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;
    }
}
```

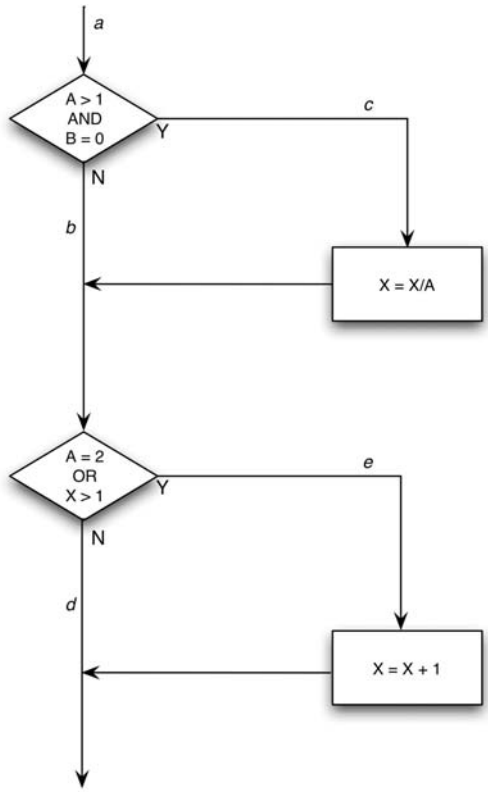

FIGURE 4.1 A Small Program to Be Tested.

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,

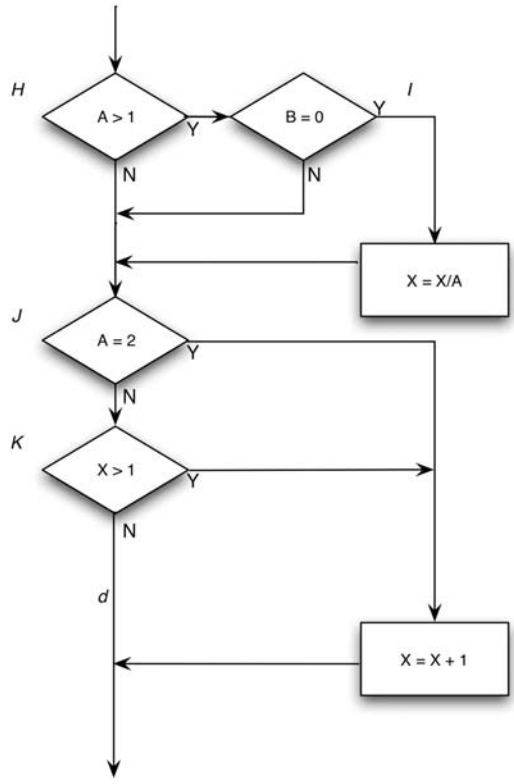

FIGURE 4.2 Machine Code for the Program in Figure 4.1.

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.

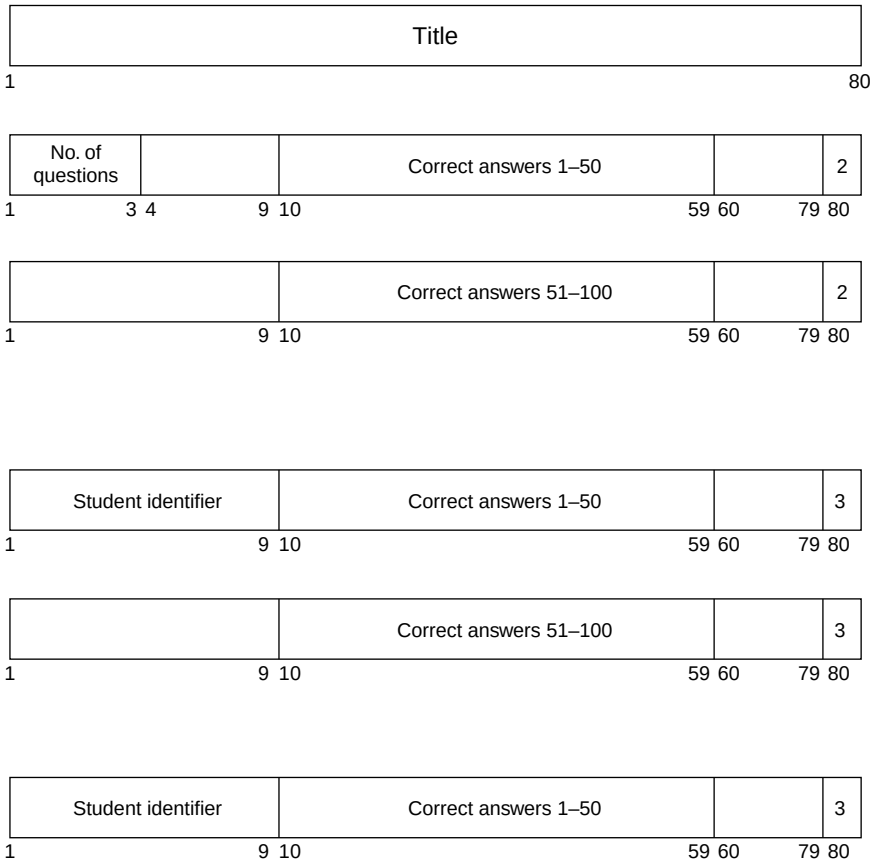

FIGURE 4.4 Input to the MTEST Program.

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.

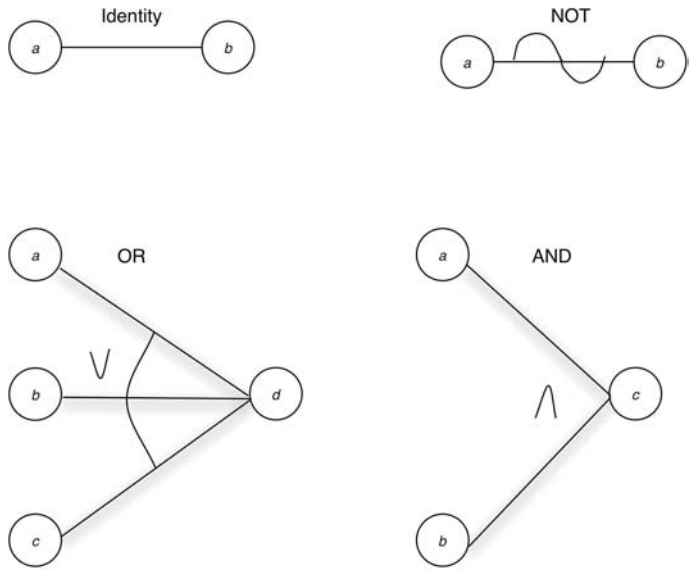

FIGURE 4.5 Basic Cause-Effect Graph Symbols.

- 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

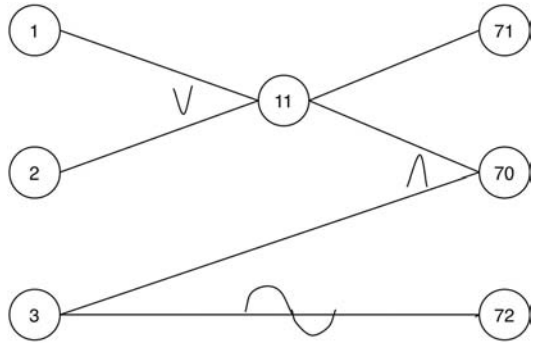

FIGURE 4.6 Sample Cause-Effect Graph.

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).

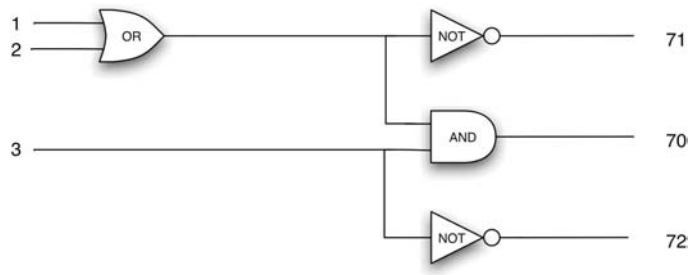

FIGURE 4.7 Logic Diagram Equivalent to Figure 4.6.

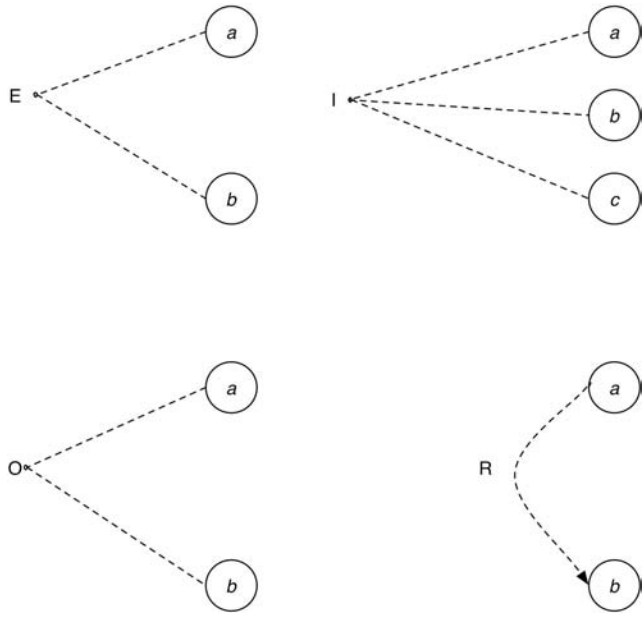

FIGURE 4.8 Constraint Symbols.

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.

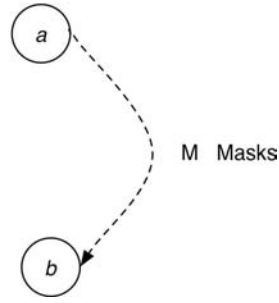

FIGURE 4.9 Symbol for "Masks" Constraint.

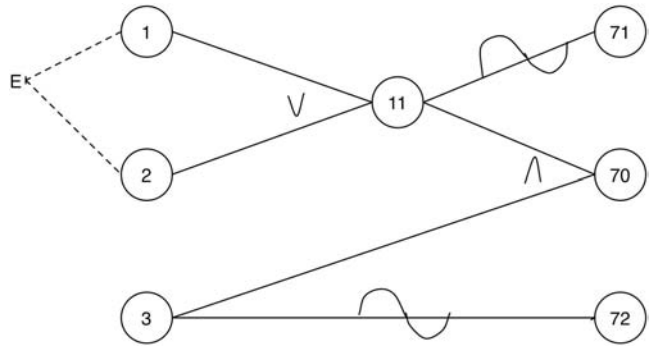

FIGURE 4.10 Sample Cause-Effect Graph with "Exclusive" Constraint.

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).

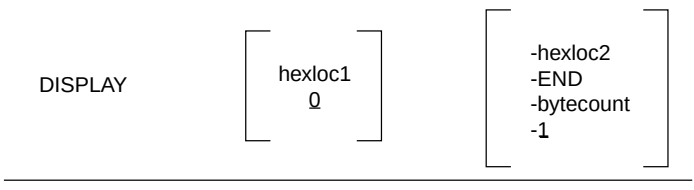

FIGURE 4.11 Syntax of the DISPLAY Command.

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

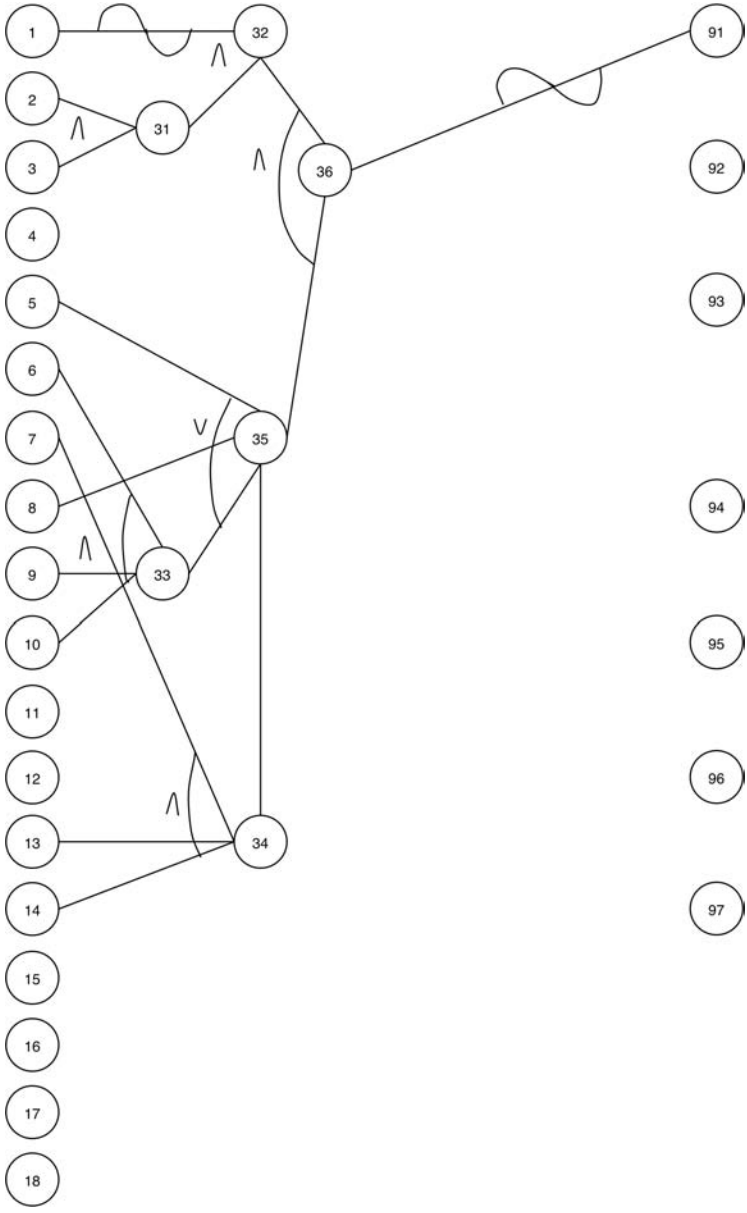

FIGURE 4.12 Beginning of the Graph for the DISPLAY Command.

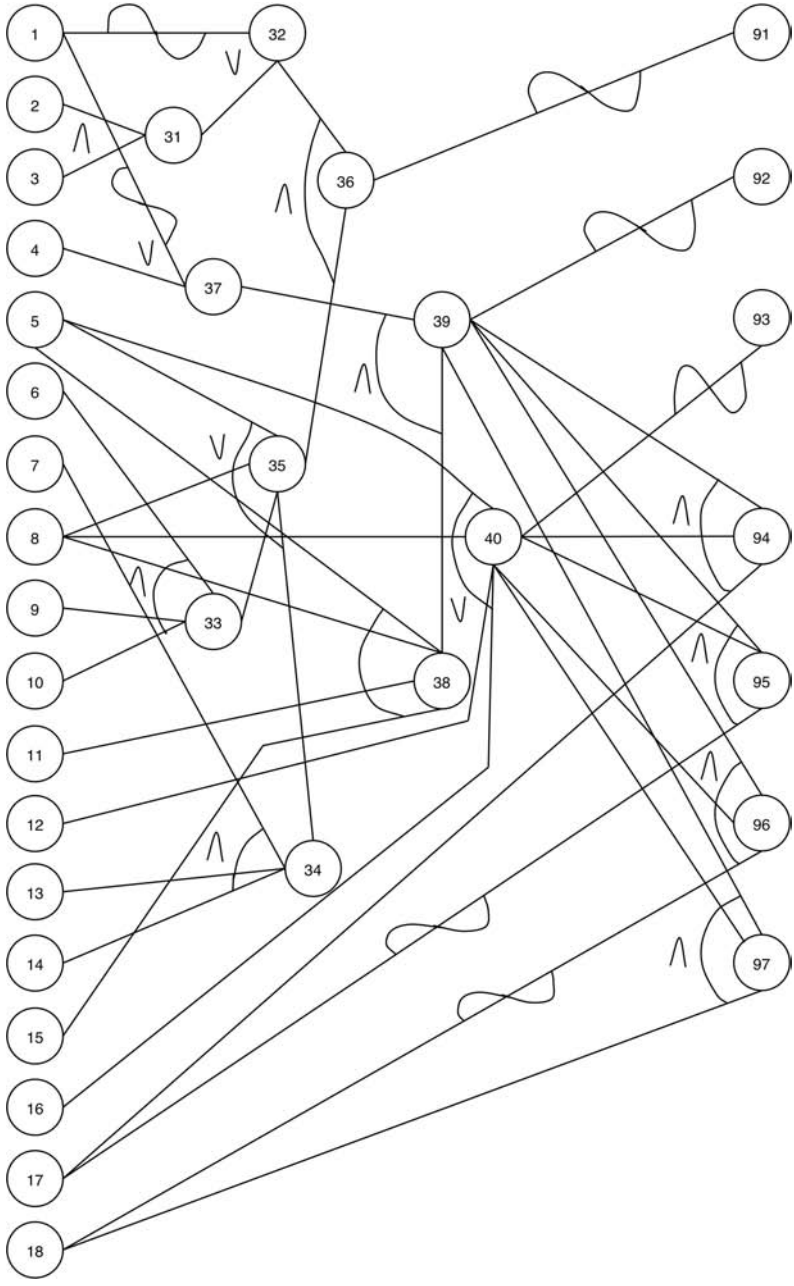

FIGURE 4.13 Full Cause-Effect Graph without Constraints.

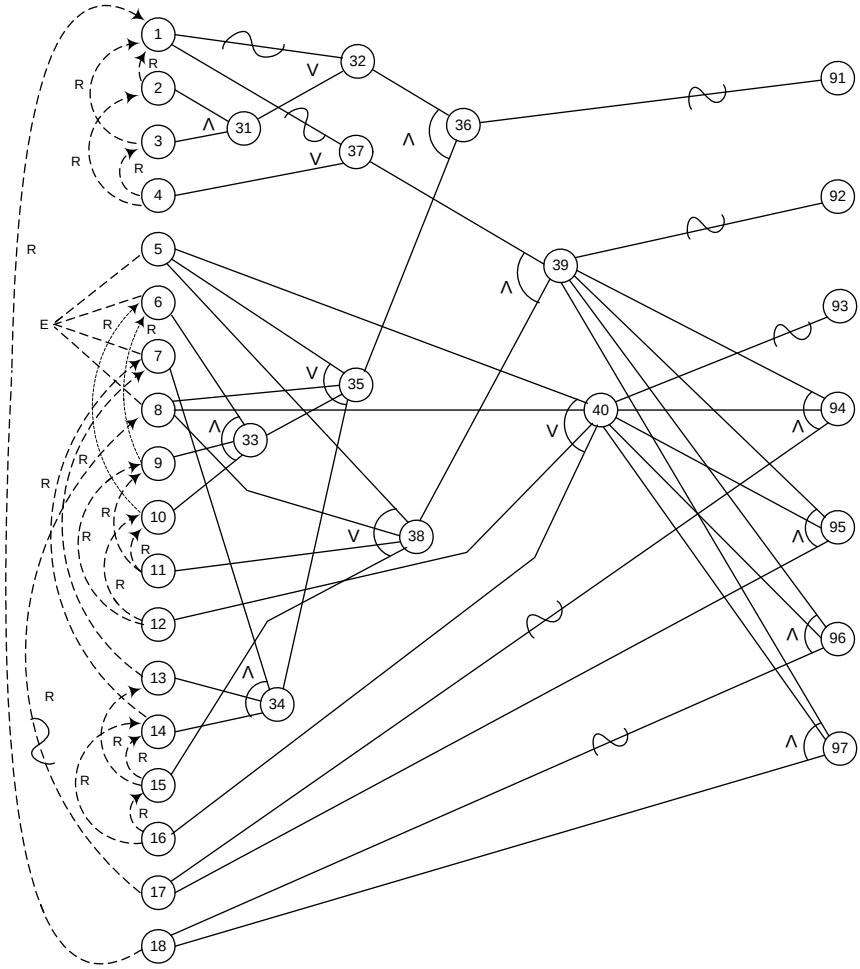

FIGURE 4.14 Complete Cause-Effect Graph of the DISPLAY Command.

(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.

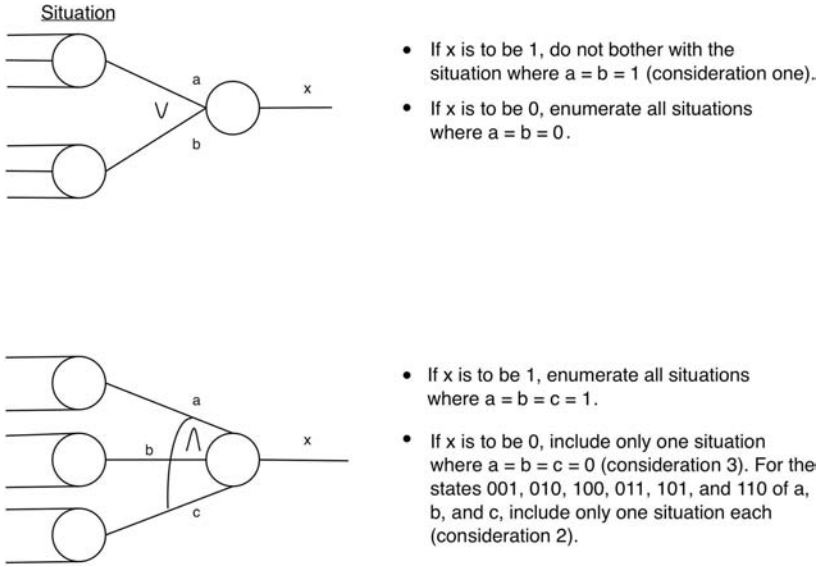

FIGURE 4.15 Considerations Used When Tracing the Graph.

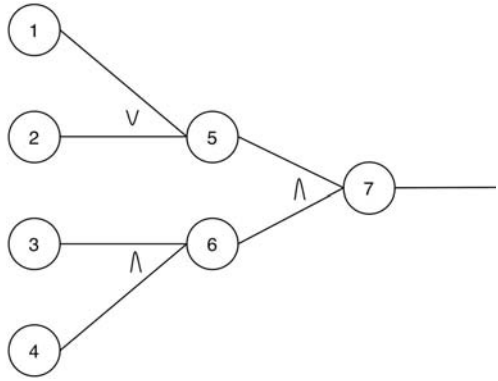

FIGURE 4.16 Sample Graph to Illustrate the Tracing Considerations.

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.

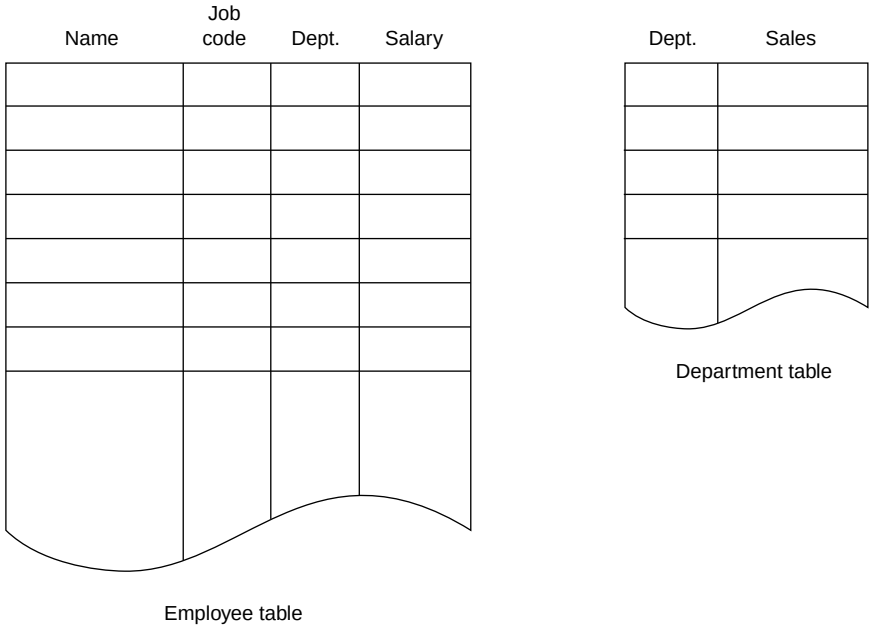

FIGURE 5.1 Input Tables to Module BONUS.

```
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

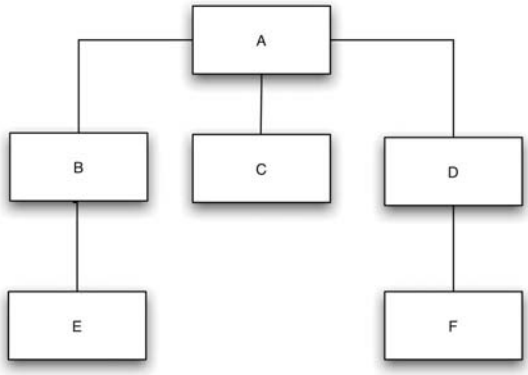

FIGURE 5.7 Sample Six-Module Program.

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.

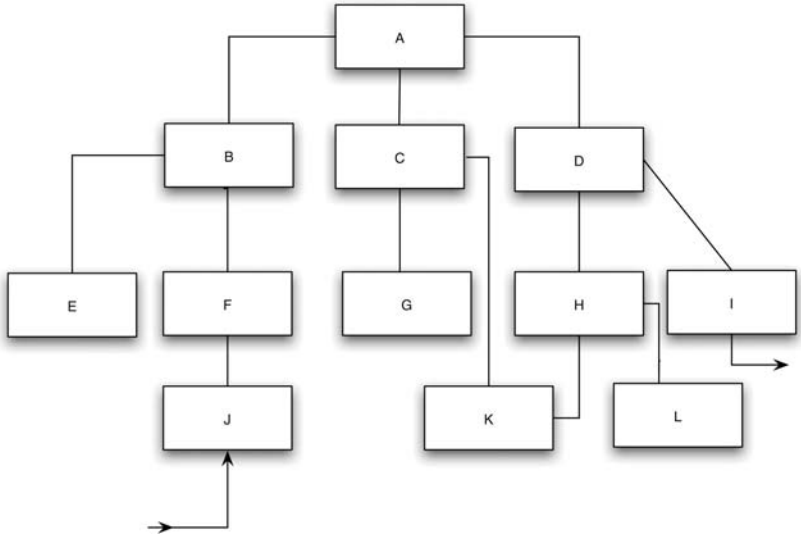

FIGURE 5.8 Sample 12-Module Program.

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

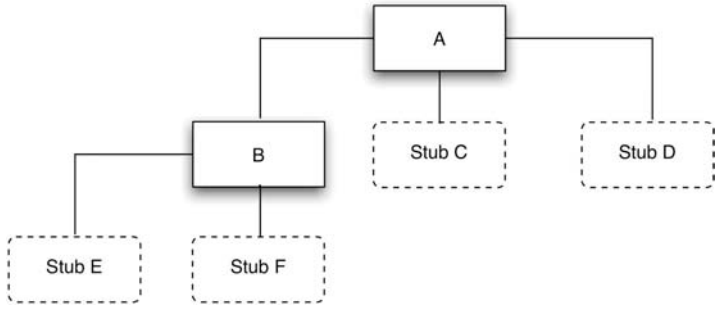

FIGURE 5.9 Second Step in the Top-Down Test.

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.

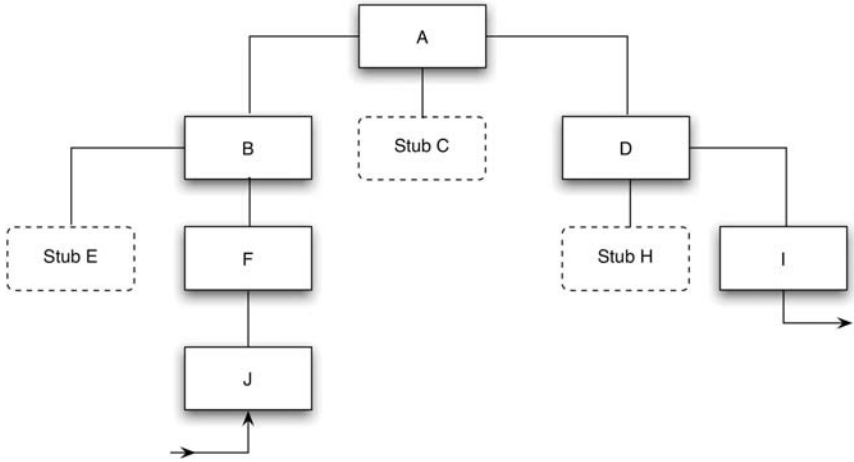

FIGURE 5.10 Intermediate State in the Top-Down Test.

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

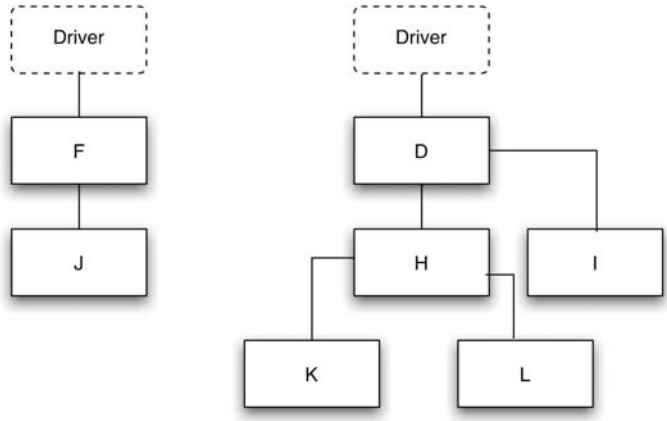

FIGURE 5.11 Intermediate State in the Bottom-Up Test.

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.