provides more information on testing Web-based applications.)

Similarly, you could stress a mobile device application—a mobile phone operating system, for example—by launching multiple applications that run and stay resident, then making or receiving one or more telephone calls. You could launch a GPS navigation program, an application that uses CPU and radio frequency (RF) resources almost continuously, then attempt to use other applications or engage telephone calls. (Chapter 11 discusses testing mobile applications in more detail.)

Although many stress tests do represent conditions that the program likely will experience during its operation, others may truly represent "never will occur" situations; but this does not imply that these tests are

not useful. If these impossible conditions detect errors, the test is valuable because it is likely that the same errors might also occur in realistic, less stressful situations.

Usability Testing

Another important test case area is usability, or user testing. Although this testing technique is nearly 30 years old, it has become more important with the advent of more GUI-based software and the deep penetration of computer hardware and software into all aspects of our society. By tasking the ultimate end user of an application with testing the software in a real-world environment, potential problems can be discovered that even the most aggressive automated testing routing likely wouldn't find. This area of software testing is so important we will cover it further in the next chapter.

Security Testing

In response to society's growing concern about privacy, many programs now have specific security objectives. Security testing is the process of attempting to devise test cases that subvert the program's security checks. For example, you could try to formulate test cases that get around an operating system's memory protection mechanism. Similarly, you could try to subvert a database system's data security mechanisms. One way to devise such test cases is to study known security problems in similar systems and generate test cases that attempt to demonstrate comparable problems in the system you are testing. For example, published sources in magazines, chat rooms, or newsgroups frequently cover known bugs in operating systems or other software systems. By searching for security holes in existing programs that provide services similar to the one you are testing, you can devise test cases to determine whether your program suffers from the same kind of problems.

Web-based applications often need a higher level of security testing than do most applications. This is especially true of e-commerce sites. Although sufficient technology, namely encryption, exists to allow customers to complete transactions securely over the Internet, you should not rely on the mere application of technology to ensure safety. In addition, you will need to convince your customer base that your application is safe,

or you risk losing customers. Again, Chapter 10 provides more information on security testing in Internet-based applications.

Performance Testing

Many programs have specific performance or efficiency objectives, stating such properties as response times and throughput rates under certain workload and configuration conditions. Again, since the purpose of a system test is to demonstrate that the program does not meet its objectives, test cases must be designed to show that the program does not satisfy its performance objectives.

Storage Testing

Similarly, programs occasionally have storage objectives that state, for example, the amount of system memory the program uses and the size of temporary or log files. You need to verify that your program can control its use of system memory so it does not negatively impact other processes running on the host. The same holds for physical files on the file system. Filling a disk drive can cause significant downtime. You should design test cases to show that these storage objectives have not been met.

Configuration Testing

Programs such as operating systems, database management systems, and messaging programs support a variety of hardware configurations, including various types and numbers of I/O devices and communications lines, or different memory sizes. Often, the number of possible configurations is too large to test each one, but at the least, you should test the program with each type of hardware device and with the minimum and maximum configuration. If the program itself can be configured to omit program components, or if the program can run on different computers, each possible configuration of the program should be tested.

Today, many programs are designed for multiple operating systems. Thus, when testing such a program, you should do so on all of the operating systems for which it was designed. Programs designed to execute within a Web browser require special attention, since there are numerous Web browsers available and they don't all function the same way. In

addition, the same Web browser will operate differently on different operating systems.

Compatibility/Conversion Testing

Most programs that are developed are not completely new; they often are replacements for some deficient system. As such, programs often have specific objectives concerning their compatibility with, and conversion procedures from, the existing system. Again, in testing the program against these objectives, the orientation of the test cases is to demonstrate that the compatibility objectives have not been met and that the conversion procedures do not work. Here you try to generate errors while moving data from one system to another. An example would be upgrading a database system. You want to ensure that the new release supports your existing data, just as you need to validate that a new version of a word processing application supports its previous document formats. Various methods exist to test this process; however, they are highly dependent on the database system you employ.

Installation Testing

Some types of software systems have complicated installation procedures. Testing the installation procedure is an important part of the system testing process. This is particularly true of an automated installation system that is part of the program package. A malfunctioning installation program could prevent the user from ever having a successful experience with the main system you are testing. A user's first experience is when he or she installs the application. If this phase performs poorly, then the user/customer may find another product, or have little confidence in the application's validity.

Reliability Testing

Of course, the goal of all types of testing is the improvement of the program reliability, but if the program's objectives contain specific statements about reliability, specific reliability tests might be devised. Testing reliability objectives can be difficult. For example, a modern online system such as a corporate wide area network (WAN) or an Internet service provider (ISP) generally has a targeted uptime of 99.97 percent over the life of the

system. There is no known way that you could test this objective within a test period of months or even years. Today's critical software systems have even higher reliability standards, and today's hardware conceivably should support these objectives. You potentially can test programs or systems with more modest mean time between failures (MTBF) objectives or reasonable (in terms of testing) operational error objectives.

An MTBF of no more than 20 hours, or an objective that a program should experience no more than 12 unique errors after it is placed into production, for example, presents testing possibilities, particularly for statistical, program-proving, or model-based testing methodologies. These methods are beyond the scope of this book, but the technical literature (online and otherwise) offers ample guidance in this area. For example, if this area of program testing is of interest to you, research the concept of inductive assertions. The goal of this method is the development of a set of theorems about the program in question, the proof of which guarantees the absence of errors in the program. The method begins by writing assertions about the program's input conditions and correct results. The assertions are expressed symbolically in a formal logic system, usually the first-order predicate calculus. You then locate each loop in the program and, for each loop, write an assertion stating the invariant (always true) conditions at an arbitrary point in the loop. The program now has been partitioned into a fixed number of fixed-length paths (all possible paths between a pair of assertions). For each path, you then take the semantics of the intervening program statements to modify the assertion, and eventually reach the end of the path. At this point, two assertions exist at the end of the path: the original one and the one derived from the assertion at the opposite end. You then write a theorem stating that the original assertion implies the derived assertion, and attempt to prove the theorem. If the theorems can be proved, you could assume the program is error free—as long as the program eventually terminates. A separate proof is required to show that the program will always eventually terminate.

As complex as this sort of software proving or prediction sounds, reliability testing and, indeed, the concept of software reliability engineering (SRE) are with us today and are increasingly important for systems that must maintain very high uptimes. To illustrate this point, examine Table 6.2 to see the number of hours per year a system must be up to support various uptime requirements. These values should indicate the need for SRE.

TABLE 6.2 Hours per Year for Various Uptime Requirements

Uptime Percent Requirements	Operational Hours per Year
100	8760.0
99.9	8751.2
98	8584.8
97	8497.2
96	8409.6
95	8322.0

Recovery Testing

Programs such as operating systems, database management systems, and teleprocessing programs often have recovery objectives that state how the system is to recover from programming errors, hardware failures, and data errors. One objective of the system test is to show that these recovery functions do not work correctly. Programming errors can be purposely injected into a system to determine whether it can recover from them. Hardware failures such as memory parity errors or I/O device errors can be simulated. Data errors such as noise on a communications line or an invalid pointer in a database can be created purposely or simulated to analyze the system's reaction.

One design goal of such systems is to minimize the mean time to recovery (MTTR). Downtime often causes a company to lose revenue because the system is inoperable. One testing objective is to show that the system fails to meet the service-level agreement for MTTR. Often, the MTTR will have an upper and lower boundary, so your test cases should reflect these bounds.

Serviceability/Maintenance Testing

The program also may have objectives for its serviceability or maintainability characteristics. All objectives of this sort must be tested. Such objectives might define the service aids to be provided with the system, including storage dump programs or diagnostics, the mean time to debug an apparent problem, the maintenance procedures, and the quality of internal logic documentation.

Documentation Testing

As we illustrated in Figure 6.4, the system test also is concerned with the accuracy of the user documentation. The principal way of accomplishing this test is to use the documentation to determine the representation of the prior system test cases. That is, once a particular stress case is devised, you would use the documentation as a guide for writing the actual test case. Also, the user documentation itself should be the subject of an inspection (similar to the concept of the code inspection in Chapter 3), to check it for accuracy and clarity. Any examples illustrated in the documentation should be encoded into test cases and fed to the program.

Procedure Testing

Finally, many programs are parts of larger, not completely automated systems involving procedures people perform. Any prescribed human procedures, such as those for the system operator, database administrator, or end user, should be tested during the system test.

For example, a database administrator should document procedures for backing up and recovering the database system. If possible, a person not associated with the administration of the database should test the procedures. However, a company must create the resources needed to adequately test the procedures. These resources often include hardware and additional software licensing.

Performing the System Test

One of the most vital considerations in implementing the system test is determining who should do it. To answer this in a negative way, (1) programmers should not perform a system test; and (2) of all the testing phases, this is the one that the organization responsible for developing the programs definitely should not perform.

The first point stems from the fact that a person performing a system test must be capable of thinking like an end user, which implies a thorough understanding of the attitudes and environment of the end user and of how the program will be used. Obviously, then, if feasible, a good testing candidate is one or more end users. However, because the typical end user will not have the ability or expertise to perform many of the

categories of tests described earlier, an ideal system test team might be composed of a few professional system test experts (people who spend their lives performing system tests), a representative end user or two, a human-factors engineer, and the key original analysts or designers of the program. Including the original designers does not violate principle 2 from Table 2.1, "Vital Program Testing Guidelines," recommending against testing your own program, since the program has probably passed through many hands since it was conceived. Therefore, the original designers do not have the troublesome psychological ties to the program that motivated this principle.

The second point stems from the fact that a system test is an "anything goes, no holds barred" activity. Again, the development organization has psychological ties to the program that are counter to this type of activity. Also, most development organizations are most interested in having the system test proceed as smoothly as possible and on schedule, hence are not truly motivated to demonstrate that the program does not meet its objectives. At the least, the system test should be performed by an independent group of people with few, if any, ties to the development organization.

Perhaps the most economical way of conducting a system test (economical in terms of finding the most errors with a given amount of money, or spending less money to find the same number of errors), is to subcontract the test to a separate company. We talk about this more in the last section of this chapter.

Acceptance Testing

Returning to the overall model of the development process shown in Figure 6.3, you can see that acceptance testing is the process of comparing the program to its initial requirements and the current needs of its end users. It is an unusual type of test in that it usually is performed by the program's customer or end user and normally is not considered the responsibility of the development organization. In the case of a contracted program, the contracting (user) organization performs the acceptance test by comparing the program's operation to the original contract. As is the case for other types of testing, the best way to do this is to devise test cases that attempt to show that the program does not meet the contract; if these test cases are unsuccessful, the program is accepted. In the

case of a program product, such as a computer manufacturer's operating system, or a software company's database system, the sensible customer first performs an acceptance test to determine whether the product satisfies its needs.

Although the ultimate acceptance test is, indeed, the responsibility of the customer or end user, the savvy developer will conduct user tests during the development cycle and prior to delivering the finished product to the end user or contract customer. See Chapter 7 for more information on user or usability testing.

Installation Testing

The remaining testing process in Figure 6.3 is the installation test. Its position in the figure is a bit unusual, since it is not related, as all of the other testing processes are, to specific phases in the design process. It is an unusual type of testing because its purpose is not to find software errors but to find errors that occur during the installation process.

Many events occur when installing software systems. A short list of examples includes the following:

- User must select a variety of options.
- Files and libraries must be allocated and loaded.
- Valid hardware configurations must be present.
- Programs may need network connectivity to connect to other programs.

The organization that produced the system should develop the installation tests, which should be delivered as part of the system, and run after the system is installed. Among other things, the test cases might check to ensure that a compatible set of options has been selected, that all parts of the system exist, that all files have been created and have the necessary contents, and that the hardware configuration is appropriate.

Test Planning and Control

If you consider that the testing of a large system could entail writing, executing, and verifying tens of thousands of test cases, handling thousands

of modules, repairing thousands of errors, and employing hundreds of people over a time span of a year or more, it is apparent that you are faced with an immense project management challenge in planning, monitoring, and controlling the testing process. In fact, the problem is so enormous that we could devote an entire book to just the management of software testing. The intent of this section is to summarize some of these considerations.

As mentioned in Chapter 2, the major mistake most often made in planning a testing process is the tacit assumption that no errors will be found. The obvious result of this mistake is that the planned resources (people, calendar time, and computer time) will be grossly underestimated, a notorious problem in the computing industry. Compounding the problem is the fact that the testing process falls at the end of the development cycle, meaning that resource changes are difficult. A second, perhaps more insidious problem is that the wrong definition of testing is being used, since it is difficult to see how someone using the correct definition of testing (the goal being to find errors) would plan a test using the assumption that no errors will be found.

As is the case for most undertakings, the plan is the crucial part of the management of the testing process. The components of a good test plan are as follows:

1. *Objectives.* The objectives of each testing phase must be defined.
2. *Completion criteria.* Criteria must be designed to specify when each testing phase will be judged to be complete. This matter is discussed in the next section.
3. *Schedules.* Calendar time schedules are needed for each phase. They should indicate when test cases will be designed, written, and executed. Some software methodologies such as Extreme Programming (discussed in Chapter 9) require that you design the test cases and unit tests before application coding begins.
4. *Responsibilities.* For each phase, the people who will design, write, execute, and verify test cases, and the people who will repair discovered errors, should be identified. And, because in large projects disputes inevitably arise over whether particular test results represent errors, an arbitrator should be identified.
5. *Test case libraries and standards.* In a large project, systematic methods of identifying, writing, and storing test cases are necessary.

6. *Tools.* The required test tools must be identified, including a plan for who will develop or acquire them, how they will be used, and when they will be needed.
7. *Computer time.* This is a plan for the amount of computer time needed for each testing phase. It would include servers used for compiling applications, if required; desktop machines required for installation testing; Web servers for Web-based applications; networked devices, if required; and so forth.
8. *Hardware configuration.* If special hardware configurations or devices are needed, a plan is required that describes the requirements, how they will be met, and when they will be needed.
9. *Integration.* Part of the test plan is a definition of how the program will be pieced together (e.g., incremental top-down testing). A system containing major subsystems or programs might be pieced together incrementally, using the top-down or bottom-up approach, for instance, but where the building blocks are programs or subsystems, rather than modules. If this is the case, a system integration plan is necessary. The system integration plan defines the order of integration, the functional capability of each version of the system, and responsibilities for producing "scaffolding," code that simulates the function of nonexistent components.
10. *Tracking procedures.* You must identify means to track various aspects of the testing progress, including the location of error-prone modules and estimation of progress with respect to the schedule, resources, and completion criteria.
11. *Debugging procedures.* You must define mechanisms for reporting detected errors, tracking the progress of corrections, and adding the corrections to the system. Schedules, responsibilities, tools, and computer time/resources also must be part of the debugging plan.
12. *Regression testing.* Regression testing is performed after making a functional improvement or repair to the program. Its purpose is to determine whether the change has regressed other aspects of the program. It usually is performed by rerunning some subset of the program's test cases. Regression testing is important because changes and error corrections tend to be much more error prone than the original program code (in much the same way that most typographical errors in newspapers are the result of last-minute editorial

changes, rather than changes in the original copy). A plan for regression testing—who, how, when—also is necessary.

Test Completion Criteria

One of the most difficult questions to answer when testing a program is determining when to stop, since there is no way of knowing if the error just detected is the last remaining error. In fact, in anything but a small program, it is unreasonable to expect that all errors will eventually be detected. Given this dilemma, and given the fact that economics dictate that testing must eventually terminate, you might wonder if the question has to be answered in a purely arbitrary way, or if there are some useful stopping criteria.

The completion criteria typically used in practice are both meaningless and counterproductive. The two most common criteria are these:

1. Stop when the scheduled time for testing expires.
2. Stop when all the test cases execute without detecting errors—that is, stop when the test cases are unsuccessful.

The first criterion is useless because you can satisfy it by doing absolutely nothing. It does not measure the quality of the testing. The second criterion is equally useless because it also is independent of the quality of the test cases. Furthermore, it is counterproductive because it subconsciously encourages you to write test cases that have a low probability of detecting errors.

As discussed in Chapter 2, humans are highly goal oriented. If you are told that you have finished a task when the test cases are unsuccessful, you will subconsciously write test cases that lead to this goal, avoiding the useful, high-yield, destructive test cases.

There are three categories of more useful criteria. The first category, but not the best, is to base completion on the use of specific test-case design methodologies. For instance, you might define the completion of module testing as the following:

The test cases are derived from (1) satisfying the multicondition-coverage criterion and (2) a boundary value analysis of the module

interface specification, and all resultant test cases are eventually unsuccessful.

You might define the function test as being complete when the following conditions are satisfied:

The test cases are derived from (1) cause-effect graphing, (2) boundary value analysis, and (3) error guessing, and all resultant test cases are eventually unsuccessful.

Although this type of criterion is superior to the two mentioned earlier, it has three problems. First, it is not helpful in a test phase in which specific methodologies are not available, such as the system test phase. Second, it is a subjective measurement, since there is no way to guarantee that a person has used a particular methodology, such as boundary value analysis, properly and rigorously. Third, rather than setting a goal and then letting the tester choose the best way of achieving it, it does the opposite; test-case-design methodologies are dictated, but no goal is given. Hence, this type of criterion is useful sometimes for some testing phases, but it should be applied only when the tester has proven his or her abilities in the past in applying the test-case design methodologies successfully.

The second category of criteria—perhaps the most valuable one—is to state the completion requirements in positive terms. Since the goal of testing is to find errors, why not make the completion criterion the detection of some predefined number of errors? For instance, you might state that a module test of a particular module is not complete until three errors have been discovered. Perhaps the completion criterion for a system test should be defined as the detection and repair of 70 errors, or an elapsed time of three months, whichever comes later.

Notice that, although this type of criterion reinforces the definition of testing, it does have two problems, both of which are surmountable. One problem is determining how to obtain the number of errors to be detected. Obtaining this number requires the following three estimates:

1. An estimate of the total number of errors in the program.
2. An estimate of what percentage of these errors can feasibly be found through testing.

3. An estimate of what fraction of the errors originated in particular design processes, and during which testing phases these errors are likely to be detected.

You can get a rough estimate of the total number of errors in several ways. One method is to obtain them through experience with previous programs. Also, a variety of predictive modules exist. Some of these require you to test the program for some period of time, record the elapsed times between the detection of successive errors, and insert these times into parameters in a formula. Other modules involve the seeding of known, but unpublicized, errors into the program, testing the program for a while, and then examining the ratio of detected seeded errors to detected unseeded errors. Another model employs two independent test teams whose members test for a while, examine the errors found by each and the errors detected in common by both teams, and use these parameters to estimate the total number of errors. Another gross method to obtain this estimate is to use industrywide averages. For instance, the number of errors that exist in typical programs at the time that coding is completed (before a code walkthrough or inspection is employed) is approximately 4 to 8 errors per 100 program statements.

The second estimate from the preceding list (the percentage of errors that can be feasibly found through testing) involves a somewhat arbitrary guess, taking into consideration the nature of the program and the consequences of undetected errors.

Given the current paucity of information about how and when errors are made, the third estimate is the most difficult. The data that exist indicate that in large programs, approximately 40 percent of the errors are coding and logic design mistakes, and that the remainder are generated in the earlier design processes.

To use this criterion, you must develop your own estimates that are pertinent to the program at hand. A simple example is presented here. Assume we are about to begin testing a 10,000-statement program, that the number of errors remaining after code inspections are performed is estimated at 5 per 100 statements, and we establish, as an objective the detection of 98 percent of the coding and logic design errors and 95 percent of the design errors. The total number of errors is thus estimated at 500. Of the 500 errors, we assume that 200 are coding and logic design errors and

TABLE 6.3 Hypothetical Estimate of When the Errors Might Be Found

	Coding and Logic Design Errors	Design Errors
Module test	65%	0%
Function test	30%	60%
System test	3%	35%
Total	98%	95%

300 are design flaws. Hence, the goal is to find 196 coding and logic design errors and 285 design errors. A plausible estimate of when the errors are likely to be detected is shown in Table 6.3.

If we have scheduled four months for function testing and three months for system testing, the following three completion criteria might be established:

1. Module testing is complete when 130 errors are found and corrected (65 percent of the estimated 200 coding and logic design errors).
2. Function testing is complete when 240 errors (30 percent of 200 plus 60 percent of 300) are found and corrected, or when four months of function testing have been completed, whichever occurs later. The reason for the second clause is that if we find 240 errors quickly, it is probably an indication that we have underestimated the total number of errors and thus should not stop function testing early.
3. System testing is complete when 111 errors are found and corrected, or when three months of system testing have been completed, whichever occurs later.

The other obvious problem with this type of criterion is one of overestimation. What if, in the preceding example, there are fewer than 240 errors remaining when function testing starts? Based on the criterion, we could never complete the function test phase.

This is a strange problem if you think about it: We do not have enough errors; the program is too good. You could label it as not a problem because it is the kind of problem a lot of people would love to have. If it does occur, a bit of common sense can solve it. If we cannot find 240 errors in four months, the project manager can employ an outsider to analyze the

test cases to judge whether the problem is (1) inadequate test cases or (2) excellent test cases but a lack of errors to detect.

The third type of completion criterion is an easy one on the surface, but it involves a lot of judgment and intuition. It requires you to plot the number of errors found per unit time during the test phase. By examining the shape of the curve, you can often determine whether to continue the test phase or end it and begin the next test phase.

Suppose a program is being function-tested and the number of errors found per week is being plotted. If, in the seventh week, the curve is the top one of Figure 6.5, it would be imprudent to stop the function test, even if we had reached our criterion for the number of errors to be found. Since in the seventh week we still seem to be in high gear (finding many errors), the wisest decision (remembering that our goal is to find errors) is to continue function testing, designing additional test cases if necessary.

On the other hand, suppose the curve is the bottom one in Figure 6.5. The error-detection efficiency has dropped significantly, implying that we have perhaps picked the function test bone clean and that perhaps the best move is to terminate function testing and begin a new type of testing (a system test, perhaps). Of course, we must also consider other factors, such as whether the drop in error-detection efficiency was due to a lack of computer time or exhaustion of the available test cases.

Figure 6.6 is an illustration of what happens when you fail to plot the number of errors being detected. The graph represents three testing phases of an extremely large software system. An obvious conclusion is that the project should not have switched to a different testing phase after period 6. During period 6, the error-detection rate was good (to a tester, the higher the rate, the better), but switching to a second phase at this point caused the error-detection rate to drop significantly.

The best completion criterion is probably a combination of the three types just discussed. For the module test, particularly because most projects do not formally track detected errors during this phase, the best completion criterion is probably the first. You should request that a particular set of test-case design methodologies be used. For the function and system test phases, the completion rule might be to stop when a predefined number of errors are detected or when the scheduled time has elapsed, whichever comes later, but provided that an analysis of the errors-versus-time graph indicates that the test has become unproductive.

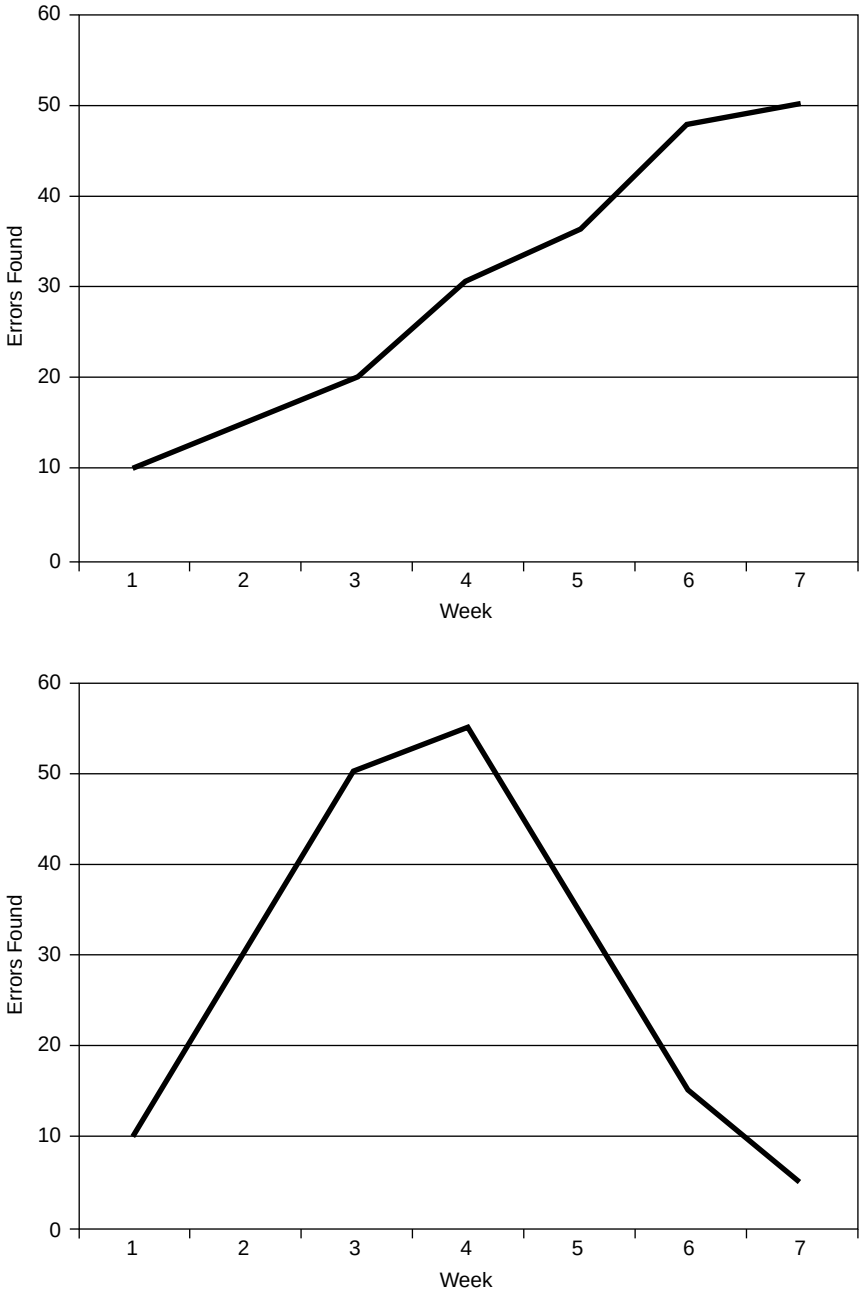

FIGURE 6.5 Estimating Completion by Plotting Errors Detected by Unit Time.

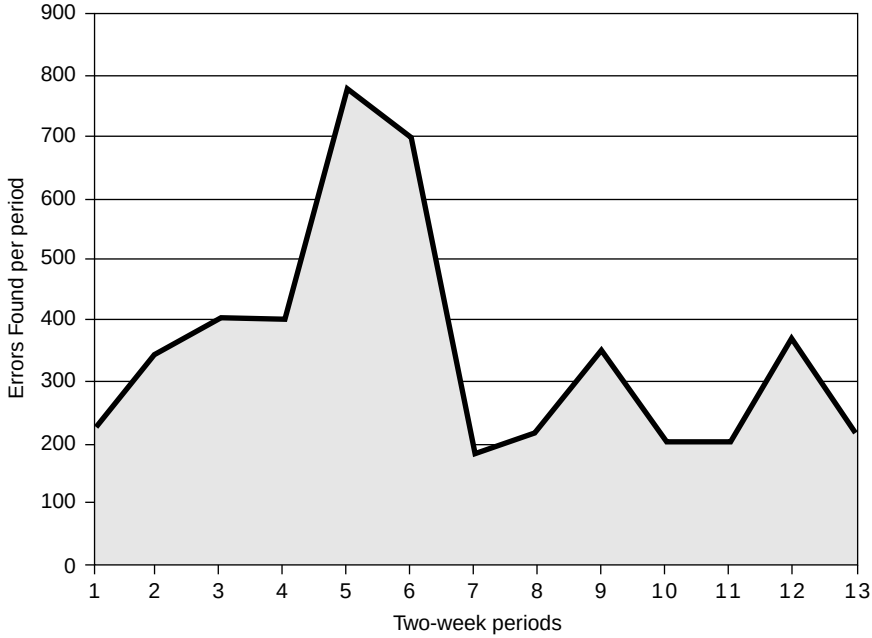

FIGURE 6.6 Postmortem Study of the Testing Processes of a Large Project.

The Independent Test Agency

Earlier in this chapter and in Chapter 2, we emphasized that an organization should avoid attempting to test its own programs. Our reasoning is that the organization responsible for developing a program has difficulty in objectively testing the same program. The test organization should be as far removed as possible, in terms of the structure of the company, from the development organization. In fact, it is desirable that the test organization not be part of the same company, for if it is, it is still influenced by the same management pressures influencing the development organization.

One way to avoid this conflict is to hire a separate company for software testing. This is a good idea, whether the company that designed the system and will use it developed the system, or whether a third-party developer produced the system. The advantages usually noted are increased motivation in the testing process, a healthy competition with the development organization, removal of the testing process from under the management

control of the development organization, and the advantages of specialized knowledge that the independent test agency brings to bear on the problem.

Summary

Higher-order testing could be considered the next step. We have discussed and advocated the concept of module testing—using various techniques to test software components, the building blocks that combine to form the finished product. With individual components tested and debugged, it is time to see how well they work together.

Higher-order testing is important for all software products, but it becomes increasingly important as the size of the project increases. It stands to reason that the more modules and the more lines of code a project contains, the more opportunity exists for coding or even design errors.

Function testing attempts to uncover design errors, that is, discrepancies between the finished program and its external specifications—a precise description of the program's behavior from the end-user's perspective.

The system test, on the other hand, tests the relationship between the software and its original objectives. System testing is designed to uncover errors made during the process of translating program objectives into the external specification and ultimately into lines of code. It is this translation step where errors have the most far-reaching effects; likewise, it is the stage in the development process that is most error prone. Perhaps the most difficult part of system testing is designing the test cases. In general you want to focus on main categories of testing, then get really creative in testing these categories. Table 6.1 summarizes 15 categories we detailed in this chapter that can guide your system testing efforts.

Make no mistake, higher-order testing certainly is an important part of thorough software testing, but it also can become a daunting process, especially for very large systems, such as an operating system. The key to success is consistent and well-planned test planning. We introduce this topic in this chapter, but if you are managing the testing of large systems, more thought and planning will be required. One approach to handling this issue is to hire an outside company for testing or for test management.

In Chapter 7 we expand on one important aspect of higher-order testing: user or usability testing.

7 Usability (User) Testing

An important category of system test cases is one that attempts to find human-factor, or usability, problems. When the first edition of this book was published, the computing industry mostly ignored the human factors associated with computer software. Developers gave little attention to how humans interacted with their software. That is not to say that there were no developers testing applications at the user level. In the early 1980s, some—including developers at the Xerox Palo Alto Research Center (PARC), for example—were conducting user-based software testing.

By 1987 or 1988, the three of us had become intimately involved in usability testing of early personal computer hardware and software, when we contracted with computer manufacturers to test and review their new desktop computers prior to release to the public. Over perhaps two years, this prerelease testing prevented potential usability problems with new hardware and software designs. These early computer manufacturers obviously were convinced that the time and expense required for this level of user testing resulted in real marketing and financial advantages.

Usability Testing Basics

Today's software systems—particularly those designed for a mass, commercial market—generally have undergone extensive human-factor studies, and modern programs, of course, benefit from the thousands of programs and systems that have gone before. Nevertheless, an analysis of human

factors is still a highly subjective matter. Here's our list of questions you might ask to derive testing considerations:

1. Has each user interface been tailored to the intelligence, educational background, and environmental pressures of the end user?
2. Are the outputs of the program meaningful, noninsulting to the user, and devoid of computer gibberish?
3. Are the error diagnostics, such as error messages, straightforward, or does the user need a PhD in computer science to comprehend them? For instance, does the program produce such messages as `IEK022A OPEN ERROR ON FILE 'SYSIN' ABEND CODE=102`? Messages such as these weren't all that uncommon in software systems of the 1970s and 1980s. Mass-market systems do better today in this regard, but users still will encounter unhelpful messages such as, "An unknown error has occurred," or "This program has encountered an error and must be restarted."

   Programs you design yourself are under your control and should not be plagued with such useless messages. Even if you didn't design the program, if you are on the testing team, you can push for improvements in this area of the human interface.
4. Does the total set of user interfaces exhibit considerable conceptual integrity, an underlying consistency, and uniformity of syntax, conventions, semantics, format, style, and abbreviations?
5. Where accuracy is vital, such as in an online banking system, is sufficient redundancy present in the input? For example, such a system should ask for an account number, a customer name, and a personal identification number (PIN) to verify that the proper person is accessing account information.
6. Does the system contain an excessive number of options, or options that are unlikely to be used? One trend in modern software is to present to users only those menu choices they are most likely to use, based on software testing and design considerations. Then a well-designed program can learn from individual users and begin to present those menu items that they frequently access. Even with such an intelligent menu system, successful programs still must be designed so that accessing the various options is logical and intuitive.
7. Does the system return some type of immediate acknowledgment to all inputs? Where a mouse click is the input, for example, the chosen

item can change color, or a button object can depress or be presented in a raised format. If the user is expected to choose from a list, the selected number should be presented on the screen when the choice is made. Moreover, if the selected action requires some processing time—which is frequently the case when the software is accessing a remote system—then a message should be displayed informing the user of what is going on. This level of testing sometimes is referred to as *component testing*, whereby interactive software components are tested for reasonable selection and user feedback.

8. Is the program easy to use? For example, is the input case-sensitive without making this fact clear to the user? Also, if a program requires navigation through a series of menus or options, is it clear how to return to the main menu? Can the user easily move up or down one level?
9. Is the design conducive to user accuracy? One test would be an analysis of how many errors each user makes during data entry or when choosing program options. Were these errors merely an inconvenience—errors the user was able to correct—or did an incorrect choice or action cause some kind of application failure?
10. Are the user actions easily repeated in later sessions? In other words, is the software design conducive to the user learning how to be more efficient in using the system?
11. Did the user feel confident while navigating the various paths or menu choices? A subjective evaluation might be the user response to using the application. At the end of the session did the user feel stressed by or satisfied with the outcome? Would the user be likely to choose this system for his or her own use, or recommend it to someone else?
12. Did the software live up to its design promise? Finally, usability testing should include an evaluation of the software specifications versus the actual operation. From the user perspective—real people using the software in a real-world environment—did the software perform according to its specifications?

Usability or user-based testing basically is a black-box testing technique. Recall from our discussion in Chapter 2 that black-box testing concentrates on finding situations in which the program does not behave according to specifications. In a black-box scenario you are not concerned

with the internal workings of the software, or even with understanding program structure. Presented this way, usability testing obviously is an important part of any development process. If users perceive, because of improper design, a cumbersome user interface, or specifications missed or ignored, that a given application does not perform according to its specifications, the development process has failed. User testing should uncover problems from design flaws to software ergonomics mistakes.

Usability Testing Process

It should be obvious from our list of items to test that usability testing is more than simply seeking user opinions or high-level reactions to a software application. When the errors have been found and corrected, and an application is ready for release or for sale, focus groups can be used to elicit opinions from users or potential purchasers. This is marketing and focusing. Usability testing occurs earlier in the process and is much more involved.

Any usability test should begin with a plan. (Review our vital software testing guidelines in Chapter 2, Table 2.1.) You should establish practical, real-world, repeatable exercises for each user to conduct. Design these testing scenarios to present the user with all aspects of the software, perhaps in various or random order. For example, among the processes you might test in a customer tracking application are:

- Locate an individual customer record and modify it.
- Locate a company record and modify it.
- Create a new company record.
- Delete a company record.
- Generate a list of all companies of a certain type.
- Print this list.
- Export a selected list of contacts to a text file or spreadsheet format.
- Import a text file or spreadsheet file of contacts from another application.
- Add a photograph to one or more records.
- Create and save a custom report.
- Customize the menu structure.

During each phase of the test, have observers document the user experience as they perform each task. When the test is complete, conduct an

interview with the user or provide a written questionnaire to document other aspects of the user's experience, such as his or her perception of usage versus specification.

In addition, write down detailed instructions for user tests, to ensure that each user starts with the same information, presented in the same way. Otherwise, you risk coloring some of the tests if some users receive different instructions.

Test User Selection

A complete usability testing protocol usually involves multiple tests from the same users, as well as tests from multiple users. Why multiple tests from the same users? One area we want to test is *user recall*, that is, how much of what a user learns about software operation is retained from session to session. Any new system presented to users for the first time will require some time to learn, but if the design for a particular application is consistent with the industry or technology with which the target user community is familiar, the learning process should be fairly quick.

A user already familiar with computer-based engineering design, for example, would expect any new software in this same industry to follow certain conventions of terminology, menu design, and perhaps even color, shading, and font usage. Certainly, a developer may stray from these conventions purposefully to achieve perceived operational improvements, but if the design goes too far afield from industry standards and expectations, the software will take longer for new users to learn; in fact, user acceptance may be so slow as to cause the application to be a commercial failure. If the application is developed for a single client, such differences may result in the client rejecting the design or requiring a complete user interface redesign. Either result is a costly developer mistake.

Therefore, software targeted for a specific end-user type or industry should be tested by what could be described as expert users, people already familiar with this class of application in a real-world environment. In contrast, software with a more general target market—mobile device software, for example, or general-purpose Web pages—might better be tested by users selected randomly. (Such test user selection sometimes is referred to as *hallway testing* or *hallway intercept testing*, meaning that users chosen for software testing are selected at random from folk passing by in the hallway.)

How Many Users Do You Need?

When designing a usability test plan, the question "How many testers do I need?" will come to the forefront. Hiring usability testers is often overlooked in the development process, and can add an unexpected and expensive cost to the project. You need to find the right number of testers who can identify the most errors for the least amount of capital investment.

Intuitively, you may think that the more testers you use the better. After all, if you have enough evaluators testing your product, then all the errors should be found. First, as mentioned, this is expensive. Second, it can become a logistics nightmare. Finally, it is unlikely that you can ever detect 100 percent of your application's usability problems.

Fortunately, significant research on usability has been conducted during the last 15 years. Based on the work of Jakob Nielsen, a usability testing expert, you may need fewer testers than you think. Nielsen's research found that the number of usability problems found in testing is:

$$E = 100 \times (1 - (1 - L)^{\wedge} n)$$

where: E = percent of errors found
n = number of testers
L = percent of usability problems found by a tester

Using the equation with L = 31 percent, a reasonable value Nielsen also gleaned from his research, produces the graph shown in Figure 7.1.

Examining the graph reveals a few interesting points. First, as we intuitively know, it will never be possible to detect all of the usability errors in the application. It's not theoretically possible, because the curve only converges on 100 percent; it never actually reaches it. Second, you only need a small number of testers. The graph shows that approximately 83 percent of the errors are detected by only 5 testers.

From a project manager's point of view, this is refreshing news. No longer do you need to incur the cost and complexity of working with a large group of testers to check your application. Instead, you can focus on designing, executing, and analyzing your tests—putting your effort and money into what will make the most difference.

Also with fewer testers, you have less analysis to do, so you can quickly implement changes to the application and the testing methodology; then

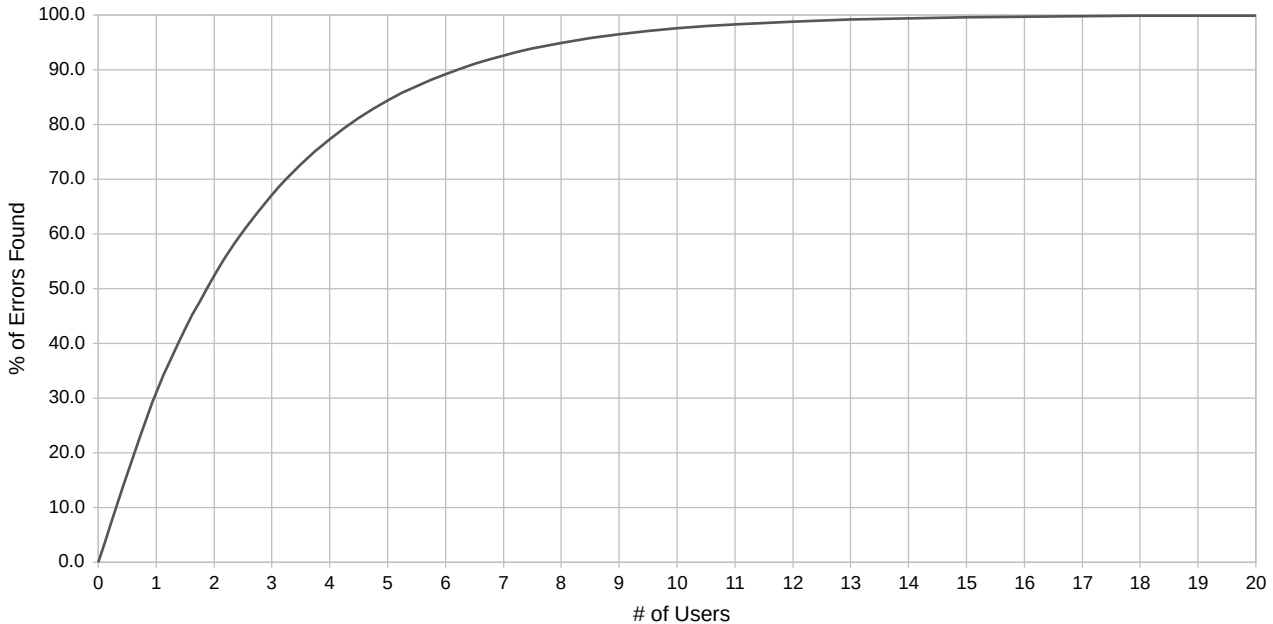

FIGURE 7.1 Percent Errors Found Versus Number of Users.

test again with a new group of testers. In this iterative fashion you can ensure that you catch most problems at minimal cost and time.

Nielsen's research was conducted in the early 1990s while he was a systems analyst at Sun Microsystems. On the one hand, his data and approach to usability testing provides concrete guidance to those of us involved in software design. On the other hand, since usability testing has become more important and commonplace, and more evidence has been gathered from practical testing and better formulaic analysis, some researchers have come to question Nielsen's firm statements that three to five users should be enough.

Nielsen himself cautions that the precise number of testers depends on economic considerations (how many testers will your budget support) and on the type of system you are testing. Critical systems such as navigation applications, banking or other financial software, or security related programs will, per force, require closer user scrutiny than less-critical software.

Among the considerations important to developers who are designing a usability testing program are whether the number of users and their individual orientations represent sufficiently the total population of potential users. In addition, as Nielsen notes, some programs are more complex than others, meaning that detecting a significantly large percentage of errors will be more difficult. And, since different users, because of their backgrounds and experiences, are likely to detect different types of errors, an individual testing situation may dictate a larger number of testers.

As with any testing methodology, it is up to the developers and project administrators to design the tests, present a reasonable budget, evaluate interim results, and conduct regressive tests as appropriate to the software system, the overall project, and the client.

Data-Gathering Methods

Test administrators or observers can gather test results in several ways. Videotaping a user test and using a *think-aloud protocol* can provide excellent data on software usability and user perceptions about the application. A think-aloud protocol involves users speaking aloud their thoughts and observations while they are performing the assigned software testing tasks. Using this process, the test participants describe out loud their task, what they are thinking about the task, and/or whatever else comes to their mind as they move through the testing scenario. Even when using think-aloud

protocol testing, developers may want to follow up with participants after the test to get posttest comments, feelings, and observations. Taken together, these two levels of user thoughts and comments can provide valuable feedback to developers for software corrections or improvements.

A disadvantage to the think-aloud process, where videotaping or observers are involved, is the possibility that the user experience will be clouded or modified by the unnatural user environment. Developers also may wish to conduct *remote user testing*, whereby the application is installed at the testing user's business where the software may ultimately be applied. Remote testing has the advantage of placing the user in a familiar environment, one in which the final application likely would be used, thus removing the potential for external influences modifying test results. Of course, the disadvantage is that developers may not receive feedback as detailed as would be possible with a think-aloud protocol.

Nevertheless, in a remote testing environment, accurate user data still can be gathered. Additional software can be installed with the application to be tested to gather user keystrokes and to capture time required for the user to complete each assigned task. This requires additional development time (and more software), but the results of such tests can be enlightening and very detailed.

In the absence of timing or keystroke capture software, testing users can be tasked with writing down the start and end times of each assigned task, along with brief one-word or short-phrase comments during the process. Posttest questionnaires or interviews can help users recall their thoughts and opinions about the software.

A sophisticated but potentially useful data-gathering protocol is *eye tracking*. When we read a printed page, view a graphical presentation, or interact with a computer screen, our eyes move over the scanned material in particular patterns. Research data gathered on eye movement over more than 100 years shows that eye movement—particularly how long an observer pauses on certain visual elements—reflects at least to some degree the thought processes of the observer. Tracking this eye movement, which can be done with video systems and other technologies, shows researchers which visual elements attract the observers attention, in what order, and for how long. Such data is potentially useful in determining the efficiency of software screens presented to users.

Despite extensive research during the last half of the twentieth century, however, some controversy remains over the ultimate value of eye

movement research in specific applications. Still, coupled with other user testing techniques, where developers need the deepest possible user input data to ensure the highest level of software efficiency (weapons guidance systems, robotic control systems, vehicle controls or other system that require fast and accurate responses), eye tracking can be a useful tool.

Usability Questionnaire

As with the software testing procedure itself, a usability questionnaire should be carefully planned to return the information required from the associated test procedure. Although you may want to include some questions that elicit free-form comments from the user, in general you want to develop questionnaires that generate responses that can be counted and analyzed across the spectrum of testers. These fall into three general types:

- Yes/no answers
- True/false answers
- Agree/disagree on a scale

For example, instead of asking "What is your opinion of the main menu system," you might ask a series of questions that require an answer from 1 to 5, where 5 is totally agree and 1 is totally disagree:

1. The main menu was easy to navigate.
2. It was easy to find the proper software operation from the main menu.
3. The screen design led me quickly to the correct software operational choices.
4. Once I had operated the system, it was easy to remember how to repeat my actions.
5. The menu operations did *not* provide enough feedback to verify my choices.
6. The main menu was more difficult to navigate than other similar programs I use.
7. I had difficulty repeating previously accomplished operations.

Notice that it may be good practice to ask the same question more than once, but present it from the opposite perspective so that one elicits

a negative response and the other a positive one. Such practice can ensure that the user understood the question and that perceptions remain constant. In addition, you want to separate the user questionnaire into sections that correspond to software areas tested or to the testing tasks assigned.

Experience will teach you quickly which types of questions are conducive to data analysis and which ones aren't very useful. Statistical analysis software is available to help capture and interpret data. With a small number of testing users, the usability test results may be obvious; or you might develop an ad hoc analysis routine within a spreadsheet application to better document results. For large software systems that undergo extensive testing with a large user base, statistical software may help uncover trends that aren't obvious with manual interpretation methods.

When Is Enough, Enough?

How do you plan usability testing so that all aspects of the software are reasonably tested while staying within an acceptable budget? The answer to that question, of course, depends in part on the complexity of the system or unit being tested. If budget and time allow, it is advisable to test software in stages, as each segment is completed. If individual components have been tested throughout the development process, then the final series of tests need only test the integrated operation of the parts.

Additionally, you may design *component tests*, which are intended to test the usability of an interactive component, something that requires user input and that responds to this input in a user-perceivable way. This kind of feedback testing can help improve the user experience, reduce operational errors, and improve software consistency. Again, if you have tested a software system at this level as the user interface was being designed, you will have collected a significant body of important testing and operational knowledge before total system testing begins.

How many individual users should test your software? Again, system complexity and initial test results should dictate the number of individual testers. For example, if three or five (or some reasonable number) of users have difficulty navigating from the opening screen to screens that support the assigned tasks, and if these users are sufficiently representative of the target market, then you likely have enough information to tell you that the user interface needs more design work.

A reasonable corollary to this might be that if none of the initial testers have a problem navigating through their assigned tasks, and none uncover any mistakes or malfunctions, then perhaps the testing pool is too small. After all, is it reasonable to assume that usability tests of a reasonably complex software system will uncover no errors or required changes? Recall principle 6, from Table 2.1: *Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.* There's a subtle difference in this comparison. You might find that a series of users determine that a program does, in fact, seem to do what it is supposed to do. They find no errors or problems in working through the software. But have they also proven that the program isn't doing anything it is *not* supposed to do? If things appear to be running too smoothly during initial testing, it probably is time for more tests.

We don't believe there is a formula that tells you how many tests each user should conduct, or how many iterations of each test should be required. We do believe, however, that careful analysis and understanding of the results you gather from some reasonable number of testers and tests can guide you to the answer of when enough testing is enough.

Summary

Modern software, coupled with the pressure of intense competition and tight deadlines, make user testing of any software product crucial to successful development. It stands to reason that the targeted software user can be a valuable asset during testing. The knowledgeable user can determine whether the product meets the goal of its design, and by conducting real-world tasks can find errors of commission and omission.

Depending on the software target market, developers also may benefit from selecting random users—persons who are not familiar with the program's specification, or perhaps even the industry or market for which it is intended—who can uncover errors or user interface problems. For the same reason that the developers don't make good error testers, expert users may avoid operational areas that might produce problems because they know how the software is supposed to work. Over many years of software development we have discovered one unavoidable testing truth: The software the developer has tested for many hours can be broken easily, and in a

short time, by an unsophisticated user who attempts a task for which the user interface or the software was not designed.

Remember, too, that a key to successful user (or usability) testing is accurate and detailed data gathering and analysis. The data-gathering process actually begins with the development of detailed user instructions and a task list. It ends by compiling results from user observation or post-test questionnaires.

Finally, the testing results must be interpreted, and then developers must effect software changes identified from the data. This may be an iterative process wherein the same testing users are asked to complete similar tasks after identified software changes have been completed.

8 Debugging

In brief, debugging is what you do after you have executed a successful test case. Remember that a successful test case is one that shows that a program *does not* do what it was designed to do. Debugging is a two-step process that begins when you find an error as a result of a successful test case. Step 1 is the determination of the exact nature and location of the suspected error within the program. Step 2 consists of fixing the error.

As necessary and integral as debugging is to program testing, it seems to be the one aspect of the software production process that programmers enjoy the least, for these reasons primarily:

- *Your ego may get in the way*. Like it or not, debugging confirms that programmers are not perfect; they commit errors in either the design or the coding of the program.
- *You may run out of steam*. Of all the software development activities, debugging is the most mentally taxing activity. Moreover, debugging usually is performed under a tremendous amount of organizational or self-induced pressure to fix the problem as quickly as possible.
- *You may lose your way*. Debugging is mentally taxing because the error you've found could occur in virtually any statement within the program. Without examining the program first, you can't be absolutely sure, for example, that the origin of a numerical error in a paycheck produced by a payroll program is not a subroutine that asks the operator to load a particular form into the printer. Contrast this with

the debugging of a physical system, such as an automobile. If a car stalls when moving up an incline (the symptom), you can immediately and validly eliminate as the cause of the problem certain parts of the system—the AM/FM radio, for example, or the speedometer or the trunk lock. The problem must be in the engine; and, based on our overall knowledge of automotive engines, we can even rule out certain engine components such as the water pump and the oil filter.

- *You may be on your own.* Compared to other software development activities, comparatively little research, literature, and formal instruction exist on the process of debugging.

Although this is a book about software testing, not debugging, the two processes are obviously related. Of the two aspects of debugging, locating the error and correcting it, locating the error represents perhaps 95 percent of the problem. Hence, this chapter concentrates on the process of finding the location of an error, given that a successful test case has found one.

Debugging by Brute Force

The most common scheme for debugging a program is the so-called brute-force method. It is popular because it requires little thought and is the least mentally taxing of the methods; unfortunately, it is inefficient and generally unsuccessful.

Brute-force methods can be partitioned into at least three categories:

- Debugging with a storage dump.
- Debugging according to the common suggestion to "scatter print statements throughout your program."
- Debugging with automated debugging tools.

The first, debugging with a storage dump (usually a crude display of all storage locations in hexadecimal or octal format) is the most inefficient of the brute-force methods. Here's why:

- It is difficult to establish a correspondence between memory locations and the variables in a source program.
- With any program of reasonable complexity, such a memory dump will produce a massive amount of data, most of which is irrelevant.

- A memory dump is a static picture of the program, showing the state of the program at only one instant in time; to find errors, you have to study the dynamics of a program (state changes over time).
- A memory dump is rarely produced at the exact point of the error, so it doesn't show the program's state at the point of the error. Program actions between the time of the dump and the time of the error can mask the clues you need to find the error.
- Adequate methodologies don't exist for finding errors by analyzing a memory dump (so many programmers stare, with glazed eyes, wistfully expecting the error to expose itself magically from the program dump).

Scattering statements throughout a failing program to display variable values isn't much better. It may be better than a memory dump because it shows the dynamics of a program and lets you examine information that is easier to relate to the source program, but this method, too, has many shortcomings:

- Rather than encouraging you to think about the problem, it is largely a hit-or-miss method.
- It produces a massive amount of data to be analyzed.
- It requires you to change the program; such changes can mask the error, alter critical timing relationships, or introduce new errors.
- It may work on small programs, but the cost of using it in large programs is quite high. Furthermore, it often is not even feasible on certain types of programs such as operating systems or process control programs.

Automated debugging tools work similarly to inserting print statements within the program, but rather than making changes to the program, you analyze the dynamics of the program with the debugging features of the programming language or special interactive debugging tools. Typical language features that might be used are facilities that produce printed traces of statement executions, subroutine calls, and/or alterations of specified variables. A common capability and function of debugging tools is to set breakpoints that cause the program to be suspended when a particular statement is executed or when a particular variable is altered, enabling the programmer to examine the current state of the program. This method,

too, is largely hit or miss, however, and often results in an excessive amount of irrelevant data.

The general problem with these brute-force methods is that they ignore the process of *thinking*. You can draw an analogy between program debugging and solving a homicide. In virtually all murder mystery novels, the crime is solved by careful analysis of the clues and by piecing together seemingly insignificant details. This is not a brute-force method; setting up roadblocks or conducting property searches would be.

There also is some evidence to indicate that whether the debugging teams are composed of experienced programmers or students, people who use their brains rather than a set of aids work faster and more accurately in finding program errors. Therefore, we could recommend brute-force methods only: (1) when all other methods fail, or (2) as a supplement to, not a substitute for, the thought processes we'll describe next.

Debugging by Induction

It should be obvious that careful thought will find most errors without the debugger even going near the computer. One particular thought process is *induction*, where you move from the particulars of a situation to the whole. That is, start with the clues (the symptoms of the error and possibly the results of one or more test cases) and look for relationships among the clues. The induction process is illustrated in Figure 8.1.

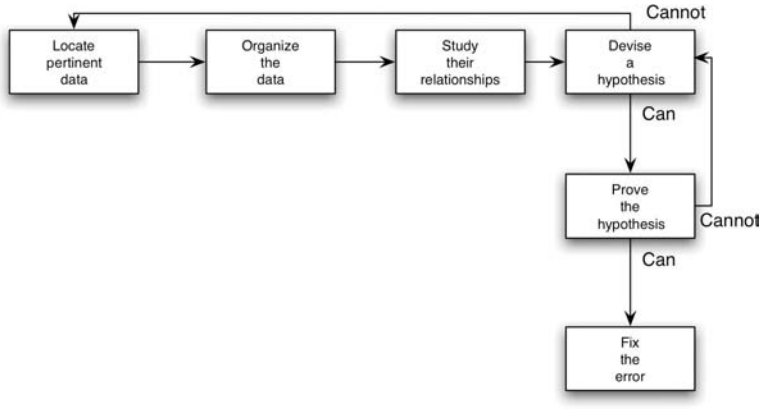

FIGURE 8.1 The Inductive Debugging Process.

The steps are as follows:

1. *Locate the pertinent data.* A major mistake debuggers make is failing to take account of all available data or symptoms about the problem. Therefore, the first step is the enumeration of all you know about what the program did correctly and what it did incorrectly—the symptoms that led you to believe there was an error. Additional valuable clues are provided by similar, but different, test cases that *do not* cause the symptoms to appear.
2. *Organize the data.* Remember that induction implies that you're processing from the particulars to the general, so the second step is to structure the pertinent data to let you observe the patterns. Of particular importance is the search for contradictions, events such as the error occurs only when the customer has no outstanding balance in his or her margin account.

   You can use a form such as the one shown in Figure 8.2 to structure the available data. In the "what" boxes list the general symptoms; in the "where" boxes describe where the symptoms were observed; in the "when" boxes list anything you know about the times when the symptoms occurred; and in the "to what extent" boxes describe the scope and magnitude of the symptoms. Notice the "is" and "is not"

?	Is	Is not
What		
Where		
When		
To what extent		

FIGURE 8.2 A Method for Structuring the Clues.

columns: In them describe the contradictions that may eventually lead to a hypothesis about the error.

3. *Devise a hypothesis*. Next, study the relationships among the clues, and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. If you can't devise a theory, more data are needed, perhaps from new test cases. If multiple theories seem possible, select the more probable one first.
4. *Prove the hypothesis*. A major mistake at this point, given the pressures under which debugging usually is performed, is to skip this step and jump to conclusions to fix the problem. Resist this urge, for it is vital to prove the reasonableness of the hypothesis before you proceed. If you skip this step, you'll probably succeed in correcting only the problem symptom, not the problem itself. Prove the hypothesis by comparing it to the original clues or data, making sure that this hypothesis *completely* explains the existence of the clues. If it does not, the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present.
5. *Fix the problem*. You can proceed with fixing the problem once you complete the previous steps. By taking the time to fully work through each step, you can feel confident that your fix will correct the bug. Remember though, that you still need to perform some type of regression testing to ensure your bug fix didn't create problems in other program areas. As the application grows larger, so does the likelihood that your fix will cause problems elsewhere.

As a simple example, assume that an apparent error has been reported in the examination grading program described in Chapter 4. The apparent error is that the median grade seems incorrect in some, but not all, instances. In a particular test case, 51 students were graded. The mean score was correctly printed as 73.2, but the median printed was 26 instead of the expected value of 82. By examining the results of this test case and a few other test cases, the clues are organized as shown in Figure 8.3.

The next step is to derive a hypothesis about the error by looking for patterns and contradictions. One contradiction we see is that the error seems to occur only in test cases that use an *odd* number of students. This might be a coincidence, but it seems significant, since you compute a median differently for sets of odd and even numbers. There's another strange pattern: In some test cases, the calculated median always is less

?	Is	Is not
What	The median printed in report 3 is incorrect.	The calculation of the mean or standard deviation.
Where	Only on report 3.	On the other reports. The students' grades seem to be calculated correctly.
When	Occurred in a test run using 51 students.	Did not occur in the test runs for 2 and 200 students.
To what extent	The median printed was 26. It also occurred in the test run using one student; the median printed in this case was 1!	

FIGURE 8.3 An Example of Clue Structuring.

than or equal to the number of students ($26 \leq 51$ and $1 \leq 1$). One possible avenue at this point is to run the 51-student test case again, giving the students different grades from before to see how this affects the median calculation. If we do so, the median is still 26, so the "to what extent→ is not" box could be filled in with, "The median seems to be independent of the actual grades." Although this result provides a valuable clue, we might have been able to surmise the error without it. From available data, the calculated median appears to equal half of the number of students, rounded up to the next integer. In other words, if you think of the grades as being stored in a sorted table, the program is printing the entry number of the middle student rather than his or her grade. Hence, we have a firm hypothesis about the precise nature of the error. Next, we prove the hypothesis by examining the code or by running a few extra test cases.

Debugging by Deduction

The process of *deduction* proceeds from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion (the location of the error), as shown in Figure 8.4.

As opposed to the process of induction in a murder case, for example, where you induce a suspect from the clues, using deduction, you start with a set of suspects and, by the process of elimination (the gardener has

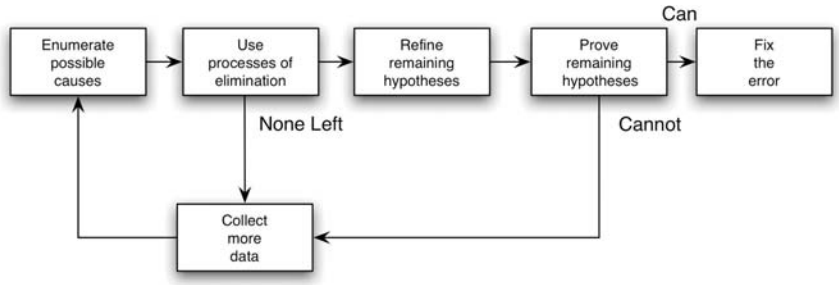

FIGURE 8.4 The Deductive Debugging Process.

a valid alibi) and refinement (it must be someone with red hair), decide that the butler must have done it. The steps are as follows:

1. *Enumerate the possible causes or hypotheses.* The first step is to develop a list of all conceivable causes of the error. They don't have to be complete explanations; they are merely theories to help you structure and analyze the available data.
2. *Use the data to eliminate possible causes.* Carefully examine all of the data, particularly by looking for contradictions (you could use Figure 8.2 here), and try to eliminate all but one of the possible causes. If all are eliminated, you need more data gained from additional test cases to devise new theories. If more than one possible cause remains, select the most probable cause—the prime hypothesis—first.
3. *Refine the remaining hypothesis.* The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory. For example, you might start with the idea that "there is an error in handling the last transaction in the file" and refine it to "the last transaction in the buffer is overlaid with the end-of-file indicator."
4. *Prove the remaining hypothesis.* This vital step is identical to step 4 in the induction method.
5. *Fix the error.* Again this step is identical to step 5 in the induction method. To re-emphasize though, you should thoroughly test your fix to ensure it does not create problems elsewhere in the application.

As an example, assume that we are commencing the function testing of the `DISPLAY` command discussed in Chapter 4. Of the 38 test cases

Test case input	Expected output	Actual Output
DISPLAY.E	000000 = 0000 4444 8888 CCCC	M1 INVALID COMMAND SYNTAX
DISPLAY 21 v- 29	0000020 = 0000 4444 8888 CCCC	000020 = 4444 8888 CCCC 0000
DISPLAY .11	000000 = 0000 4444 8888 CCCC 000010 = 0000 4444 8888 CCCC	000000 = 0000 4444 8888 CCCC
DISPLAY 8000 - END	M2 STORAGE REQUESTED IS BEYOND ACTUAL MEMORY LIMITS	008000 = 0000 4444 8888 CCCC

FIGURE 8.5 Test Case Results from the DISPLAY Command.

identified by the process of cause-effect graphing, we start by running four test cases. As part of the process of establishing input conditions, we will initialize memory that the first, fifth, ninth, . . . , words have the value 000; the second, sixth, . . . , words have the value 4444; the third, seventh, . . . , words have the value 8888; and the fourth, eighth, . . . , words have the value CCCC. That is, each memory word is initialized to the low-order hexadecimal digit in the address of the first byte of the word (the values of locations 23FC, 23FD, 23FE, and 23FF are C).

The test cases, their expected output, and the actual output after the test are shown in Figure 8.5.

Obviously, we have some problems, since apparently none of the test cases produced the expected results (all were successful). But let's start by debugging the error associated with the first test case. The command indicates that, starting at location 0 (the default), E locations (14 in decimal) are to be displayed. (Recall that the specification stated that all output will contain four words, or 16 bytes per line.)

Enumerating the possible causes for the unexpected error message, we might get:

1. The program does not accept the word *DISPLAY*.
2. The program does not accept the period.
3. The program does not allow a default as a first operand; it expects a storage address to precede the period.
4. The program does not allow an E as a valid byte count.

The next step is to try to eliminate the causes. If all are eliminated, we must retreat and expand the list. If more than one remain, we might want to examine additional test cases to arrive at a single error hypothesis, or proceed with the most probable cause. Since we have other test cases at hand, we see that the second test case in Figure 8.5 seems to eliminate the first hypothesis; and the third test case, although it produced an incorrect result, seems to eliminate the second and third hypotheses.

The next step is to refine the fourth hypothesis. It seems specific enough, but intuition might tell us that there is more to it than meets the eye—it sounds like an instance of a more general error. We might contend, then, that the program does not recognize the special hexadecimal characters A–F. This absence of such characters in the other test cases makes this sound like a viable explanation.

Rather than jumping to a conclusion, however, we should first consider *all* of the available information. The fourth test case might represent a totally different error, or it might provide a clue about the current error. Given that the highest valid address in our system is 7FFF, how could the fourth test case display an area that appears to be nonexistent? The fact that the displayed values are our initialized values, and not garbage, might lead to the supposition that this command is somehow displaying something in the range 0–7FFF. One idea that may arise is that this could occur if the program were treating the operands in the command as *decimal* values rather than hexadecimal, as stated in the specification. This is borne out by the third test case: Rather than displaying 32 bytes of memory, the next increment above 11 in hexadecimal (17 in base 10), it displays 16 bytes of memory, which is consistent with our hypothesis that the 11 is being treated as a base-10 value. Hence, the refined hypothesis is that the program is treating the byte count as storage address operands, and the storage addresses on the output listing as decimal values.

The last step is to prove this hypothesis. Looking at the fourth test case, if 8000 is interpreted as a decimal number, the corresponding base-16 value is 1F40, which would lead to the output shown. As further proof, examine the second test case. The output is incorrect, but if 21 and 29 are treated as decimal numbers, the locations of storage addresses 15–1D would be displayed; this is consistent with the erroneous result of the test case. Hence, we have almost certainly located the error: The program is assuming that the operands are decimal values and is printing the memory addresses as decimal values, which is inconsistent with the specification.

Moreover, this error seems to be the cause of the erroneous results of all four test cases. A little thought has led to the error, and it also solved three other problems that, at first glance, appear to be unrelated.

Note that the error probably manifests itself at two locations in the program: the part that interprets the input command and the part that prints memory addresses on the output listing.

As an aside, this error, likely caused by a misunderstanding of the specification, reinforces the suggestion that a programmer should not attempt to test his or her own program. If the programmer who created this error is also designing the test cases, he or she likely will make the same mistake while writing the test cases. In other words, the programmer's expected outputs would not be those of Figure 8.5; they would be the outputs calculated under the assumption that the operands are decimal values. Therefore, this fundamental error probably would go unnoticed.

Debugging by Backtracking

An effective method for locating errors in small programs is to backtrack the incorrect results through the logic of the program until you find the point where the logic went astray. In other words, start at the point where the program gives the incorrect result—such as where incorrect data were printed. Here, you deduce from the observed output what the values of the program's variables must have been. By performing a mental reverse execution of the program from this point and repeatedly applying the if-then logic that states "if this was the state of the program at this point, then this must have been the state of the program up here," you can quickly pinpoint the error. You're looking for the location in the program between the point where the state of the program was what it was expected to be and the first point where the state of the program was not what it was expected to be.

Debugging by Testing

The last "thinking type" debugging method is the use of test cases. This probably sounds a bit peculiar since, at the beginning of this chapter, we distinguished debugging from testing. However, consider two types of test cases: *test cases for testing*, whose purpose is to expose a previously undetected error, and *test cases for debugging*, whose purpose is to provide

information useful in locating a suspected error. The difference between the two is that test cases for testing tend to be "fat," in that you are trying to cover many conditions in a small number of test cases. Test cases for debugging, on the other hand, are "slim," because you want to cover only a single condition or a few conditions in each test case.

In other words, after you have discovered a symptom of a suspected error, you write variants of the original test case to attempt to pinpoint the error. Actually, this is not an entirely separate method; it often is used in conjunction with the induction method to obtain information needed to generate a hypothesis and/or to prove a hypothesis. It also is used with the deduction method to eliminate suspected causes, refine the remaining hypothesis, and/or prove a hypothesis.

Debugging Principles

In this section, we want to discuss a set of debugging principles that are psychological in nature. As with the testing principles in Chapter 2, many of these debugging principles are intuitively obvious, yet they are often forgotten or overlooked.

Since debugging is a two-part process—locating an error and then repairing it—we discuss two sets of principles here.

Error-Locating Principles

Think As implied in the previous section, debugging is a problem-solving process. The most effective method of debugging involves a mental analysis of the information associated with the error's symptoms. An efficient program debugger should be able to pinpoint most errors without going near a computer. Here's how:

1. Position yourself in a quiet place, where outside stimuli—voices of coworkers, telephones, radio or other potential interruptions—won't interfere with your concentration.
2. Without looking at the program code, review in your mind how the program is designed, how the software should be performing within the area that is performing incorrectly.
3. Concentrate on the process for correct performance, and then imagine ways in which the code may be incorrectly designed.

This sort of prethinking the physical debugging process will, in many cases, lead you directly to the area of the program that is causing problems and help you achieve a fix, quickly.

If You Reach an Impasse, Sleep on It The human subconscious is a potent problem solver. What we often refer to as inspiration is simply the subconscious mind working on a problem when the conscious mind is focused on something else, such as eating, walking, or watching a movie. If you cannot locate an error in a reasonable amount of time (perhaps 30 minutes for a small program, several hours for a larger one), drop it and turn your attention to something else, since your thinking efficiency is about to collapse anyway. After putting aside the problem for a while, your subconscious mind will have solved the problem, or your conscious mind will be clear for a fresh examination of its symptoms.

We have used this technique regularly over the years, both as a development process as well as a debugging process. It may take some practice to accept this extraordinary functioning of the human brain, and make efficient use of it, but it does work. We have actually awakened in the night to realize we have solved a software problem while asleep. For this reason, we recommend that you keep by your bedside a small tape recorder, a telephone capable of voice recording, a PDA, or a notepad to capture the solution you found while sleeping. Resist the temptation to return to sleep believing you will be able to regenerate the solution in the morning. You probably won't—at least not in our experience.

If You Reach an Impasse, Describe the Problem to Someone Else Talking about the problem with someone else may help you discover something new. In fact, often, simply by describing the problem to a good listener, you will suddenly see the solution without any assistance from the person.

Use Debugging Tools Only as a Second Resort Turn to debugging tools only after you've tried other methods, and then only as an adjunct to, not a substitute for, thinking. As noted earlier in this chapter, debugging tools, such as dumps and traces, represent a haphazard approach to debugging. Experiments show that people who shun such tools, even when they are debugging programs that are unfamiliar to them, are more successful than people who use the tools.

Why should this be so? Depending on a tool to solve a problem can short-circuit the diagnostic process. If you believe that the tool can solve the problem, you are likely to be less attentive to the clues you already have picked up, information that could help you solve the problem directly, without the help of a generic diagnostic tool.

Avoid Experimentation—Use It Only as a Last Resort The most common mistake novice debuggers make is to try to solve a problem by making experimental changes to the program. You might think, "I know what is wrong, so I'll change this *DO* statement and see what happens." This totally haphazard approach cannot even be considered debugging; it represents an act of blind hope. Not only does it have a minuscule chance of success, but you will often compound the problem by adding new errors to the program.

Error-Repairing Techniques

Where There Is One Bug, There Is Likely to Be Another This is a restatement of principle 9 in Chapter 2, which states that when you find an error in a section of a program, the probability of the existence of another error in that same section is higher than if you hadn't already found one error. In other words, errors tend to cluster. When repairing an error, examine its immediate vicinity for anything else that looks suspicious.

Fix the Error, Not Just a Symptom of It Another common failing is repairing the symptoms of the error, or just one instance of the error, rather than the error itself. If the proposed correction does not match all the clues about the error, you may be fixing only a part of the error.

The Probability of the Fix Being Correct Is Not 100 Percent Tell this to someone in general conversation and of course he or she would agree; but tell it to someone in the process of correcting an error and you may get a different answer—"Yes, in most cases, but this correction is so minor that it just has to work." Never assume that code added to a program to fix an error is correct. Statement for statement, corrections are much more error prone than the original code in the program. One implication is that error corrections must be tested, perhaps more rigorously than the original

program. A solid regression testing plan can help ensure that correcting an error does not introduce another error somewhere else in the application.

The Probability of the Fix Being Correct Drops as the Size of the Program Increases Stating it differently, in our experience, the ratio of errors caused by incorrect fixes, versus original errors, increases in large programs. In one widely used large program, one of every six new errors discovered is an error in a prior correction to the program.

If you accept this as fact, how can you avoid causing problems by trying to fix them? Read the first three techniques in this section, for starters. One error found does not mean all errors have been found, and you must be sure you are correcting the actual error, not just its symptom.

Beware of the Possibility That an Error Correction Creates a New Error Not only do you have to worry about incorrect corrections, you also have to worry about a seemingly valid correction having an undesirable side effect, thus introducing a new error. Not only is there a probability that a fix will be invalid, but there also is a probability that a fix will introduce a new error. One implication is that not only do you have to test the error situation after the correction is made, but you must also perform regression testing to determine whether a new error has been introduced.

The Process of Error Repair Should Put You Temporarily Back into the Design Phase Realize that error correction is a form of program design. Given the error-prone nature of corrections, common sense says that whatever procedures, methodologies, and formalism were used in the design process should also apply to the error-correction process. For instance, if the project rationalized that code inspections were desirable, then it must be doubly important that they be implemented after correcting an error.

Change the Source Code, Not the Object Code When debugging large systems, particularly those written in an assembly language, occasionally there is the tendency to correct an error by making an immediate change to the object code, with the intention of changing the source program later. Two problems are associated with this approach: (1) It usually is a sign that "debugging by experimentation" is being practiced; and (2) the object code and source program are now out of synchronization, meaning that the error could easily resurface when the program is recompiled or

reassembled. This practice is an indication of a sloppy, unprofessional approach to debugging.

Error Analysis

The last point to realize about program debugging is that in addition to its value in removing an error from the program, it can have another valuable effect: It can tell us something about the nature of software errors, something we still know too little about. Information about the nature of software errors can provide valuable feedback in terms of improving future design, coding, and testing processes.

Every programmer and programming organization could improve immensely by performing a detailed analysis of the detected errors, or at least a subset of them. Admittedly, it is a difficult and time-consuming task, for it implies much more than a superficial grouping such as "*x* percent of the errors are logic design errors," or "*x* percent of the errors occur in `IF` statements." A careful analysis might include the following studies:

- *Where was the error made?* This question is the most difficult one to answer, because it requires a backward search through the documentation and history of the project; at the same time, it also is the most valuable question. It requires that you pinpoint the original source and time of the error. For example, the *original* source of the error might be an ambiguous statement in a specification, a correction to a prior error, or a misunderstanding of an end-user requirement.
- *Who made the error?* Wouldn't it be useful to discover that 60 percent of the design errors were created by one of the 10 analysts, or that programmer X makes three times as many mistakes as the other programmers? (Not for the purposes of punishment but for the purposes of education.)
- *What was done incorrectly?* It is not sufficient to determine when and by whom each error was made; the missing link is a determination of exactly why the error occurred. Was it caused by someone's inability to write clearly? Someone's lack of education in the programming language? A typing mistake? An invalid assumption? A failure to consider valid input?
- *How could the error have been prevented?* What can be done differently in the next project to prevent this type of error? The answer to this

question constitutes much of the valuable feedback or learning for which we are searching.

- *Why wasn't the error detected earlier?* If the error was detected during a test phase, you should study why the error was not unearthed during earlier testing phases, code inspections, and design reviews.
- *How could the error have been detected earlier?* The answer to this offers another piece of valuable feedback. How can the review and testing processes be improved to find this type of error earlier in future projects? Providing that we are not analyzing an error found by an end user (that is, the error was found by a test case), we should realize that something valuable has happened: We have written a successful test case. Why was this test case successful? Can we learn something from it that will result in additional successful test cases, either for this program or for future programs?

We repeat, this analysis process is difficult, and costly, but the answers you may discover by going through it can be invaluable in improving subsequent programming efforts. The quality of future products will increase while the capital investment will decrease. It is alarming that the vast majority of programmers and programming organizations do not employ it.

Summary

The main focus of this book is on software testing: How do you go about uncovering as many software errors as possible? Therefore, we don't want to spend too much time on the next step—debugging—but the simple fact is, errors found by successful test cases lead directly to it.

In this chapter we touched on some of the more important aspects of software debugging. The least desirable method, debugging by brute force, involves such techniques as dumping memory locations, placing print statements throughout the program, or using automated tools. Brute-force techniques may point you to the solution for some errors uncovered during testing, but they are not an efficient way to go about debugging.

We demonstrated that you can begin debugging by studying the error symptoms, or clues, and moving from them to the larger picture (inductive debugging). Another technique begins the debugging process by considering general theories, then, through the process of elimination, identifies

the error locations (deductive debugging). We also covered program backtracking—starting with the error and moving backwards through the program to determine where incorrect information originated. Finally, we discussed debugging by testing.

If, however, we were to offer a single directive to those tasked with debugging a software system, we would say, "Think!" Review the numerous debugging principles described in this chapter. We believe they can lead you in the right direction, toward accurate and efficient debugging. But the bottom line is, depend on your expertise and knowledge of the program itself. Open your mind to creative solutions, review what you know, and let your knowledge and subconscious lead you to the error locations.

In the next chapter we take on the subject of extreme testing, techniques well suited to help uncover errors in extreme programming environments such as agile development.

9 Testing in the Agile Environment

Increased competition and interconnectedness in all markets have forced businesses to shorten their time-to-market while continuing to provide high-quality products to their customers. This is particularly true in the software development industry where the Internet makes possible near-instant delivery of software applications and services. Whether creating a product for the masses or for the human resources department, one fact remains immutable: The twenty-first century customer demands a quality application delivered almost immediately. Unfortunately, traditional software development processes cannot keep up in this competitive environment.

In the early 2000s, a group of developers met to discuss the state of lightweight and rapid development methodologies. At the gathering they compared notes to identify what successful software projects look like; what made some projects succeed while others limped along. In the end, they created the "Manifesto for Agile Software Development," a document that became the cornerstone of the Agile movement. Less a discrete methodology, the Agile Manifesto (Figure 9.1) is a unique philosophy that focuses on customers and employees, in lieu of rigid approaches and hierarchies.

Features of Agile Development

Agile development promotes iterative and incremental development, with significant testing, that is customer-centric and welcomes change during the process. All attributes of traditional software development approaches

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck	Mike Beedle	Arie van Bennekum
Alistair Cockburn	Ward Cunningham	Martin Fowler
James Grenning	Jim Highsmith	Andrew Hunt
Ron Jeffries	Jon Kern	Brian Marick
Robert C. Martin	Steve Mellor	Ken Schwaber
	Jeff Sutherland	Dave Thomas

FIGURE 9.1 Manifesto of Agile Software Development.

neglect or minimize the importance of the customer. Although Agile methodologies incorporate flexibility into their processes, the main emphasis is on customer satisfaction. The customer is a key component of the process; simply put, without customer involvement, the Agile method fails. And knowing their interaction is welcomed helps customers build satisfaction and confidence in the end product and development team. If the customer is not committed, then more traditional processes may be a better development choice.

Ironically, Agile development has no single development methodology or process; many rapid development approaches may be considered Agile. These approaches do, however, share three common threads: They rely on customer involvement, mandate significant testing, and have short, iterative development cycles. It is beyond the scope of this book to cover each methodology in detail, but in Table 9.1 we identify the methodologies considered Agile and give a brief description of each. (We urge you to learn

TABLE 9.1 Agile Development Methodologies

Methodology	Description
Agile Modeling	Not so much a single modeling methodology, but a collection of principles and practices for modeling and documenting software systems. Used to support other methods such as Extreme Programming and Scrum.
Agile Unified Process	Simplified version of the Rational Unified Process (RUP) tailored for Agile development.
Dynamic Systems Development Method	Based on rapid application development approaches, this methodology relies on continuous customer involvement and uses an iterative and incremental approach, with the goal of delivering software on time and within budget.
Essential Unified Process (EssUP)	An adaptation of RUP in which you choose the practices, (e.g. use cases or team programming) that fit your project. RUP generally uses all practices, whether needed or not.
Extreme Programming	Another iterative and incremental approach that relies heavily on unit and acceptance testing. Probably the best known of the Agile methodologies.
Feature Driven Development	A methodology that uses industry best practices, such as regular builds, domain object modeling, and feature teams, that are driven by the customer's feature set.
Open Unified Process	An Agile approach to implementing standard Unified practices that allows a software team to rapidly develop their product.
Scrum	An iterative and incremental project management approach that supports many Agile methodologies.
Velocity Tracking	Applies to all Agile development methodologies. It attempts to measure the rate, or "velocity," at which the development process is moving.

more about them because they represent the essence of the Agile philosophy.) In addition, we cover Extreme Programming, one of the more popular Agile methodologies, in greater detail later in this chapter, and offer a practical example.

It's worth noting that some Agile methodologies are collections, or adaptations, of traditional software development processes. The Essential Unified Process (EssUP) is an example. EssUP takes processes from the Rational Unified Process (RUP) and other well-known software development process models that support the Agile development philosophy.

Make no mistake, adopting an Agile development methodology is challenging. It takes the right combination of developers, managers, and customers to make it work. But in the end, the product will benefit from constant testing and heavy customer involvement.

Agile Testing

In essence, Agile testing is a form of collaborative testing, in that everyone is involved in the process through design, implementation, and execution of the test plan. Customers are involved in defining acceptance tests by defining use cases and program attributes. Developers collaborate with testers to build test harnesses that can test functionality automatically. Agile testing requires that everyone be engaged in the test process, which requires a lot of communication and collaboration.

As with most aspects of Agile development, Agile testing necessitates engaging the customer as early as possible and throughout the development cycle. For example, once developers produce a stable code base, customers should begin acceptance testing and provide feedback to the development team. It also means that testing is not a phase; rather, it is integrated with development efforts to compel continuous progress.

To ensure that the customer receives a stable product with which to perform acceptance testing, developers generally begin by writing unit tests first, then move to coding software units. The unit tests are failure tests, in that developers design them to cause their software to fail some requirement. Paradoxically, developers must write failing software to, in effect, test the test. Once test harnesses are in place, developers proceed to write software that passes the unit tests.

To facilitate the timely feedback needed for rapid development, Agile testing relies on automated testing. Development cycles are short, so time is valuable, and automated testing is more reliable than manual testing approaches. Not only is manual testing time-consuming, it may itself introduce bugs. Numerous open-source and commercial testing suites exist. It really does not matter which of these available testing suites is used, only

that developers and testers use one. Although some problems may require exploratory manual testing, automated testing is preferred.

Agile development environments often comprise only small teams of developers, who also act as testers. Larger projects with more resources may include an individual tester or a testing group. In either case, testers should not be considered finger-pointers. Their job is to move the project forward by providing feedback about the quality of the software so that developers can implement bug fixes and make requirement changes and general improvements.

Agile testing fits well into the Extreme Programming methodology whereby developers create unit tests first, then the software. In the remainder of this chapter we cover Extreme Programming and Extreme Testing in more detail.

Extreme Programming and Testing

In the 1990s an innovative software development methodology termed Extreme Programming (XP) was born. A project manager named Kent Beck is credited with conceiving this lightweight, Agile development process, first testing it while working on a project at Daimler-Chrysler in 1996. Although several other Agile software development processes have since been created, XP is still the most popular. In fact, numerous open-source tools exist to support it, which is testimony to XP's popularity among developers and project managers.

XP likely was developed to support the adoption of programming languages such as Java, Visual Basic, and C#.

These object-based languages allow developers to create large, complex applications much more quickly than with traditional languages such as C, Fortran, or COBOL. Developing with these languages often requires building general-purpose libraries to support the application's coding efforts. Methods for common tasks such as printing, sorting, networking, and statistical analysis are not standard components. Languages such as C# and Java ship with full-featured application programming interfaces (APIs) that eliminate or reduce the need for creating custom libraries.

However, along with the benefits of rapid application development languages came liabilities. Although developers were creating applications much more quickly, their quality was not guaranteed. If an application compiled, it often failed to meet the customer's specifications or

expectations. The XP development methodology facilitates the creation of quality programs in short time frames. Although classical software processes still work, they often take too much time, which equates to lost income in the highly competitive arena of software development.

Besides customer involvement, the XP model relies heavily on unit and acceptance testing. In general, developers run unit tests for every incremental code change, no matter how small, to ensure that the code base still meets its specification. In fact, testing is of such importance in XP that the process requires you to create the unit (module) and acceptance tests first, then your code base. This form of testing is called, appropriately, Extreme Testing (XT).

Extreme Programming Basics

As mentioned, XP is a software process that helps developers create high-quality code, rapidly. Here, we define "quality" as a code base that meets the design specification and customer expectation.

XP focuses on:

- Implementing simple designs.
- Communicating between developers and customers.
- Continually testing the code base.
- Refactoring, to accommodate specification changes.
- Seeking customer feedback.

XP tends to work well for small to medium-size development efforts in environments that have frequent specification changes, and where near-instant communication is possible.

XP differs from traditional development processes in several ways. First, it avoids the large-scale project syndrome in which the customer and the programming team meet to design every detail of the application before coding begins. Project managers know this approach has its drawbacks, not the least of which is that customer specifications and requirements constantly change to reflect new business rules or marketplace conditions. For example, the finance department may want payroll reports sorted by processed date instead of check numbers; or the marketing department may determine that consumers will not buy product XYZ if it doesn't send an e-mail after website registration. In contrast, XP planning sessions

focus on collecting *general application requirements*, not narrowing in on every detail.

Another difference with the XP methodology is that it avoids coding unneeded functionality. If your customer thinks the feature is needed but not required, it generally is left out of the release. Thus, you can focus on the task at hand, adding value to a software product. Concentrating only on the required functionality helps you produce quality software in short time frames.

But the primary difference of XP compared to traditional methodologies is its approach to testing. After an all-inclusive design phase, traditional software development models suggest you code first and create testing interfaces later. In XP, you *must* create the unit tests first, and then write the code to pass the tests. You design unit tests in an XP environment by following the concepts discussed in Chapter 5.

The XP development model has 12 core practices that drive the process, summarized in Table 9.2. In a nutshell, you can group the 12 core XP practices into four concepts:

1. Listening to the customer and other programmers.
2. Collaborating with the customer to develop the application's specification and test cases.
3. Coding with a programming partner.
4. Testing, and retesting, the code base.

Most of the comments for each practice listed in Table 9.2 are self-explanatory. However, a couple of the more important principles, namely planning and testing, warrant further discussion.

XP Planning A successful planning phase lays the foundation of the XP process. The planning phase in XP differs from that in traditional development models, which often combine requirements gathering and application design. Planning in XP focuses on identifying your customer's application requirements and designing user stories (or case stories) that meet them. You gain significant insight into the application's purpose and requirements by creating user stories. In addition, the customer employs the user stories when performing acceptance tests at the end of a release cycle. Finally, an intangible benefit of the planning phase is that the customer gains ownership and confidence in the application by participating intimately in it.

TABLE 9.2 The 12 Practices of Extreme Programming

Practice	Comment
1. Planning and requirements	Marketing and business development personnel work together to identify the maximum business value of each software feature. Each major software feature is written as a user story. Programmers provide time estimates to complete each user story. The customer chooses the software features based on time estimates and business value.
2. Small, incremental releases	Strive to add small, tangible, value-added features and release a new code base often.
3. System metaphors	Your programming team identifies an organizing metaphor to help with naming conventions and program flow.
4. Simple designs	Implement the simplest design that allows your code to pass its unit tests. Assume change will come, so don't spend a lot of time designing; just implement.
5. Continuous testing	Write unit tests before writing the code module. Each unit is not complete until it passes its unit test. Further, the program is not complete until it passes all unit tests, and acceptance tests are complete.
6. Refactoring	Clean up and streamline your code base. Unit tests help ensure that you do not destroy the functionality in the process. You must rerun all unit tests after any refactoring.
7. Pair programming	You and another programmer work together, at the same machine, to create the code base. This allows for real-time code review, which dramatically facilitates bug detection and resolution.
8. Collective ownership of the code	All code is owned by all programmers. No single programmer is dedicated to a specific code base.
9. Continuous integration	Every day, integrate all changes; after the code passes the unit tests, add it back into the code base.
10. Forty-hour workweek	No overtime is allowed. If you work with dedication for 40 hours per week, overtime will not be needed. The exception is the week before a major release.

Table 9.2 (*continued*)

Practice	Comment
11. On-site customer presence	You and your programming team have unlimited access to the customer, to enable you to resolve questions quickly and decisively, which keeps the development process from stalling.
12. Coding standards	All code should look the same. Developing a system metaphor helps meet this principle.

XP Testing Continuous testing is central to the success of a XP-based effort. Although acceptance testing falls under this principle, unit testing occupies the bulk of the effort. Unit tests are devised to make the software fail. Only by ensuring that your tests detect errors can you begin correcting the code so it passes the tests. Assuring that your unit tests catch failures is key to the testing process—and to a developer's confidence. At this point, the developer can experiment with different implementations, knowing that the unit tests will catch any mistakes.

You want to ensure that any code changes improve the application and do not introduce bugs. The continuous testing principle also supports refactoring efforts used to optimize and streamline the code base. Constant testing also leads to that intangible benefit already mentioned: confidence. The programming team gains confidence in the code base because you constantly validate it with unit tests. In addition, your customers' confidence in their investment soars because they know the code base passes unit tests every day.

Example XP Project Flow Now that we've presented the 12 practices of the XP process, you may be wondering, how does a typical XP project flow? Here is a quick example of what you might experience if you worked on an XP-based project:

1. Programmers meet *with* the customer to determine the product requirements and build user stories.
2. Programmers meet *without* the customer to divide the requirements into independent tasks and estimate the time to complete each task.

3. Programmers present the customer with the task list and with time estimates, and ask them to generate a priority list of features.
4. The programming team assigns tasks to pairs of programmers, based on their skill sets.
5. Each pair creates unit tests for their programming task using the application's specification.
6. Each pair works on their task with the goal of creating a code base that passes the unit tests.
7. Each pair fixes, then retests their code until all unit tests have passed.
8. All pairs gather every day to integrate their code bases.
9. The team releases a preproduction version of the application.
10. Customers run acceptance tests and either approve the application or produce a report identifying the bugs/deficiencies.
11. Upon successful acceptance tests, programmers release a version into production.
12. Programmers update time estimates based on latest experience.

Although compelling, XP is not for every project or every organization. Proponents of XP conclude that if a programming team fully implements the 12 practices, then the chances of successful application development increase dramatically. Detractors say that because XP is a process, you must do all or nothing; if you skip a practice, then you are not properly implementing XP, and your program quality may suffer. Detractors also claim that the cost of changing a program in the future to add more features is higher than the cost of initially anticipating and coding the requirement. Finally, some programmers find working in pairs very cumbersome and invasive; therefore, they do not embrace the XP philosophy.

Whatever your views, we recommend that you consider XP as a software methodology for your project. Carefully weigh its pros and cons against the attributes of your project and make the best decision based on that assessment.

Extreme Testing: The Concepts

To meet the pace and philosophy of XP, developers use Extreme Testing, which focuses on constant testing. As mentioned earlier, two forms of testing make up the bulk of XT: unit testing and acceptance testing. The theory used when writing the tests does not vary significantly from the theory presented in Chapter 5; however, the stage in the development process in

which you create the tests does differ. XT mandates creating tests *before* coding begins, not after. Nonetheless, XT and traditional testing share the same goal: to identify errors in a program.

In the rest of this section we provide more information on unit and acceptance testing, from an Extreme Programming perspective.

Extreme Unit Testing

Unit testing, the primary testing approach used in Extreme Testing, and has two simple rules: All code modules must have unit tests before coding begins, and all code modules must pass unit tests before being released into acceptance testing. At first glance this may not seem so extreme. Closer inspection reveals the big difference between unit testing, as previously described, and XT unit testing: The unit tests must be defined and created before coding the module.

Initially, you may wonder why you should, or how you can, create test drivers for code you haven't yet written. You may also think that you do not have time to create the tests and still meet the project deadline. These are valid concerns, but concerns we can address easily by listing a number of important benefits associated with writing unit tests before you start coding the application:

- You gain confidence that your code will meet its specification and requirements.
- You express the end result before you start coding.
- You better understand the application's specification and requirements.
- You may implement simple designs initially and confidently refactor the code later to improve performance, without worrying about breaking the specification.

Of these benefits, the insight and understanding you gain of the application's specification and requirements cannot be underestimated. For example, if you start coding first, you may not fully understand the acceptable data types and boundaries for the input values of an application. How can you write a unit test to perform boundary analysis without understanding the acceptable inputs? Can the application accept only numbers, only characters, or both? If you create the unit tests first, you *must* understand the specification. The practice of creating unit tests first is the shining star in the XP methodology, as it forces you to understand the specification to resolve ambiguities *before* you begin coding.

As mentioned in Chapter 5, you determine the unit's scope. Given that today's popular programming languages such as Java, C#, and Visual Basic are mostly object-oriented, modules are often classes, or even individual class methods. You may sometimes define a module as a group of classes or methods that represent some functionality. Only you, as the programmer, know the architecture of the application and how best to build the unit tests for it.

Manually running unit tests, even for the smallest application, can be a daunting task. As the application grows, you may generate hundreds or thousands of unit tests. Therefore, you typically use an automated software testing suite to ease the burden of running these unit tests. With these suites you script the tests and then run all or part of them. In addition, testing suites typically allow you to generate reports and classify the bugs that frequently occur in your application. This information may help you proactively eliminate bugs in the future.

Interestingly enough, once you create and validate your unit tests, the "testing" code base becomes as valuable as the software application you are trying to create. As a result, you should keep the tests in a code repository, for protection. Likewise, you should institute adequate backups of the test code, and ensure that needed security is in place.

Extreme Acceptance Testing

Acceptance testing represents the second, and equally important, type of XT that occurs in the XP methodology. Acceptance testing determines whether the application meets other requirements, such as functionality and usability. You and the customer create the acceptance tests during the design/planning phases.

Unlike the other forms of testing discussed thus far, customers, not you or your programming partners, conduct the acceptance tests. In this manner, customers provide the unbiased verification that the application meets their needs. Customers create the acceptance tests from user stories. The ratio of user stories to acceptance tests is usually one too many; that is, more than one acceptance test may be needed for each user story.

Acceptance tests in XT may or may not be automated. For example, an unautomated test is required when the customer must validate that a user input screen meets its specification with respect to color and screen layout. An example of an automated test is when the application must calculate payroll values using data input via some data source such as a flat file to simulate production values.

Through acceptance tests, the customer validates an expected result from the application. A deviation from the expected result is considered a bug and is reported to the development team. If the customer discovers several bugs, then he or she must prioritize them before passing the list to your development group. After you correct the bugs, or after any change, the customer reruns the acceptance tests. In this manner, the acceptance tests also become a form of regression testing.

An important note is that a program may pass all unit tests but fail the acceptance tests. How is this possible? Because a unit test validates whether a program unit meets some specification, such as calculating payroll deductions, correctly, not some defined functionality or aesthetics. For a commercial application, the look and feel is a very important component. Understanding the specification, but not the functionality, generally results in this scenario.

Extreme Testing Applied

In this section we create a small Java application and employ JUnit, a Java-based open-source unit testing suite, to illustrate the concepts of Extreme Testing (see Figure 9.2). The example itself is trivial; the concepts, however, apply to most programming situations.

Our example is a command-line application that simply determines whether an input value is a prime number. For brevity, the source code,

---

JUnit is a freely available open-source tool used to automate unit tests of Java applications in Extreme Programming environments. The creators, Kent Beck and Erich Gamma, developed JUnit to support the significant unit testing that occurs in the Extreme Programming environment. JUnit is very small, but very flexible and feature rich. You can create individual tests or a suite of tests. You can automatically generate reports detailing the errors.

Before using JUnit, or any testing suite, you must fully under- stand how to use it. JUnit is powerful but only after you master its API. However, whether or not you adopt an XP methodology, JUnit is a useful tool to provide sanity checks for your own code.

Visit www.junit.org for more information and to download the test suite. In addition, there is a wealth of information on XP and XT at this website.

FIGURE 9.2 JUnit Description and Background.

`check4Prime.java`, and its test harness, `check4PrimeTest.java`, are listed in Appendix. In this section we provide snippets from the application to illustrate the main points.

The specification of this program is as follows:

> Develop a command-line application that accepts any positive integer, *n,* where 0<=*n*<=1,000, and determine whether it is a prime number. If *n* is a prime number, then the application should return a message stating it is a prime number. If *n* is not a prime number, then the application should return a message stating it is not a prime number. If *n* is not a valid input, then the application should display a help message.

Following the XP methodology and the principles listed in Chapter 5, we begin the application by designing unit tests. With this application, we can identify two discrete tasks: validating inputs and determining prime numbers. We could use black-box and white-box testing approaches, boundary value analysis, and the decision coverage criterion, respectively. However, the XT practice mandates a hands-off black-box approach, to eliminate any bias.

Test-Case Design We begin designing test cases by identifying a testing approach. In this instance, we will use boundary analysis to validate the inputs because this application can only accept positive integers within a certain range. All other input values, including character datatypes and negative numbers, should raise an error and not be used. Of course, you could certainly make the case that input validation could fall under the decision coverage criterion, as the application must decide whether the input is valid. The important concept is to identify, and commit to, a testing approach when designing your tests.

With the testing approach identified, the next step is to develop a list of test cases based on possible inputs and expected outcome. Table 9.3 shows the eight test cases we identified for this example. (Note: As stated, we are using a very simple example here to illustrate the basics of Extreme Testing. In practice, you would have a much more detailed program specification, which might include items such as user interface requirements and output verbiage. As a result, the list of test cases would increase substantially.)

TABLE 9.3 Test Case Descriptions for check4Prime.java

Case Number	Input	Expected Output	Comments
1	$n = 3$	Affirm *n* is a prime number.	Tests for a valid prime number. Tests input within boundaries.
2	$n = 1{,}000$	Affirm *n* is not a prime number.	Tests input equal to upper bounds. Tests whether *n* is an invalid prime.
3	$n = 0$	Affirm *n* is not a prime number.	Tests input equal to lower bounds.
4	$n = -1$	Print help message.	Tests input below lower bounds.
5	$n = 1{,}001$	Print help message.	Tests input greater than the upper bounds.
6	$n =$ "a"	Print help message.	Tests input is an integer and not a character datatype.
7	Two or more inputs	Print help message.	Tests for correct number of input values.
8	*n* is empty (blank)	Print help message.	Tests whether an input value is supplied.

Test case 1 from Table 9.3 combines two test scenarios. It checks whether the input is a valid prime and how the application behaves with a valid input value. You may use any valid prime in this test.

We also test two scenarios with test case 2: What happens when the input value is equal to the upper bounds and when the input is not a prime number? This case could have been broken out into two unit tests, but one goal of software testing in general is to minimize the number of test cases while still adequately checking for error conditions.

Test case 3 checks the lower boundary of valid inputs, as well as testing for invalid primes. The second part of the check is not needed because test case 2 handles this scenario. However, it is included by default because 0 is not a prime number. Test cases 4 and 5 ensure that the inputs are within the defined range, which is greater than or equal to 0 and less than or equal to 1,000.

Case 6 tests whether the application properly handles character input values. Because we are doing a calculation, it is obvious that the

TABLE 9.4 Test Driver Methods

Methods	Test Case(s) Examined
`testCheckPrime_true()`	1
`testCheckPrime_false()`	2, 3
`testCheck4Prime_checkArgs_char_input()`	6
`testCheck4Prime_checkArgs_above_upper_bound()`	5
`testCheck4Prime_checkArgs_neg_input()`	4
`testCheck4Prime_checkArgs_2_inputs()`	7
`testCheck4Prime_checkArgs_0_inputs()`	8

application should reject character datatypes. The assumption with this test case is that Java will handle the datatype check. This application must handle the exception raised when an invalid datatype is supplied. This test will ensure that the exception is thrown. Last, tests 7 and 8 check for the correct number of input values; any number of inputs other than 1 should fail.

Test Driver and Application Now that we have designed both test cases, we can create the test driver class, `check4PrimeTest`. Table 9.4 maps the JUnit methods in `check4PrimeTest` to the test cases covered.

Note that the `testCheckPrime_false()` method tests two conditions, because the boundary values are not prime numbers. Therefore, we can check for boundary value errors and for invalid primes with one test method. Examining this method in detail reveals that the two tests actually do occur within it. Here is the complete JUnit method from the `check4JavaTest` class listed in the Appendix.

```
public void testCheckPrime_false(){
  assertFalse(check4prime.primeCheck(0));
  assertFalse(check4prime.primeCheck(10000));
}
```

Notice that the JUnit method, `assertFalse()`, checks to see whether the parameter supplied causes the method to return a *false* Boolean value. If *false* is returned, the test is considered a success.

The snippet also demonstrates one of the benefits of creating test cases and test harnesses first. You may notice that the parameter in the

`assertFalse()` method is another method, `check4prime.primeCheck(n)`. This method will reside in a class of the application. Creating the test harness first forced us to think about the structure of the application. In some respects, the application is designed to support the test harness. Here we need a method to check whether the input is a prime number, so we included it in the application.

With the test harness complete, application coding can begin. Based on the program specification, test cases, and the test harness, the resultant Java application will consist of a single class, `check4Prime`, with the following definition:

```
public class check4Prime {
  public static void main (String [] args)
  public void checkArgs(String [] args) throws
    Exception
  public boolean primeCheck (int num)
}
```

Briefly, per Java requirements, the `main()` procedure provides the entry point into the application. The `checkArgs()` method asserts that the input value is a positive integer, *n,* where 0<=*n*<=1,000. The `primeCheck()` procedure checks the input value against a calculated list of prime numbers. We implemented the sieve of Eratosthenes to quickly calculate the prime numbers. This approach is acceptable because of the small number of prime numbers involved.

Summary

With the heightened competitiveness of software development today, there is a growing need to introduce products very quickly into the marketplace. The Agile development process, when strictly adopted, provides a way for developers to create quality software for their customers at a faster rate than using traditional software development models. The end result is a satisfied customer, whether an internal or commercial consumer.

The Extreme Programming model is one of more popular Agile methodologies. This lightweight development process focuses on communication, planning, and testing. The testing aspect of Extreme Programming, termed Extreme Testing, focuses on unit and acceptance tests. You run unit tests during development and whenever a change to the code base occurs. The customer runs the acceptance tests at major release points.

Extreme Testing also requires you to create the test harness, based on the program specification, before you start coding your application. In this manner, you design your application to pass the unit tests, thus increasing the probability that it will meet the specification.

10 Testing Internet Applications

Just a few years ago, Internet-based applications seemed to be the wave of the future; today, the wave has arrived onshore, and customers, employees, and business partners expect companies to have a Web presence. This expectation is not limited only to business. Most churches, civic groups, schools, and governments all have Internet presences to serve their patrons.

Generally, small to medium-size businesses have simple Web pages they use to tout their products and services. Larger enterprises often build full-fledged e-commerce applications to sell their wares, from cookies to cars and from consulting services to entire virtual companies that exist only on the Internet.

Internet applications are essentially client-server applications in which the client is a Web browser, and the server is a Web or application server. Although conceptually simple, the complexity of these applications varies wildly. Some companies have applications built for business-to-consumer uses such as banking services and retail stores, while others have business-to-business applications such as supply chain or sales force management. Development and user presentation/user interface strategies vary for these different types of websites, and, as you might imagine, the testing approach varies as well.

The goal of testing Internet-based applications is no different from that of traditional applications. You need to uncover errors in the application before deploying it to the Internet and the end user. And, given the

complexity of these applications and the interdependency of the components, you likely will succeed in finding plenty of errors.

The importance of rooting out the errors in an Internet application cannot be overstated. As a result of the openness and accessibility of the Internet, competition in the business-to-consumer and business-to-business arena is intense. Thus, the Internet has created a buyer's market for goods and services. Consumers have developed high expectations, and if your site does not load quickly, respond immediately, and provide intuitive navigation features, chances are that the user will find another company with which to conduct business. This issue is not confined to strictly e-commerce or product promotion sites. Websites that are developed as research or information resources frequently are maintained by advertising or user donations. Either way, ample competition exists to lure users away, thereby reducing activity and concomitant revenue.

It would seem that consumers have higher-quality expectations for Internet applications than they do for those that come shrink-wrapped. When people buy boxed software from a store or an online retailer, as long as the quality is "average," they will continue to use them. One reason for this behavior is that they have paid for the application, so it must be a product they perceived as useful or desirable. And even a less-than-satisfactory program can't be corrected easily, so if it at least satisfies the users' basic needs, they likely will retain the program. In contrast, a poor, or even average, quality application on the Internet, will likely cause your customer to switch to a competitor's site. Not only will the customer leave your site if it exhibits poor quality, your corporate image will become tarnished as well. After all, who feels comfortable buying a car from a company that cannot build a suitable website? Like it or not, websites have become the new first impression for business. In general, consumers don't pay to access most websites, so there is little incentive to remain loyal in the face of mediocre design or performance.

This chapter covers some of the basics of testing Internet applications. This subject is large and complex, and many references exist that explore its details. However, you will find that the techniques explained in the early chapters apply to Internet testing as well. Nevertheless, because there are, indeed, functional and design differences between Web and conventional applications, we want to point out some of the particulars of Web-based application testing.

Basic E-Commerce Architecture

Before diving into testing Internet-based applications, we will provide an overview of the three-tier client-server (C/S) architecture used in a typical Internet-based e-commerce application. Conceptually, each tier is treated as a black box with well-defined interfaces. This model allows you to change the internals of each tier without worrying about breaking another tier. Figure 10.1 illustrates each tier and the associated components used by most e-commerce sites.

Although not an official tier in the architecture, the client side and its relevance are worth explaining. Most of the access to your applications occurs from a Web browser running on a computer, although many devices, such as cell phones, PDAs, game consoles, music players, pagers, and even refrigerators and automobiles, increasingly are being developed with Internet connectivity in mind. Browsers vary dramatically in how they render content from a website. As we discuss later in this chapter, testing for browser compatibility is one challenge associated with testing Internet applications. Vendors loosely follow published standards to help make browsers behave consistently, but they also build in proprietary enhancements that cause inconsistent behavior. The remainder of the clients employ custom applications that use the Internet as a pipeline to a particular site. In this scenario, the application mimics a standard client-server application you might find on a company's local area network.

The Web server represents the first tier in the three-tier architecture and houses the website. The look and feel of an Internet application

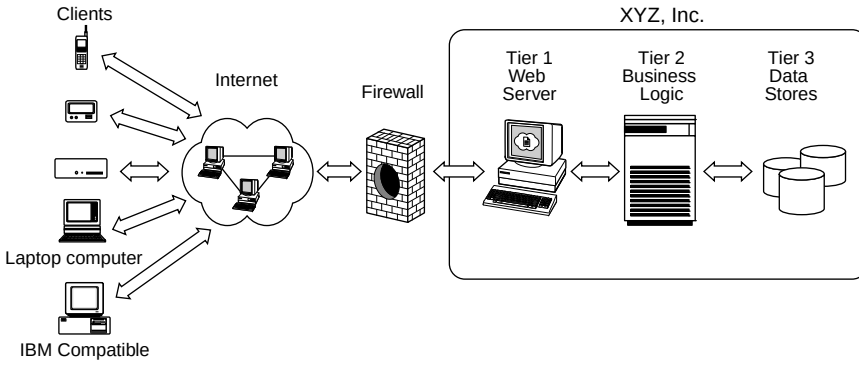

FIGURE 10.1 Typical Architecture of an E-Commerce Site.

comes from the first tier. Thus, another term for this tier is the *presentation tier* or *layer,* so dubbed because it provides the visual content to the end user. The Web server can use static HyperText Markup Language (HTML) pages or Common Gateway Interface (CGI) scripts to create dynamic HTML, but most likely it uses a combination of static and dynamic pages.

Tier 2, or the *business layer,* houses the application server. Here, you run the software that models your business processes. The following lists some of the functionality associated with the Business layer:

- Transaction processing
- User authentication
- Data validation
- Application logging

The third tier focuses on storing and retrieving data from a data source, typically a relational database management system (RDBMS). Another term for tier 3 is the *data layer*. This tier consists of a database infrastructure to communicate with the second tier. The interface into the data layer is defined by the data model, which describes how you want to store data. Sometimes, several database servers make up this tier. You typically tune database systems into this layer to handle the high transaction rates encountered in an e-commerce site. In addition to a database server, some e-commerce sites may place an authentication server in this layer. Most often, you use a Lightweight Directory Application Protocol (LDAP) server for this function.

Testing Challenges

You will face many challenges when designing and testing Internet-based applications due to the large number of elements you cannot control and the number of interdependent components. Adequately testing your application requires that you make some assumptions about your customers and how they use the site.

An Internet-based application has many failure points that you should consider when designing a testing approach. The following list provides some examples of the challenges associated with testing Internet-based applications:

- *Large and varied user base.* The users of your website possess different skill sets, employ a variety of browsers, and use different operating systems or devices. You can also expect your customers to access your website using a wide range of connection speeds. Ten years ago not everyone had broadband Internet access. Today, most do. However, you still need to consider bandwidth as Internet content becomes "richer" and more interactive.
- *Business environment.* If you operate an e-commerce site, then you must consider issues such as calculating taxes, determining shipping costs, completing financial transactions, and tracking customer profiles. These requirements may necessitate a number of external links to third-party servers or databases to manage these billing and shipping tasks, for example. The developer must thoroughly understand the structure of the remote system, and work closely with its owners and developers to ensure security and accuracy.
- *Locales.* Users may reside in other countries, in which case you will have internationalization issues such as language translation, time zone differences, and currency conversion.
- *Security.* Because your site is open to the world, you must protect it from hackers. They can bring your website to a grinding halt with denial-of-service (DoS) attacks, or rip off your customers' credit card information.
- *Testing environments.* To properly test your application, you will need to duplicate the production environment. This means you should use Web servers, application servers, and database servers that are identical to the production equipment. For the most accurate testing results, the network infrastructure will have to be duplicated as well, which includes routers, switches, and firewalls.

Even from this list, which could be expanded considerably by including viewpoints from a wide variety of developers and businesses, you can see that configuring a testing environment is one of the most challenging aspects of e-commerce development. Testing applications that process financial transactions requires the most effort and expense. You must replicate all the components, both hardware and software, used for the application to produce valid test results. Configuring such an environment is a costly endeavor. You will incur not only equipment costs, but labor costs as well. Most companies fail to factor in these expenses when creating a

budget for their applications, and those that do generally underestimate the time and monetary requirements. In addition, the testing environment needs a maintenance plan to support application upgrade efforts.

Another significant testing challenge you face is testing browser compatibility. There are several different browsers on the market today, and each behaves differently. Although standards exist for browser operation, most vendors enhance their products in an effort to attract a loyal user base. Unfortunately, this causes the browsers to operate in a nonstandard way. We cover this topic in greater detail later in this chapter.

As noted, you will face many challenges when testing Internet-based applications; therefore, the best way to proceed is to narrow your testing efforts to specific areas. Table 10.1 identifies some of the most important areas to test, to help ensure that users have a positive experience on your website.

TABLE 10.1 Examples of Presentation, Business, and Data Tier Testing

Presentation Tier	Business Tier	Data Tier
Ensure fonts are the same across browsers.	Verify proper calculation of sales tax and shipping charges.	Ensure database operations meet performance goals.
Confirm that all links point to valid files or websites.	Ensure documented performance rates are met for response times and throughput rates.	Verify that data are stored correctly and accurately.
Verify that graphics are the correct resolution and size.	Verify that transactions complete properly.	Verify that you can recover using current backups.
Spell-check each page.	Confirm that failed transactions roll back correctly.	Test failover or redundancy operations.
Have a copy editor check grammar and style.	Ensure data are collected correctly.	Test for proper data encryption and security (credit card and user's personal information, in particular).
Check cursor positioning when page loads to ensure it is in the correct text box.		

Table 10.1 (*continued*)

Presentation Tier	Business Tier	Data Tier
Confirm that default button is selected when the page loads. Check for consistent and user-friendly feedback on interactive operations. Check for business- or industry-specific terms and style.		Test backend data entry and management routines for usability and accuracy.

Because the first impression is the most important impression, some of your testing will focus on usability and human-factor concerns. This area concentrates on the look and feel of your application. Items such as fonts, colors, and graphics play a major role in whether users accept or reject your application. Keep in mind, the developer has little or no control over who will access a given application, how much computer knowledge they have, whether or not they are motivated to stay with an application in the face of navigation issues, or what users might ultimately expect in terms of information or performance.

System performance greatly influences a customer's first impression. As mentioned earlier, Internet users want instant gratification. They will not wait long for pages to load or transactions to complete. Literally, a few seconds' delay can cause a customer to try another site. Poor performance may also lead customers to doubt the reliability of your site. Therefore, you should set performance goals then design tests that reveal problems that cause your site to miss the goals.

Users also demand that their transactions complete rapidly and accurately when purchasing products or services from your site. They do not, and should not, tolerate inaccurate billings or shipping errors. Probably worse than losing a customer is finding yourself liable for more than the transaction amount if your application does not process financial transactions correctly.

Your application likely will collect data to complete tasks such as purchases or e-mail registrations. Therefore, you should ensure that the data you collect are valid. For example, make sure that phone numbers, ID

numbers, currencies, e-mail addresses, and credit card numbers are the correct length and are properly formatted. In addition, verify the integrity of your data. Localization issues can easily cause data corruption via truncation due to character-set issues.

In the Internet environment, it is critical to keep the website available for customer use. This requires that you develop and implement maintenance guidelines for all the supporting applications and servers. A Web server and RDBMS require a high level of management. You must monitor logs, system resources, and backups, and respond to any anomalies immediately. As described in Chapter 6, you want to maximize the mean time between failures (MTBF) and minimize the mean time to recovery (MTTR) for these systems.

Finally, network connectivity is another area where it is important to focus your testing efforts. At some point, you can count on losing network connectivity. The source of the failure might be the Internet itself, your service provider, or your internal network. Therefore, you need to create contingency plans for your application and infrastructure so your systems respond gracefully when an outage occurs. Keeping with the theme of testing, design your tests to break your contingency plans.

Testing Strategies

Developing a testing strategy for Internet-based applications requires a solid understanding of the hardware and software components that make up the application. As is critical to successful testing of standard applications, you will need a specification document to describe the expected functionality and performance of your website. Without this document, you will not be able to design the appropriate tests.

You need to test components developed internally and those purchased from a third party. For the components developed in-house you should employ the tactics presented in earlier chapters. This includes creating unit/module tests and performing code reviews. Integrate the components into your system only after verifying that they meet the design specifications and functionality outlined in the specification document.

If you purchase components, then you need to develop a series of system tests to validate that they perform correctly, independently of your application. Do not rely on the vendor's quality-control program to detect

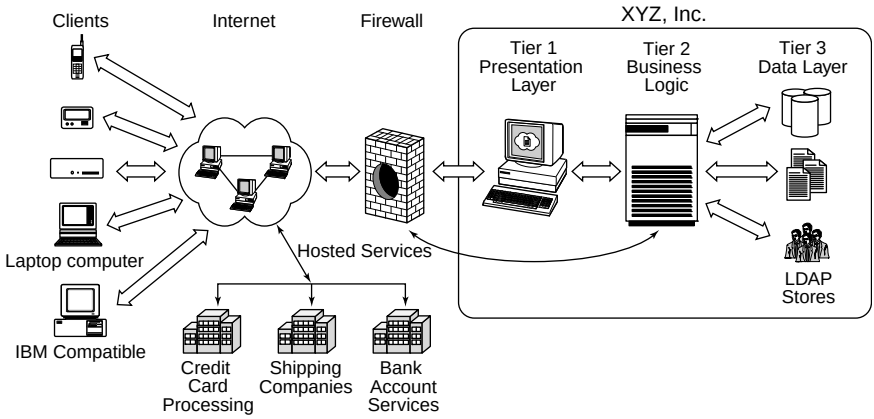

FIGURE 10.2 Detailed View of Internet Application Architecture.

errors in its components. Ideally, you should complete this task independently of your application testing. Integrate these components only once you have determined that they perform acceptably. Including a nonfunctional third-party component in your architecture makes it difficult to interpret test results and identify the source of errors. Generally, you will use black-box approaches for third-party components because you rarely will have access to the component internals.

Testing Internet-based applications is best tackled with a divide- and-conquer approach. Fortunately, the architecture of Internet applications allows you to identify discrete areas to target testing. Figure 10.1 presented the basic architecture of Internet applications. Figure 10.2 provides a more detailed view of each tier.

As mentioned earlier in this chapter, Internet applications are considered three-tier client-server applications. Each tier, or layer, from Figure 10.2 is defined as follows:

- *Presentation layer*. The layer of an Internet application that provides the user interface (UI; or GUI, graphical user interface).
- *Business layer*. The layer that models your business processes, such as user authentication and transactions.
- *Data layer*. The layer that houses data used by the application or that is collected from the end user.

Each tier has its own characteristics that encourage test segmentation. Testing each tier independently allows you to more easily identify bugs

TABLE 10.2 Items to Test in Each Tier

Test Area	Comments
Usability/human factors	Review overall look and feel. Fonts, colors, and graphics play a major role in the application aesthetics. Ensure that all user input is acknowledged so that it is clear to the user that input has been accepted.
Performance	Check for fast-loading pages. Check for quick transactions. Poor performance often creates a bad impression.
Business rules	Check for accurate representation of business process. Consider business environment for target user groups. Ensure that business or industry conventions of terminology and style are followed.
Transaction accuracy	Verify that transactions complete accurately. Confirm that cancelled transactions roll back correctly. Is input verification sufficiently strong to support security and accuracy requirements?
Data validity and integrity	Check for valid formats of phone number, e-mail addresses, and currency amounts. Ensure proper character sets.
System reliability	Test the failover capabilities of your Web, application, and database servers. Maximize MTBF and minimize MTTR.
Network architecture	Test connectivity redundancy. Test application behavior during network outages.

and errors before complete system testing begins. If you rely only on system testing, then you may have a difficult time locating the specific components that are creating the problem.

Table 10.2 lists items that you should test in each tier. The list is not comprehensive, but provides you with a starting point to develop your own testing criteria. In the remainder of this chapter we provide more details on how to test each tier.

Presentation Layer Testing

Testing the presentation layer consists of finding errors in the GUI, or front end, of your application. This important layer serves as the "curb appeal" of your site, so detecting and correcting errors here are critical to presenting a quality, robust website. If your customers encounter errors in this layer, they may not return. They may conclude, for example, that if your company posts Web pages with misspelled words, it cannot be trusted to successfully execute a credit card transaction.

In a nutshell, presentation layer testing is very labor intensive. However, just as you can segment the testing of an Internet application into discrete entities, you can do the same when testing the presentation layer. Here are the three major areas of presentation layer testing:

1. *Content testing*. Overall aesthetics, fonts, colors, spelling, content accuracy, default values.
2. *Website architecture*. Broken links or graphics.
3. *User environment*. Web browser versions and operating system configuration.

Content testing involves checking the human-interface element of a website. You need to search for errors in font type, screen layout, colors, graphic resolutions, and other features that directly affect the end-user experience. In addition, you should verify the accuracy of the information on your website. Providing grammatically correct, but inaccurate, information harms your company's credibility as much as any other GUI bug. Inaccurate information may also cause legal problems for your company.

Test the website architecture by trying to find navigational and structural errors. Search for broken links, missing pages, wrong files, or anything that sends the user to the wrong area of the site. These errors can occur very easily, especially for dynamic websites and during development or upgrade phases. All a project team member needs to do is rename a file, and its hyperlink becomes invalid. Similarly, if a graphic element is renamed or moved, then a hole will exist in your Web page because the file cannot be found. You can validate your website's architecture by creating a unit test that checks each page for architectural problems. As a best practice, you should migrate architecture testing into the

regression-testing process as well. Numerous tools exist that can automate the process of verifying links and checking for missing files.

White-box testing techniques are useful when testing website architecture. Just as program units have decision points and execution paths, so do Web pages. Users may click on links and buttons in any order, which will navigate to another page. For large sites, there exist many combinations of navigation events that can occur. Review Chapter 4 for more information on white-box testing and logic coverage theory.

As mentioned earlier, testing the end-user environment—also known as *browser-compatibility testing*—is often the most challenging aspect of testing Internet-based applications. The combination of browsers and an operating system (OS) is very large. Not only should you test each browser configuration, but different versions of the same browser as well. Vendors often improve some feature of their browsers with each release, which may or may not be compatible with older versions. It is interesting (and frustrating) to see that even in this era of advanced Internet development and functionality, you can still encounter Web pages that display a message saying the site is not compatible with the Web browser you are using. It should not be the user's responsibility to choose the right browser to access your site. To ensure a successful user visit, spend extra time in application design, development, and testing with a wide variety of browsers and operating systems.

User environment testing becomes more convoluted when your application relies heavily on client-side script processing. Every browser has a different scripting engine or virtual machine to run scripts and code on the client's computer. Pay particular attention to browser-compatibility issues if you use any of the following:

- ActiveX controls
- JavaScript
- VBScript
- Java applets
- HTML 5
- Adobe Flash
- PHP

You can overcome most of the challenges associated with browser compatibility testing by generating well-defined functional requirements. For

example, during the requirements-gathering phase, your marketing department may decide that the application should be certified to work only with certain browsers. On the one hand, this requirement eliminates a significant amount of testing because you will have a well-defined target platform to test against. On the other hand, while this might be a cost- and time-saving decision, it may not be a smart business decision. The days when a single (or even a few) Web browser applications dominated the user community are long past. Good business practice would be to design and test against a wide range of possible user Web browser applications.

Business Layer Testing

Business layer testing focuses on finding errors in the business logic of your Internet application. You will find testing this layer very similar to that of stand-alone applications, in that you can employ both white- and black-box techniques. You will want to create test plans and procedures that detect errors in the application's performance specification, data acquisition, and transaction processing.

You should employ white-box approaches for components developed in-house, because you have access to the program logic. For third-party components, however, black-box testing techniques should comprise your primary testing approach. You will start by developing test drivers to unit-test the individual components. Next, you can perform a system test to determine whether all the components work together correctly.

When conducting a system test for this layer, you need to mimic the steps a user performs when purchasing a product or service. For example, for an e-commerce site you may need to build a test driver that searches inventory, fills a shopping cart, and checks out. Pragmatically modeling these steps can prove challenging.

The technologies that you use to build the business logic dictate how you build and conduct your tests. There are numerous technologies and techniques you may use to build this layer, which makes it impossible to suggest a cookie-cutter testing method. For instance, you might architect your solution using a dedicated application server such as JBoss. Or you could have stand-alone CGI modules written in C, Python, or Perl.

Regardless of your approach, there exist certain characteristics of your application that you should always test. These areas include the following:

- *Performance.* Test to see whether the application meets documented performance specifications (generally specified in response times and throughput rates).
- *Data validity.* Test to detect errors in data collected from customers.
- *Transactions.* Test to uncover errors in transaction processing, which may include credit card processing, e-mailing verifications, and calculating sales tax.

Performance Testing A poorly performing Internet application raises doubt in your user's mind about its robustness, and often turns the person away. Lengthy page loads and slow transactions are typical examples. To help achieve adequate performance levels, you need to ensure that operational specifications are written during the requirements-gathering phase. Without written specifications or goals, you cannot know whether your application performs acceptably. Operational specifications are often stated in terms of response times or throughput rates. For instance, a page should load in x seconds, or the application server will complete y credit card transactions per minute.

A common approach you may use when evaluating performance is stress testing. Often, performance degrades to the point of being unusable when the system becomes overloaded with requests. This might cause time-sensitive transactional components to fail. If you perform financial transactions, then component failures could cause you or your customer to lose money. The concepts on stress testing presented in Chapter 6 apply to testing business layer performance.

As a quick review, stress testing involves blasting the application with multiple logins, and simulating transactions to the point of failure so you can determine whether your application meets its performance objectives. Of course, you need to model a typical user visit for valid results. Just loading the homepage does not equate to the overhead of filling a shopping cart and processing a transaction. You must fully tax the system to uncover processing errors.

Stress-testing the application also allows you to investigate the robustness and scalability of your network infrastructure. You may think that

your application has bottlenecks that allow only *x* transactions per second. But further investigation shows that a misconfigured router, server, or firewall is throttling bandwidth. Therefore, you should ensure that your supporting infrastructure components are in order before beginning stress testing. Not doing so may lead to erroneous results.

Data Validation An important function of the business layer is to ensure that data collected from users are valid. If your system operates with invalid information, such as erroneous credit card numbers or malformed addresses, then egregious errors may occur. If you are unlucky, the errors could have financial implications for you and your customers. You should test for data collection errors much like you search for user-input or parameter errors when testing stand-alone applications. Refer to Chapter 5 for more information on designing tests of this nature.

Transactional Testing Your e-commerce site must process transactions correctly 100 percent of the time. No exceptions. Customers will not tolerate failed transactions. Besides a tarnished reputation and lost customers, you may also incur legal liabilities associated with failed transactions.

You can consider transactional testing as system testing of the business layer. In other words, you test the business layer from start to finish, trying to uncover errors. Once again, you should have a document specifying exactly what constitutes a transaction. Does it include a user searching a site and filling a shopping cart, or does it consist only of processing the purchase?

For a typical Internet application, the transaction component is more than completing a financial transaction (such as processing credit cards). Typical events related to customer transactions include:

- Searching inventory.
- Collecting items the user wants to purchase.
- Presenting the user with related items that might be of interest.
- Presenting users with product or company reviews from other users.
- Soliciting and capturing product or company reviews from the current user.
- Creating or accessing a user account.

- Purchasing items, which may involve calculating sales tax and shipping costs, as well as processing financial transactions.
- Notifying the user of the completed transaction, usually via e-mail.

In addition to testing internal transaction processes, you must test the external services, such as credit card validation, banking, and address verification. You typically will use third-party components and well-defined interfaces to communicate with financial institutions when conducting financial transactions. Don't assume these items work correctly. You must test and validate that you can communicate with the external services and that you receive correct data back from them.

Data Layer Testing

Once your site is up and running, the data you collect become very valuable. Credit card numbers, payment information, and user profiles are examples of the types of data you may collect while running your e-commerce site. Losing this information could prove disastrous and crippling to your business. Therefore, you should develop a set of procedures to protect your data storage systems.

Testing the data layer consists primarily of testing the database management system that your application uses to store and retrieve information. Smaller sites may store data in text files or open-source databases. Larger, more complex sites, use full-featured enterprise-level databases. Depending upon your needs, you may use both approaches.

One of the biggest challenges associated with testing this layer is duplicating the production environment. You must use equivalent hardware platforms and software versions to conduct valid tests. In addition, once you obtain the resources, both financial and labor, you must develop a methodology for keeping production and test environments synchronized.

As with the other tiers, you should search for errors in certain areas when testing the data layer. These include the following:

- *Response time*. Quantifying completion times for Structured Query Language (SQL) operations.
- *Data integrity*. Verifying that the data are stored correctly and accurately.

- *Fault tolerance and recoverability*. Maximizing the MTBF and minimizing the MTTR.

Response Time Testing Slow e-commerce applications cause unhappy and untrusting customers. Thus, it is in your interest to ensure that your website responds in a timely manner to user requests and actions. Response time testing in this layer does not include timing page loads; rather, it focuses on identifying database operations that do not meet performance objectives. When testing the data-tier response time, you want to ensure that individual database operations occur quickly so as not to bottleneck other operations.

That said, before you can measure database operations, you must understand what constitutes one. For this discussion, a database operation involves inserting, deleting, updating, or querying data from the RDBMS. Measuring the response time simply consists of determining how long each operation takes. You are not interested in measuring transactional times, as that may involve multiple database operations. Profiling transaction speeds occurs while testing the business layer.

Because you want to isolate problem database operations, you do not want to measure the speed of a complete transaction when testing data layer response times. Too many factors may skew the test results if you test the whole transaction. For example, if it takes a long time for users to retrieve their profiles, you need to determine where the bottleneck for that operation resides. Is it the SQL statement, Web server, or firewall? Testing the database operation independently allows you to identify the problem. In this example, if the SQL statement is poorly written, it will reveal itself when you test response time.

Data layer response-time testing is plagued with challenges. You must have a test environment that matches what you use in production; otherwise, you may get invalid test results. Also, you must have a thorough understanding of your database system to make certain that it is set up correctly and operating efficiently. You may find that a database operation is performing poorly because the RDBMS is configured incorrectly.

Generally speaking, though, you perform most response-time testing using black-box methods. All you are interested in is the elapsed time for database transactions. Many tools exist to help with these efforts, or you may write your own.

Data Integrity Testing

Data integrity testing is the process of finding inaccurate data in your data stores. This test differs from data validation, which you conduct while testing the business layer. Data validation testing tries to find errors in data collection. Data integrity testing strives to find errors in how you store data.

Many factors can affect how the database stores data. The datatype and length can cause data truncation or loss of precision. For date and time fields, time zone issues come into play. For instance, do you store time based on the location of the client, the Web server, the application server, or the RDBMS? Internationalization and character sets can also affect data integrity. For example, multibyte character sets can double the amount of storage required, plus they can cause queries to return padded data.

You should also investigate the accuracy of the reference tables used by your application, such as sales tax, zip codes, and time zone information. Not only must you ensure that this information is accurate, you must keep it up to date.

Fault Tolerance and Recoverability Testing

If your e-commerce site relies on an RDBMS, then the system must stay up and running. There is very little, if any, downtime availability in this scenario. Thus, you must test the fault tolerance and recoverability of your database system.

One goal of database operations, in general, is to maximize MTBF and minimize MTTR. You should find these values specified in the system requirements documentation for your e-commerce site. Your goal when testing the database system robustness is to try to exceed these numbers.

Maximizing MTBF depends on the fault-tolerance level of your database system. You might have a failover architecture that allows active transactions to switch to a new database when the primary system fails. In this case, your customers might experience a short service disruption, but the system should remain usable. Another scenario is that you build fault tolerance into your application so that a downed database affects the system very little. The types of tests you run depend on the architecture.

You should consider database recovery as equally important. The objective of recoverability testing is to create a scenario in which you cannot recover that database. At some point, your database will crash, so you need to have procedures in place to recover it very quickly. The planning for recovery begins in obtaining valid backups. If you cannot recover the database during recoverability testing, then you need to modify your

backup plan. A fault-tolerant database system may reside in multiple locations connected over a private or shared network. This aspect of database management must be tested as well. If the local server fails, are the remote systems current, and can your software connect to a remote system quickly? What happens if one or more network connections fail? What happens if a system failure occurs while data is being written?

In general, strive to test all aspects of the system, everything required to support all levels of activity and data integrity for which your application is designed.

Summary

The public Internet did not exist when the first edition of this book was written. Indeed, remotely accessed systems, and applications in general, were infantile compared to those of today's Internet. Users in those early days mostly were sophisticated, computer-savvy folk who could tolerate a fairly high level of difficulty in accessing and using remote applications. Today, Internet users may know very little about the actual operation of computers and computer software, yet they have a virtually infinite choice of commercial sites from which to choose. Consequently, they have little or no patience for a Web-based application that is unattractive, difficult to use, or dysfunctional. Therefore, in-depth testing of any Internet application is extremely important.

Testing software in the Internet environment presents many challenges, particularly the large and varied user base and the need for extreme accuracy and security for electronic commerce applications. In general, we want to test three main Internet application areas: presentation (or user interface), business logic, and data management. As might be expected, large user-base applications require extensive user testing (see Chapter 7 for more information on this process) to ensure that the software meets design specifications and user acceptance criteria. It is important for any software application to be attractive and easy to use, but applications for the Internet are judged more harshly. In this environment, software success often equates to business success, and this factor alone should drive developers toward aggressive and thorough testing.

11 Mobile Application Testing

Computer technology changes rapidly. In a blink of an eye the computer went from the desktop to the laptop and now to the handheld mobile device. This migration has changed the way we conduct our lives, businesses, and governments. It has also significantly affected the way software developers and testers do their jobs.

Most software testing professionals find testing mobile applications very challenging—more so than almost any other software types or platforms. Actually, it's the devices and mobile environment more than the "application" that impose the challenge. These two components add many variables and complexities that may skew or mask problems in your application, which makes designing a robust test plan difficult. Briefly, you need to consider network performance and reliability, consistent user interfaces, transcoder influences, device diversity, and limited resource platforms.

In this chapter, we introduce a relatively new area of software testing: testing mobile and smartphone applications. We begin by describing the mobile application environment, which differs from that of a stand-alone application on desktops, laptops, and servers. Next, we enumerate the challenges of testing mobile applications—some of which we touched on earlier in this book. Finally, we cover some testing approaches and test-case considerations to help lower your learning curve in this new territory. After reading this chapter you should better understand the challenges and hurdles of testing mobile application.

Mobile Environment

With the widespread rollout of wireless hotspots, the line between mobile computing and "traditional" wireless network-based activities has blurred. Thus, to begin here we need to define the terms *mobile device* and *mobile applications*, with respect to the content of this chapter. In that light, we refer to a mobile device as one that has the capability to run network-based applications over a cellular or satellite data link. This encompasses most smartphones, tablets, and PDAs. That said, don't make the mistake of identifying mobile devices only by their appearance. Modern laptops can accept plug-in cellular or satellite cards, and some laptop devices have this access built in. Based on this definition of a mobile device then, a mobile application is a network-based program that runs on a mobile device.

This distinction is important. Yes, it is true that most mobile devices can use hotspots and wireless access points without a problem. However, those connections provide greater reliability and higher speeds than cellular networks, even with the adoption of 3G and 4G technologies. Thus, you design your mobile application with the expectation that it will use relatively slow, comparatively unreliable data links. You can also develop stand-alone applications, such as games, that run on a mobile device without the need to use the carrier's network. But for the purposes of this chapter, we do not consider stand-alone applications as mobile applications. Our focus is on the challenges associated with applications running on cellular data networks.

The key to creating successful test plans for your mobile applications is to understand the mobile computing environment. Table 11.1 identifies a number of crucial areas you should investigate while designing test plans. First, you must understand device connectivity issues and network speeds, regional availability, and latency. Keep in mind the underlying philosophy of this book: Your tests should not prove that your application works, but that your application does *not work* for the use cases. For example, if you have a location-based service or e-mail application, then your tests should identify software problems when the carriers network is slow or unavailable.

Next are three areas regarding devices—diversity, constraints, and input methods—which we cover in great detail later in the chapter. To create successful test plans, you and your testing staff must consider the numerous devices in the marketplace, the varying capabilities of each, and how the user interacts with the devices.

TABLE 11.1 Mobile Environment Test Design Considerations

Area	Comment
Connectivity	Device provisioning Network speed Network latency Network availability in remote areas Service reliability
Diversity Devices	Numerous web browsers to test Multiple versions of runtimes for Java or other languages
Device Donstraints	Limited memory or processor Small screen size Multiple operating systems Multitasking capabilities Data cache sizes
Input Devices	Touch screens Stylus Mouse Buttons Rollers
Installation and Maintenance	Installing and uninstalling Patching Upgrading

Last, you need to determine how you will install and maintain your application. Some vendors, such as Apple, maintain online stores where the user purchases the application, but only after Apple certifies your application for its platform. This makes installation and maintenance a little easier, as you have a single, certified distribution system.

Testing Challenges

As stated, mobile application testing is fraught with challenges. To help meet them, we can categorize most into four categories: device diversity, carrier network infrastructure, scripting, and usability. You need to think through each carefully when designing test cases. The shear combination of device types, operating systems, user input methods, and network concerns mean that trade-offs must be balanced with time, financial, and labor

resources to arrive at an economical test plan that detects most bugs in a reasonable time frame. Building a testing strategy that combines the methods discussed in earlier chapters will help.

In the rest of this section we discuss these categories and offer advice on how to tackle each one.

Mobile Device Diversity

The ever-expanding diversity of devices presents an often-underestimated and significant testing challenge to someone new to mobile application testing. It sometimes seems that manufacturers introduce new devices daily, making it almost impossible to keep up with the release cycles. Worse, more devices means more items to consider in your testing. Here's a simple example to illustrate only a few items you need to evaluate when a new device is released:

> Suppose Motorola develops a new method of text input via the touch screen for its Android-based phones. Can you design a test to determine whether the device's new input method breaks your application? If it does, can you fix your application without breaking support for other Android-based devices such as tablets? Can you even obtain a device to test? Do you have access to a supported carrier network?

Almost by definition, along with diversity of the devices comes diversity of operating systems, browsers, application runtime environments, screen resolutions, user interfaces, ergonomics, screen size, and more. You must be aware of all of these factors when creating tests. Device diversity also forces usability testing front and center, which at some point requires testers to evaluate your application on target devices. Using emulators is great way to start, but ultimately you will need to test real devices on real carrier networks.

This raises another facet of mobile application testing: testing on real devices versus emulators. From an economic standpoint you should do as much testing as you can with emulators. It may be financially unfeasible, even if you can obtain a device and access the wireless network, to test on the real platform. That said, emulators only emulate; they are not the real devices. So it is likely that you will observe differences between testing

with an emulator and the actual device. For example, the colors and shapes of buttons and input boxes may pass acceptance tests on an emulator but fail on the target device because of screen resolution and color depth differences between the device and a PC-based emulator.

In short, you need to realize there may be hundreds of mobile devices with the potential to access your application. Therefore, during the requirements-gathering and specification-writing phases, you will be called upon to make some tough decisions and choose a reasonable subset of devices to support and test. Be mindful that every device you do *not* test may not work with your application; hence, you may lose not only a customer but a customer base.

Carrier Network Infrastructure

Testing your application on a carrier network sets up another challenge. This is especially true if you want to support multiple carriers. Two of the highest hurdles to jump are: understanding and adapting to the carrier's infrastructure, and overcoming location-based obstacles.

Understanding a carrier's infrastructure is fundamental to developing a good test plan. Initially, you would think that your mobile application uses a carrier's network like an IP wireless hotspot. Not so. Figure 11.1 illustrates the "typical" infrastructure of most wireless carriers. The first difference is that the protocol is not IP-based; it is usually an RF-based protocol

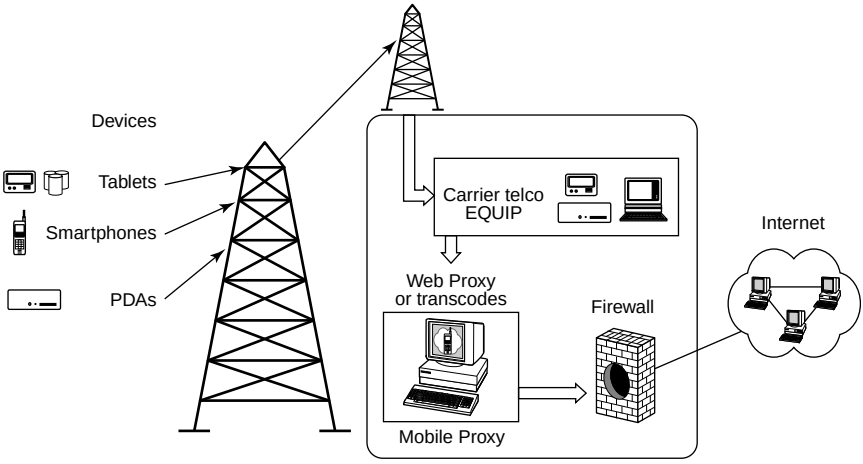

FIGURE 11.1 Generic Wireless Carrier Data Network.

such as code division multiple access (CDMA), time division multiple access (TDMA), or global system for mobile (GSM). The RF-based protocols treat the IP-based protocols as a "payload" and delivers them to the mobile device, which then decodes the payload and presents it to the application.

Also most carriers use some form of transcoder or Web proxy between the Internet and the device. These devices may perform a variety of functions. And, it is sometimes hard to determine exactly what occurs unless you work directly with the carriers. Often, they do not reveal this information for competitive purposes. The following is a short list of what may occur at a carrier's Web proxy or transcoder:

- Transform or transcode content into WAP or HTTP.
- Compress data for better throughput.
- Encrypt traffic for privacy and security.
- Block access to certain high-bandwidth sites.
- Strip HTML headers and other metadata from Web pages that your application may use.

Transcoding may cause UI inconsistencies across multiple devices. Some devices support Wireless Application Protocol (WAP) while others support HTTP. WAP uses Wireless Markup Language (WML) for content delivery. WAP and WML were intended to be the "standard" for wireless content delivery, but never gained a strong foothold. Nonetheless, numerous devices implement it, so you may encounter it during your tests. However, most smartphones and tablets support HTML and therefore rely on HTTP to deliver content. If you have UI problems across devices and carriers, check with each to determine whether WAP/WML or HTTP/HTML is being used.

Although data compression is intended to improve throughput, often during periods of high activity throughput may slow due to the overhead of compression. The same holds with security: Firewalls and similar layers may slow throughput during high-volume hours.

Finally, you must overcome location-based hurdles. Obviously, to test on a carrier's network, you need access to it. For instance, what if you have a travel application for a smartphone: How do you test carrier networks in other parts of the country or in other countries? Answer: You must travel there or hire someone there to test it for you. Both add to the cost of testing.

Scripting

An often-overlooked area of mobile applications testing is creating and running test scripts. Real devices do not allow you to load automated, repeatable scripts onto the device; test personnel manually execute all scripts. That is, someone walks through a written test script designed to find errors in a test case on the target device. Notice we said "target" device. There exist many targets in the mobile environment.

As we pointed out in previous chapters, manual testing is error prone. Unfortunately, it is unavoidable when testing mobile applications on real devices. As mentioned, most emulators have rich scripting functionality and can perform the bulk of regression testing and system tests. However, in the end you still need to have someone work with the device. (Later in the chapter, we explain how to create a generic manual test script to support multiple devices.)

The refreshing news is that mobile devices are becoming much more sophisticated and powerful. Given the competitiveness of the marketplace, it is reasonable to expect an automated scripting product to appear. Apple's iOS, Windows Mobile OS, and the Android OS are maturing rapidly, so it is likely that this problem may be a non-issue in future versions.

Usability

Usability testing presents challenges similar to those of test scripts. Recall from previous chapters that usability testing is mostly a white-box approach. Just like testing stand-alone desktop applications, a testing staff must manually try to find bugs in the user interface and user interaction layers of your application.

Unlike testing stand-alone desktop applications, mobile device testing involves more than one platform to test. For instance, you will want to search for UI consistency issues between Apple's products and the Android-based platforms. Although you are testing mobile applications, much of Chapter 7's discussions apply.

Testing Approaches

Some areas of testing mobile devices are similar to testing Internet applications, especially when evaluating the back-end infrastructures. The major

difference lies in how you approach testing the device itself. With Internet testing, you have only a handful of browsers to evaluate; with mobile devices, you have exponentially more.

Naturally, when testing back-end components, you should employ similar techniques and evaluate similar considerations as those discussed in Chapter 10, "Testing Internet Applications." Referring back to Figure 10.1, tiers 2 and 3 should have approximately the same configuration as a normal Internet application. As a quick review, you should test the performance specifications, data validation routines, and transaction processing components of tier 2. Testing tier 3 also is the same as with Internet applications; test response times, data integrity, fault tolerance, and recoverability on this tier. If possible test the tier 2 and 3 components separately from the device to ensure they meet your design specifications using function testing.

Testing tier 1, the user environment, differs from traditional Internet testing. The concepts presented on testing your content and website architecture still apply. However, user environment testing equates to device testing.

We should note the importance of use cases when developing test plans for your devices. Knowing who will use your application, and how and when, is imperative, as mobile applications have numerous points of failure. Table 11.2 lists items you might not generally consider when designing test cases for standard applications, whether stand-alone or Web-based. For example, testing your application on the carrier's network is extremely important. You want to find problems related to spotty coverage or sudden loss of connectivity. If your application involves data transfers, look for problems with data caching and incomplete synchronization with back-end data stores. What happens when coverage is suddenly restored after an interruption during an application download? Does a purchase occur twice? Check for bugs related to handling session reinitialization and data corruption. Some of these issues apply to Web-based applications running in a PC-based browser. However, LANs/WANs are much more stable. When dealing with cellular networks, you should expect to lose connectivity.

A test case specific to mobile testing is how your application handles incoming voice calls and text messages. Chances are end users will want to suspend your application, or run it in the background, while they answer the phone or read the text message. Try to build test cases where incoming calls and messages cause problems in your application.

TABLE 11.2 Test Categories for Mobile Application Testing

Test Category	Description
Install/Uninstall	Ensure the user can correctly install your application. Ensure the user can completely uninstall your application.
Network Infrastructure	Verify the application responds appropriately to loss of network. Verify the application responds appropriately to network restoration. Verify the application responds appropriately to weak signals.
Incoming Call/ Message Handling	Test whether user can accept calls/text messages while application is running. Test whether user can resume application when finishing calls/text messages. Test whether user can reject calls/text messages without disrupting application. Test whether user can initiate a call/text message without disrupting the application.
Low Memory	Ensure application remains stable when device encounters a low memory situation.
Key Mappings	Test that all key mapping works as specified.
Feedback	Ensure user feedback to keypress occurs within application design specifications.
Exiting	Verify that the application exits gracefully when initiated through pressing keys, closing the cover, or using the slider. Confirm the application meets design specifications when the user initiates a shutdown of device.
Charging	Ensure that application works as designed when entering charge mode. Ensure that application works as designed while in charge mode. Ensure that application works as designed when exiting charge mode.
Battery Conditions	Test how the application behaves on a low battery. Measure how quickly application drains the battery. Ensure the application responds per specification when the battery is removed while the device is powered on.
Device Interaction	Ensure the application does not overload the CPU. Ensure the application does not consume too much memory.

TABLE 11.3 Devices versus Emulators

Testing Approach	Disadvantages	Advantages
Real Devices	Expensive, especially if you target a broad base of mobile devices	Ability to test responsiveness of the application
	Inability to install metering or diagnostic development tools	Visual inspection of application on real device to verify UI consistency
	Unable to install on run test scripts	Test carriers' network responsiveness
	Network availability	Identify device-specific bugs
Emulators	Inability to identify device- related bugs	Cost-effective
	Underlying hardware may skew performance on real device	Easy to manage; multiple device support with single emulator

In the rest of the chapter we will cover some approaches to device testing in which you basically have two choices: test on real devices or use device emulators. Table 11.3 offers some advantages and disadvantages of each approach.

Testing with Real Devices

Manual testing with real devices is inevitable. Although costly, it has some advantages. Only by testing with the device can you experience its nuances and get a true feel for the user's experience. In addition, you can only test certain cases with real devices. Testing the reliability of a carrier's network and determining the effect of an incoming call or text message are obvious examples. On a real device you also can evaluate how your application behaves. Does it load fast and run at an acceptable speed? Does it look okay? Is the UI consistent across your target devices? Last but not least, you can determine device-specific bugs. This is almost impossible with an emulator. If you do find a device-specific bug, the challenge is to fix it without breaking compatibility with other devices.

Despite the advantages, testing with real devices also has some serious drawbacks. For example, it is costly because you must purchase the device,

as well as pay for carrier airtime. Neither is inexpensive, and if you are testing multiple devices from multiple carriers in multiple regions, the expenses grow accordingly. Some device manufacturers and service providers have devices that you can rent or access remotely, which may mitigate some costs. If you target an individual platform, such as the Apple iPhone family, you may be spared much of this expense. Still, you will need enough of each type (iPad, iPhone, iTouch) to test.

In addition, testing with real devices is a manual, white-box process. Someone must push the buttons, tap the screens, and enter data. As you know, manual testing is error prone, even with the best instructions and trained testers. Plus, it adds another expense to the process. You should keep exacting notes about each well-documented test script and its results. Then evaluate the effectiveness of the scripts and eliminate those with little or no value (i.e., fail to find bugs).

As we noted earlier, using real devices eliminates one important weapon in the software tester's arsenal: automated test scripts. Therefore, you should use written, manual scripts that specify generic actions, not details on how to perform the action on a device. Detailed test scripts for every device would be a challenge to create and maintain. In short order you would have a library of scripts, which may be obsolete when the device is updated. Generic scripts allow you to test system specifications across multiple devices.

For example, iPhones, iPads, and Android-based devices rely heavily on touch screens for user input. Other devices, such as BlackBerries or "standard" phones, have keyboards or keypads to allow for user input. Table 11.4 provides an example script to check whether your application, an e-reader, aborts if you receive a text message while reading an e-book. Notice the script does not specify exactly how to do any one step, only to perform the step using the user-input facilities of the device. These may be buttons, touch screens, or voice commands. At no point do you specify "Press OK" or "Press Send." This generic approach will allow you to evaluate test cases across multiple devices.

Last, manufacturers often "lock down" real devices, meaning you cannot load tools to monitor or debug your application. So when you hit a bug it is more challenging to isolate the problem. For instance, if your application is running slowly, you do not know whether it is the carrier's network, transcoding issues, your application, or a combination. Only by trial and error can you identify problems.

TABLE 11.4 Generic Device Test Script

1. Start e-reader application.
2. Open e-book.
3. Initiate SMS message to device from another device.
4. Verify SMS message alert is displayed.
5. Open SMS message.
6. Choose Reply to SMS message.
7. Compose SMS message.
8. Send SMS message.
9. Verify SMS message sent notification.
10. Return to e-book.
11. Verify e-book application is running.
12. Verify return to same page or bookmark.
13. Exit e-reader application.

Testing with Emulators

Testing with emulators may not be the preferred approach, but it is usually the most practical and cost-effective, and it even has some advantages. First, emulators allow for inexpensive and quick functional testing of your application. You can step through the application to find events and circumstances that do not meet the program requirements. Identify these bugs using emulators before you get into the expense of device testing.

Second, emulators are easy to manage, and because they run on PCs, every tester or developer can have an emulator. Developers can manage the software themselves, precluding the need of system administrators. Third, most emulator packages support multiple devices. To test a different device, just load a different device profile. Best of all, you incur no expensive carrier airtime costs. Fourth, emulators run on computers with more resources, such as faster CPUs and more memory. Fast response times during testing enables you to complete tests more quickly.

The last and probably most significant advantage is that most emulators employ high-level scripting languages, so you can create consistent, automated tests, which are less error prone and quicker than manual testing. Automated scripting also allows for easier and faster regression testing, which is especially important when verifying that changes made to your

application to support one device don't break support for another. The scripting languages in emulators generally are device-agnostic. Referring to Table 11.4, when you script Step 8, "Send SMS Message," the emulator will perform that function regardless of the device. This allows scripts to be used across devices.

The disadvantage of using emulators for testing is that you cannot identify the nuances and bugs of each device. As we've said before, at some point, you must test your application on the target devices. Without testing on real devices, you never can be 100 percent sure that you meet compatibility and performance specifications. Nonetheless, do not rule out using emulators for the bulk of your testing. It is a cost-effective and efficient way to eliminate most of your bugs.

Summary

Mobile application testing represents a new frontier in software testing. The mobile environment adds greater complexity and more interactions not experienced when testing standard stand-alone applications. That said, with an understanding of the challenges, you can greatly improve your chances of successfully testing your application.

Begin by trying to gain a handle on the device universe you want to support. Do you want to support only Android-based smartphones and tablets, or go for broke and support most major tablet and smartphone vendors? Next, understand the carrier's network infrastructure. Does it transcode, encrypt, compress, or in any way modify the data before sending it to the device?

You also need to find a balance between emulator and real device testing. Both have their pros and cons. Due to costs, you will likely use emulators more, and save device testing for the final phases. Use the test categories in Table 11.2 as a starting point to developing your own. Refer to the categories often when defining your test cases. Also, treat any written test script and result like source code; ensure you have adequate backups and some form of change control on the test documents. To save time and money, review the effectiveness of each script and eliminate ones that fail to add value.

Once you understand the fundamentals of mobile application testing, you should have no problems creating test plans and use cases. One thing is certain, mobile applications are here, and sooner or later you will need to learn how to test these unique applications. Why not start now?

Appendix
Sample Extreme Testing Application

1. check4Prime.java

To compile:

```
&> javac check4Prime.java
```

To run:

```
$> java -cp check4Prime 5
```

Right . . . 5 is a prime number!

```
$> java -cp check4Prime 10
```

Sorry . . . 10 is NOT a prime number!

```
$> java -cp check4Prime A
Usage: check4Prime x
       - where 0<=x<=1000
```

Source code:

```
//check4Prime.java
//Imports
import java.lang.*;

public class check4Prime {

  static final int max = 1000;    // Set upper bounds.
  static final int min = 0;       // Set lower bounds
  static int input =0;            // Initialize input variable

public static void main (String [] args) {

    //Initialize class object to work with
    check4Prime check = new check4Prime();

    try{
    //Check arguments and assign value to input variable
       check.checkArgs(args);
```

```
      //Check for Exception and display help
      }catch (Exception e){
         System.out.println("Usage: check4Prime x");
         System.out.println("       - where 0<=x<=1000");
     System.exit(1);
    }
    //Check if input is a prime number
    if (check.primeCheck(input))
       System.out.println("Right... " + input + " is a prime number!");
    else
       System.out.println("Sorry... " + input + " is NOT a prime number!");

} //End main

//Calculates prime numbers and compares it to the input
public boolean primeCheck (int num){

    double sqroot = Math.sqrt(max);    // Find square root of n

//Initialize array to hold prime numbers
boolean primeBucket [] = new boolean [max+1];

//Initialize all elements to true, then set non-primes to false
for (int i=2; i<=max; i++){
   primeBucket[i]=true;
}


//Do all multiples of 2 first
int j=2;
for (int i=j+j; i<=max; i=i+j){        //start with 2j as 2 is prime
   primeBucket[i]=false;              //set all multiples to false
}
for (j=3; j<=sqroot; j=j+2){           // do up to sqrt of n
   if (primeBucket[j]==true){          // only do if j is a prime
    for (int i=j+j; i<=max; i=i+j){    // start with 2j as j is prime
      primeBucket[i]=false;            // set all multiples to false
    }
  }
}
//Check input against prime array
    if (primeBucket[num] == true) {
     return true;
    }else{
     return false;
    }

}//end primeCheck()
```

```
//Method to validate input
public void checkArgs(String [] args) throws Exception{
    //Check arguments for correct number of parameters
    if (args.length != 1) {
       throw new Exception();
    }else{
     //Get integer from character
     Integer    num = Integer.valueOf(args[0]);
     input = num.intValue();

    //If less than zero
    if (input < 0)              //If less than lower bounds
      throw new Exception();
    else if (input > max)       //If greater than upper bounds
      throw new Exception();
    }
  }

}//End check4Prime
```

2. check4PrimeTest.java
Requires the JUnit api, junit.jar

To compile:

```
$> javac -classpath .:junit.jar check4PrimeTest.java
```

To run:

```
$> java -cp .:junit.jar check4PrimeTest
```

Examples:

Starting test . . .

. . . . . .

Time: 0.01

OK (7 tests)

Test finished . . .

Source code:

```
//check4PrimeTest.java
//Imports
import junit.framework.*;

public class check4PrimeTest extends TestCase{

  //Initialize a class to work with.
  private check4Prime check4prime = new check4Prime();

  //constructor
  public check4PrimeTest (String name){
    super(name);
  }
```

```
//Main entry point
public static void main(String[] args) {
    System.out.println("Starting test...");
    junit.textui.TestRunner.run(suite());
    System.out.println("Test finished...");
} // end main()
//Test case 1
public void testCheckPrime_true(){
    assertTrue(check4prime.primeCheck(3));
}

//Test cases 2,3
public void testCheckPrime_false(){
    assertFalse(check4prime.primeCheck(0));
    assertFalse(check4prime.primeCheck(1000));
}

//Test case 7
public void testCheck4Prime_checkArgs_char_input(){
    try {
      String [] args= new String[1];
      args[0]="r";
      check4prime.checkArgs(args);
      fail("Should raise an Exception.");
    } catch (Exception success){
      //successfull test
    }
} //end testCheck4Prime_checkArgs_char_input()

//Test case 5
public void testCheck4Prime_checkArgs_above_upper_bound(){
    try {
      String [] args= new String[1];
      args[0]="10001";
      check4prime.checkArgs(args);
      fail("Should raise an Exception.");
    } catch (Exception success){
      //successfull test
    }
} // end testCheck4Prime_checkArgs_upper_bound()

//Test case 4
public void testCheck4Prime_checkArgs_neg_input(){
    try {
      String [] args= new String[1];
      args[0]="-1";
      check4prime.checkArgs(args);
      fail("Should raise an Exception.");
    } catch (Exception success){
      //successfull test
    }
}// end testCheck4Prime_checkArgs_neg_input()

//Test case 6
public void testCheck4Prime_checkArgs_2_inputs(){
    try {
      String [] args= new String[2];
      args[0]="5";
      args[1]="99";
```

```
            check4prime.checkArgs(args);
            fail("Should raise an Exception.");
          } catch (Exception success){
            //successfull test
          }
    } // end testCheck4Prime_checkArgs_2_inputs
    //Test case 8
    public void testCheck4Prime_checkArgs_0_inputs(){
          try {
            String [] args= new String[0];
            check4prime.checkArgs(args);
            fail("Should raise an Exception.");
          } catch (Exception success){
            //successfull test
          }
    } // end testCheck4Prime_checkArgs_0_inputs

    //JUnit required method.
    public static Test suite() {
          TestSuite suite = new TestSuite(check4PrimeTest.class);
          return suite;
    }//end suite()

} //end check4PrimeTest
```

Index