#### CHAPTER

# 17

# SOFTWARE TESTING TECHNIQUES

#### KEY CONCEPTS

CONCEPTS
basis path testing445
behavioral
<b>testing</b>
black-box
<b>testing</b>
<b>BVA</b> 465
control structure
<b>testing</b>
cyclomatic
complexity 446
equivalence
partitioning 463
flow graphs 445
loop testing 458
<b>OA testing</b> 466
testability 440
testing
objectives 439

he importance of software testing and its implications with respect to software quality cannot be overemphasized. To quote Deutsch [DEU79],

The development of software systems involves a series of production activities where opportunities for injection of human fallibilities are enormous. Errors may begin to occur at the very inception of the process where the objectives . . . may be erroneously or imperfectly specified, as well as [in] later design and development stages . . . Because of human inability to perform and communicate with perfection, software development is accompanied by a quality assurance activity.

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation.

The increasing visibility of software as a system element and the attendant "costs" associated with a software failure are motivating forces for well-planned, thorough testing. It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing. In the extreme, testing of human-rated software (e.g., flight control, nuclear reactor monitoring) can cost three to five times as much as all other software engineering steps combined!

#### QUICK LOOK

What is it? Once source code has been generated, software must be tested to uncover (and correct)

as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic of software components, and (2) exercise the input and output domains of the program to uncover errors in program function, behavior, and performance.

Who does it? During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved.

Why is it important? Reviews and other SQA activities can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps? Software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test case design tech-

## QUICK LOOK

niques. Software requirements are exercised using "black box" test case design techniques. In

both cases, the intent is to find the maximum number of errors with the minimum amount of effort and time.

What is the work product? A set of test cases designed to exercise both internal logic and exter-

nal requirements is designed and documented, expected results are defined, and actual results are recorded.

How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness.

In this chapter, we discuss software testing fundamentals and techniques for software test case design. Software testing fundamentals define the overriding objectives for software testing. Test case design focuses on a set of techniques for the creation of test cases that meet overall testing objectives. In Chapter 18, testing strategies and software debugging are presented.

#### 17.1 SOFTWARE TESTING FUNDAMENTALS



"A working program remains an elusive thing of beauty."

Robert Dunn

Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. The engineer creates a series of test cases that are intended to "demolish" the software that has been built. In fact, testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.

Software engineers are by their nature constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and overcome a conflict of interest that occurs when errors are uncovered. Beizer [BEI90] describes this situation effectively when he states:

There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if only everyone used structured programming, top-down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test case design is an admission of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt for what? For failing to achieve inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For failing to be telepathic? For not solving human communications problems that have been kicked around . . . for forty centuries?

Should testing instill guilt? Is testing really destructive? The answer to these questions is "No!" However, the objectives of testing are somewhat different than we might expect.

#### 17.1.1 Testing Objectives

In an excellent book on software testing, Glen Myers [MYE79] states a number of rules that can serve well as testing objectives:

- Testing is a process of executing a program with the intent of finding an error.
- **2.** A good test case is one that has a high probability of finding an as-yet-undiscovered error.
- **3.** A successful test is one that uncovers an as-yet-undiscovered error.

These objectives imply a dramatic change in viewpoint. They move counter to the commonly held view that a successful test is one in which no errors are found. Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

If testing is conducted successfully (according to the objectives stated previously), it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met. In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present. It is important to keep this (rather gloomy) statement in mind as testing is being conducted.

#### 17.1.2 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis [DAV95] suggests a set<sup>1</sup> of testing principles that have been adapted for use in this book:

- All tests should be traceable to customer requirements. As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- Tests should be planned long before testing begins. Test planning (Chapter 18) can begin as soon as the requirements model is complete.
   Detailed definition of test cases can begin as soon as the design model has

What are primary objectives when we test software?



'Errors are more common, more pervasive, and more troublesome in software than with other technologies."

**David Parnas** 

Only a small subset of Davis's testing principles are noted here. For more information, see [DAV95].

been solidified. Therefore, all tests can be planned and designed before any code has been generated.

- The Pareto principle applies to software testing. Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
- Testing should begin "in the small" and progress toward testing "in the large." The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system (Chapter 18).
- Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large (see Section 17.2 for further discussion). For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
- To be most effective, testing should be conducted by an independent third party. By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing). For reasons that have been introduced earlier in this chapter and are considered in more detail in Chapter 18, the software engineer who created the system is not the best person to conduct all tests for the software.

#### 17.1.3 Testability

In ideal circumstances, a software engineer designs a computer program, a system, or a product with "testability" in mind. This enables the individuals charged with testing to design effective test cases more easily. But what is *testability?* James Bach² describes testability in the following manner.

Software testability is simply how easily [a computer program] can be tested. Since testing is so profoundly difficult, it pays to know what can be done to streamline it. Sometimes programmers are willing to do things that will help the testing process and a checklist of possible design points, features, etc., can be useful in negotiating with them.

There are certainly metrics that could be used to measure testability in most of its aspects. Sometimes, testability is used to mean how adequately a particular set of



A useful paper entitled "Improving Software Testability" can be found at

www.stlabs.com/ newsletters/testnet /docs/testability. htm

<sup>2</sup> The paragraphs that follow are copyright 1994 by James Bach and have been adapted from an Internet posting that first appeared in the newsgroup comp.software-eng. This material is used with permission.

tests will cover the product. It's also used by the military to mean how easily a tool can be checked and repaired in the field. Those two meanings are not the same as *software testability*. The checklist that follows provides a set of characteristics that lead to testable software.

Operability. "The better it works, the more efficiently it can be tested."

- The system has few bugs (bugs add analysis and reporting overhead to the test process).
- No bugs block the execution of tests.
- The product evolves in functional stages (allows simultaneous development and testing).

Observability. "What you see is what you test."

- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected through self-testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

**Controllability.** "The better we can control the software, the more the testing can be automated and optimized."

- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.

**Decomposability.** "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."

- The software system is built from independent modules.
- Software modules can be tested independently.

**Simplicity.** "The less there is to test, the more quickly we can test it."

• Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements).



"Testability" occurs as a result of good design. Data design, architecture, interfaces, and component-level detail can either facilitate testing or make it difficult.

- Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).
- Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability. "The fewer the changes, the fewer the disruptions to testing."

- Changes to the software are infrequent.
- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.
- The software recovers well from failures.

**Understandability.** "The more information we have, the smarter we will test."

- The design is well understood.
- Dependencies between internal, external, and shared components are well understood.
- · Changes to the design are communicated.
- · Technical documentation is instantly accessible.
- · Technical documentation is well organized.
- Technical documentation is specific and detailed.
- Technical documentation is accurate.

The attributes suggested by Bach can be used by a software engineer to develop a software configuration (i.e., programs, data, and documents) that is amenable to testing.

And what about the tests themselves? Kaner, Falk, and Nguyen [KAN93] suggest the following attributes of a "good" test:

- What are the attributes of a "good" test?
- 1. A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.
- **2.** A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different). For example, a module of the *SafeHome* software (discussed in earlier chapters) is designed to recognize a user password to activate and deactivate the system. In an effort to uncover an error in password input, the tester designs a series of tests that input a sequence of passwords. Valid and invalid passwords (four numeral sequences) are input as separate tests. However, each

valid/invalid password should probe a different mode of failure. For example, the invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, say 1235, would have the same purpose as 1234 and is therefore redundant. However, the invalid input 8081 or 8180 has a subtle difference, attempting to demonstrate that an error exists for passwords "close to" but not identical with the valid password.

- **3.** A good test should be "best of breed" [KAN93]. In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.
- **4.** A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

#### 17.2 TEST CASE DESIGN

Quote:

'There is only one rule in designing test cases: cover all features, but do not make too many test cases."

Tsuneo Yamaura



The *Testing Techniques Newsletter* is an excellent source of information on testing methods:

www.testworks. com/News/TTN-Online/ The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. Yet, for reasons that we have already discussed, software engineers often treat testing as an afterthought, developing test cases that may "feel right" but have little assurance of being complete. Recalling the objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort.

A rich variety of test case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any engineered product (and most other things) can be tested in one of two ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function; (2) knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been adequately exercised. The first test approach is called black-box testing and the second, white-box testing.

When computer software is considered, *black-box testing* alludes to tests that are conducted at the software interface. Although they are designed to uncover errors, black-box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of



White-box tests can be designed only after a component-level design (or source code) exists. The logical details of the program must be available.



It is not possible to exhaustively test every program path because the number of paths is simply too large. external information (e.g., a database) is maintained. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. For example, consider the 100 line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately  $10^{14}$  possible paths that may be executed in this program!

To put this number in perspective, we assume that a magic test processor ("magic" because no such processor exists) has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond. Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. This would, undeniably, cause havoc in most development schedules. Exhaustive testing is impossible for large software systems.

White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity. The attributes of both black- and white-box testing can be combined to provide an approach that validates the software interface and selectively ensures that the internal workings of the software are correct.

#### 17.3 WHITE-BOX TESTING

White-box testing, sometimes called *glass-box testing*, is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity.

A reasonable question might be posed at this juncture: "Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort

ensuring that program requirements have been met?" Stated another way, why don't we spend all of our energy on black-box tests? The answer lies in the nature of software defects (e.g., [JON81]):

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement function, conditions, or control that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. The logical flow of a program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.
- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur.
   Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.

Each of these reasons provides an argument for conducting white-box tests. Black-box testing, no matter how thorough, may miss the kinds of errors noted here. White-box testing is far more likely to uncover them.

#### 17.4 BASIS PATH TESTING

Basis path testing is a white-box testing technique first proposed by Tom McCabe [MCC76]. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

#### 17.4.1 Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced.<sup>3</sup> The flow graph depicts logical control flow using the notation illustrated in Figure 17.1. Each structured construct (Chapter 16) has a corresponding flow graph symbol. To illustrate the use of a flow graph, we consider the procedural design represen-

tation in Figure 17.2A. Here, a flowchart is used to depict program control structure.

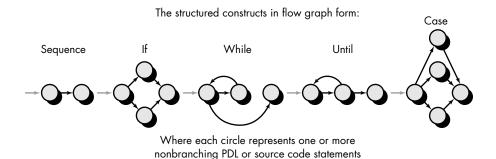
# Quote:

Bugs lurk in corners and congregate at boundaries."

**Boris Beizer** 

<sup>3</sup> In actuality, the basis path method can be conducted without the use of flow graphs. However, they serve as a useful tool for understanding control flow and illustrating the approach.

FIGURE 17.1
Flow graph
notation





Draw a flow graph when the logical control structure of a module is complex. The flow graph enables you to trace program paths more readily. Figure 17.2B maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 17.2B, each circle, called a *flow graph node*, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links*, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions*. When counting regions, we include the area outside the graph as a region.<sup>4</sup>

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 17.3, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions a and b in the statement IF a OR b. Each node that contains a condition is called a *predicate node* and is characterized by two or more edges emanating from it.

## 17.4.2 Cyclomatic Complexity

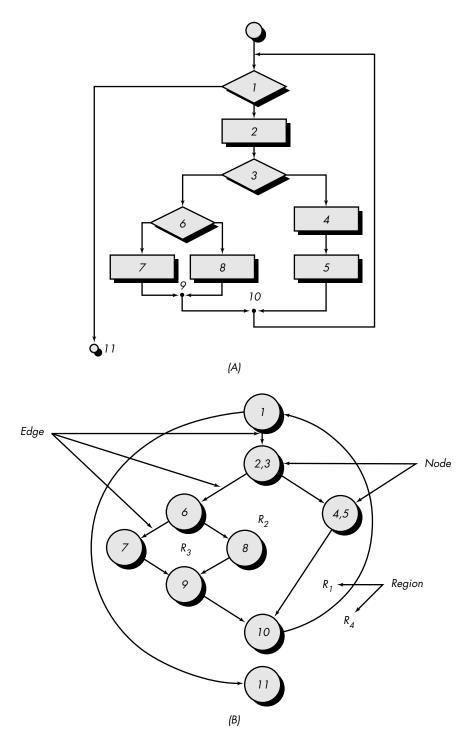
*Cyclomatic complexity* is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow

<sup>4</sup> A more-detailed discussion of graphs and their use in testing is contained in Section 17.6.1.

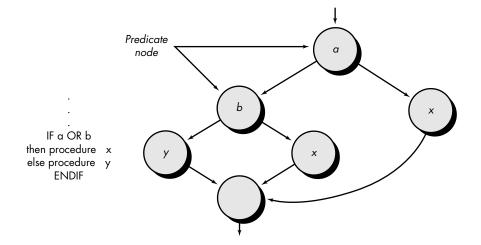
FIGURE 17.2
Flowchart, (A) and flow

graph (B)



#### FIGURE 17.3

Compound logic



graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 17.2B is

path 1: 1-11

path 2: 1-2-3-4-5-10-1-11

path 3: 1-2-3-6-8-9-10-1-11

path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a *basis set* for the flow graph in Figure 17.2B. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways:

- 1. The number of regions of the flow graph correspond to the cyclomatic complexity.
- **2.** Cyclomatic complexity, V(G), for a flow graph, G, is defined as

$$V(G) = E - N + 2$$



Cyclomatic complexity is a useful metric for predicting those modules that are likely to be error prone. It can be used for test planning as well as test case design.





Cyclomatic complexity provides the upper bound on the number of test cases that must be executed to guarantee that every statement in a component has been executed at least once.

where E is the number of flow graph edges, N is the number of flow graph nodes.

**3.** Cyclomatic complexity, V(G), for a flow graph, G, is also defined as

$$V(G) = P + 1$$

where *P* is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph in Figure 17.2B, the cyclomatic complexity can be computed using each of the algorithms just noted:

- 1. The flow graph has four regions.
- **2.** V(G) = 11 edges 9 nodes + 2 = 4.
- **3.** V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 17.2B is 4.

More important, the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

#### 17.4.3 Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure *average*, depicted in PDL in Figure 17.4, will be used as an example to illustrate each step in the test case design method. Note that *average*, although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

- 1. Using the design or code as a foundation, draw a corresponding flow graph. A flow graph is created using the symbols and construction rules presented in Section 16.4.1. Referring to the PDL for average in Figure 17.4, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 17.5.
- **2. Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity, *V*(*G*), is determined by applying the algorithms described in Section 17.5.2. It should be noted that *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average*, compound conditions count as two) and adding 1. Referring to Figure 17.5,

$$V(G) = 6$$
 regions

$$V(G) = 17 \text{ edges} - 13 \text{ nodes} + 2 = 6$$

$$V(G) = 5$$
 predicate nodes + 1 = 6



'To err is human, to find a bug, devine."

Robert Dunn

#### FIGURE 17.4

PDL for test case design with nodes identified

#### PROCEDURE average;

END average

\* This procedure computes the average of 100 or fewer numbers that lie between bounding values; it also computes the sum and the total number valid. INTERFACE RETURNS average, total.input, total.valid; INTERFACE ACCEPTS value, minimum, maximum; TYPE value[1:100] IS SCALAR ARRAY; TYPE average, total.input, total.valid; minimum, maximum, sum IS SCALAR; TYPE i IS INTEGER; /i = 1: total.input = total.valid = 0;sum = 0: DO WHILE value[i] <> -999 AND total.input < 100 3 4 increment total.input by 1; IF value[i] > = minimum AND value[i] < = maximum 6 THEN increment total.valid by 1; sum = s sum + value[i] ELSE skip ENDIF ₹ increment i by 1; 9 ENDDO IF total.valid > 0 10 11 THEN average = sum / total.valid; → ELSE average = -999; 13 ENDIF

**3. Determine a basis set of linearly independent paths.** The value of V(G) provides the number of linearly independent paths through the program control structure. In the case of procedure *average*, we expect to specify six paths:

```
path 1: 1-2-10-11-13

path 2: 1-2-10-12-13

path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-...

path 5: 1-2-3-4-5-6-8-9-2-...

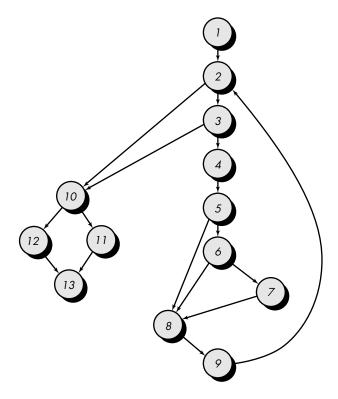
path 6: 1-2-3-4-5-6-7-8-9-2-...
```

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4. Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Test cases that satisfy the basis set just described are

#### FIGURE 17.5

Flow graph for the procedure **average** 



#### Path 1 test case:

value(k) = valid input, where k < i for  $2 \le i \le 100$ 

value(*i*) = −999 where  $2 \le i \le 100$ 

Expected results: Correct average based on k values and proper totals.

*Note:* Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

#### Path 2 test case:

value(1) = -999

Expected results: Average = -999; other totals at initial values.

#### Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

#### Path 4 test case:

value(i) = valid input where i < 100

value(k) < minimum where k < i

*Expected results:* Correct average based on *k* values and proper totals.

# Quote:

'[Software engineers] considerably underestimate the number of tests required to verify a straightforward program."

Martyn Ould and Charles Unwin

#### Path 5 test case:

value(i) = valid input where i < 100

value(k) > maximum where k <= i

Expected results: Correct average based on *n* values and proper totals.

#### Path 6 test case:

value(i) = valid input where i < 100

*Expected results:* Correct average based on *n* values and proper totals.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

It is important to note that some independent paths (e.g., path 1 in our example) cannot be tested in stand-alone fashion. That is, the combination of data required to traverse the path cannot be achieved in the normal flow of the program. In such cases, these paths are tested as part of another path test.

#### 17.4.4 Graph Matrices

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a *graph matrix*, can be quite useful.

A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. A simple example of a flow graph and its corresponding graph matrix [BEI90] is shown in Figure 17.6.

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b.

To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be executed.
- The processing time expended during traversal of a link.
- The memory required during traversal of a link.
- The resources required during traversal of a link.

What is a graph matrix and how do we extend it for use in testing?

**FIGURE 17.6** Graph matrix

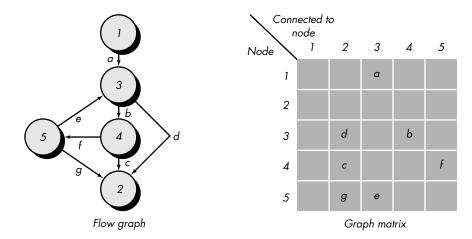
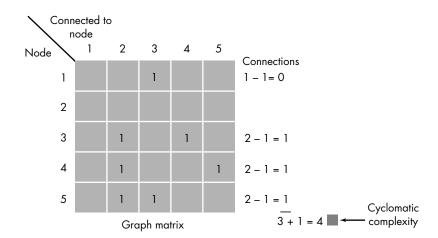


FIGURE 17.7
Connection
matrix



To illustrate, we use the simplest weighting to indicate connections (0 or 1). The graph matrix in Figure 17.6 is redrawn as shown in Figure 17.7. Each letter has been replaced with a 1, indicating that a connection exists (zeros have been excluded for clarity). Represented in this form, the graph matrix is called a *connection matrix*.

Referring to Figure 17.7, each row with two or more entries represents a predicate node. Therefore, performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining cyclomatic complexity (Section 17.4.2).

Beizer [BEI90] provides a thorough treatment of additional mathematical algorithms that can be applied to graph matrices. Using these techniques, the analysis required to design test cases can be partially or fully automated.

#### 17.5 CONTROL STRUCTURE TESTING

The basis path testing technique described in Section 17.4 is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.



Errors are much more common in the neighborhood of logical conditions than they are in the locus of sequential processing statements.

#### 17.5.1 Condition Testing<sup>5</sup>

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT  $(\neg)$  operator. A relational expression takes the form

 $E_1$  <relational-operator>  $E_2$ 

where  $E_1$  and  $E_2$  are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥. A *compound condition* is composed of two or more simple conditions, Boolean operators, and parentheses. We assume that Boolean operators allowed in a compound condition include OR (|), AND (&) and NOT ( $\neg$ ). A condition without relational expressions is referred to as a *Boolean expression*.

Therefore, the possible types of elements in a condition include a Boolean operator, a Boolean variable, a pair of Boolean parentheses (surrounding a simple or compound condition), a relational operator, or an arithmetic expression.

If a condition is incorrect, then at least one component of the condition is incorrect. Therefore, types of errors in a condition include the following:

- Boolean operator error (incorrect/missing/extra Boolean operators).
- Boolean variable error.
- Boolean parenthesis error.
- Relational operator error.
- Arithmetic expression error.

The condition testing method focuses on testing each condition in the program. Condition testing strategies (discussed later in this section) generally have two advantages. First, measurement of test coverage of a condition is simple. Second, the test coverage of conditions in a program provides guidance for the generation of additional tests for the program.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program. If a test set for a program *P* is effective

<sup>5</sup> Section 17.5.1 and 17.5.2 have been adapted from [TAI89] with permission of Professor K. C. Tai.

for detecting errors in the conditions contained in *P*, it is likely that this test set is also effective for detecting other errors in *P*. In addition, if a testing strategy is effective for detecting errors in a condition, then it is likely that this strategy will also be effective for detecting errors in a program.

A number of condition testing strategies have been proposed. *Branch testing* is probably the simplest condition testing strategy. For a compound condition *C*, the true and false branches of *C* and every simple condition in *C* need to be executed at least once [MYE79].

Domain testing [WHI80] requires three or four tests to be derived for a relational expression. For a relational expression of the form

 $E_1$  <relational-operator>  $E_2$ 

three tests are required to make the value of  $E_1$  greater than, equal to, or less than that of  $E_2$  [HOW82]. If <relational-operator> is incorrect and  $E_1$  and  $E_2$  are correct, then these three tests guarantee the detection of the relational operator error. To detect errors in  $E_1$  and  $E_2$ , a test that makes the value of  $E_1$  greater or less than that of  $E_2$  should make the difference between these two values as small as possible.

For a Boolean expression with n variables, all of  $2^n$  possible tests are required (n > 0). This strategy can detect Boolean operator, variable, and parenthesis errors, but it is practical only if n is small.

Error-sensitive tests for Boolean expressions can also be derived [FOS84, TAI87]. For a singular Boolean expression (a Boolean expression in which each Boolean variable occurs only once) with n Boolean variables (n > 0), we can easily generate a test set with less than  $2^n$  tests such that this test set guarantees the detection of multiple Boolean operator errors and is also effective for detecting other errors.

Tai [TAI89] suggests a condition testing strategy that builds on the techniques just outlined. Called *BRO* (branch and relational operator) testing, the technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variables.

The BRO strategy uses condition constraints for a condition C. A condition constraint for C with n simple conditions is defined as  $(D_1, D_2, \ldots, D_n)$ , where  $D_i$   $(0 < i \le n)$  is a symbol specifying a constraint on the outcome of the ith simple condition in condition C. A condition constraint D for condition C is said to be covered by an execution of C if, during this execution of C, the outcome of each simple condition in C satisfies the corresponding constraint in D.

For a Boolean variable, B, we specify a constraint on the outcome of B that states that B must be either true (t) or false (f). Similarly, for a relational expression, the symbols >, =, < are used to specify constraints on the outcome of the expression.



Even if you decide against condition testing, you should spend time evaluating each condition in an effort to uncover errors. This is a primary hiding place for bugs!

As an example, consider the condition

$$C_1$$
:  $B_1 \& B_2$ 

where  $B_I$  and  $B_2$  are Boolean variables. The condition constraint for  $C_I$  is of the form  $(D_I, D_2)$ , where each of  $D_I$  and  $D_2$  is t or f. The value (t, f) is a condition constraint for  $C_I$  and is covered by the test that makes the value of  $B_I$  to be true and the value of  $B_2$  to be false. The BRO testing strategy requires that the constraint set  $\{(t, t), (f, t), (t, f)\}$  be covered by the executions of  $C_I$ . If  $C_I$  is incorrect due to one or more Boolean operator errors, at least one of the constraint set will force  $C_I$  to fail.

As a second example, a condition of the form

$$C_2$$
:  $B_1 & (E_3 = E_4)$ 

where  $B_I$  is a Boolean expression and  $E_3$  and  $E_4$  are arithmetic expressions. A condition constraint for  $C_2$  is of the form  $(D_I, D_2)$ , where each of  $D_I$  is t or f and  $D_2$  is >, =, <. Since  $C_2$  is the same as  $C_I$  except that the second simple condition in  $C_2$  is a relational expression, we can construct a constraint set for  $C_2$  by modifying the constraint set  $\{(t, t), (f, t), (t, f)\}$  defined for  $C_I$ . Note that t for  $(E_3 = E_4)$  implies = and that f for  $(E_3 = E_4)$  implies either < or >. By replacing (t, t) and (f, t) with (t, =) and (f, =), respectively, and by replacing (t, f) with (t, <) and (t, >), the resulting constraint set for  $C_2$  is  $\{(t, =), (f, =), (t, <), (t, >)\}$ . Coverage of the preceding constraint set will guarantee detection of Boolean and relational operator errors in  $C_2$ .

As a third example, we consider a condition of the form

$$C_3$$
:  $(E_1 > E_2) & (E_3 = E_4)$ 

where  $E_1$ ,  $E_2$ ,  $E_3$ , and  $E_4$  are arithmetic expressions. A condition constraint for  $C_3$  is of the form  $(D_1, D_2)$ , where each of  $D_1$  and  $D_2$  is >, =, <. Since  $C_3$  is the same as  $C_2$  except that the first simple condition in  $C_3$  is a relational expression, we can construct a constraint set for  $C_3$  by modifying the constraint set for  $C_2$ , obtaining

$$\{(>, =), (=, =), (<, =), (>, >), (>, <)\}$$

Coverage of this constraint set will guarantee detection of relational operator errors in  $C_3$ .

#### 17.5.2 Data Flow Testing

The *data flow testing* method selects test paths of a program according to the locations of definitions and uses of variables in the program. A number of data flow testing strategies have been studied and compared (e.g., [FRA88], [NTA88], [FRA93]).

To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number,

```
DEF(S) = {X \mid statement S contains a definition of X}
USE(S) = {X \mid statement S contains a use of X}
```

If statement *S* is an *if* or *loop* statement, its DEF set is empty and its USE set is based on the condition of statement *S*. The definition of variable *X* at statement *S* is said to be *live* at statement *S'* if there exists a path from statement *S* to statement *S'* that contains no other definition of *X*.

A *definition-use* (DU) *chain* of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.

One simple data flow testing strategy is to require that every DU chain be covered at least once. We refer to this strategy as the *DU testing strategy*. It has been shown that DU testing does not guarantee the coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare situations such as if-then-else constructs in which the *then part* has no definition of any variable and the *else part* does not exist. In this situation, the else branch of the *if* statement is not necessarily covered by DU testing.

Data flow testing strategies are useful for selecting test paths of a program containing nested *if* and *loop* statements. To illustrate this, consider the application of DU testing to select test paths for the PDL that follows:

```
proc x
   B1;
   do while C1
       if C2
          then
              if C4
                 then B4:
                 else B5;
              endif:
          else
              if C3
                 then B2;
                 else B3:
              endif:
          endif:
       enddo:
       B6;
   end proc;
```

To apply the DU testing strategy to select test paths of the control flow diagram, we need to know the definitions and uses of variables in each condition or block in the PDL. Assume that variable X is defined in the last statement of blocks B1, B2, B3, B4, and B5 and is used in the first statement of blocks B2, B3, B4, B5, and B6. The DU testing strategy requires an execution of the shortest path from each of  $B_i$ ,  $0 < i \le 5$ ,



It is unrealistic to assume that data flow testing will be used extensively when testing a large system. However, it can be used in a targeted fashion for areas of the software that are suspect.

to each of  $B_{j'}$   $1 < j \le 6$ . (Such testing also covers any use of variable X in conditions C1, C2, C3, and C4.) Although there are 25 DU chains of variable X, we need only five paths to cover these DU chains. The reason is that five paths are needed to cover the DU chain of X from  $B_{j'}$   $0 < i \le 5$ , to B6 and other DU chains can be covered by making these five paths contain iterations of the loop.

If we apply the branch testing strategy to select test paths of the PDL just noted, we do not need any additional information. To select paths of the diagram for BRO testing, we need to know the structure of each condition or block. (After the selection of a path of a program, we need to determine whether the path is feasible for the program; that is, whether at least one input exists that exercises the path.)

Since the statements in a program are related to each other according to the definitions and uses of variables, the data flow testing approach is effective for error detection. However, the problems of measuring test coverage and selecting test paths for data flow testing are more difficult than the corresponding problems for condition testing.

#### 17.5.3 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests.

*Loop testing* is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [BEI90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 17.8).

**Simple loops.** The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.

- 1. Skip the loop entirely.
- 2. Only one pass through the loop.
- **3.** Two passes through the loop.
- **4.** m passes through the loop where m < n.
- **5.** n-1, n, n+1 passes through the loop.

**Nested loops.** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [BEI90] suggests an approach that will help to reduce the number of tests:

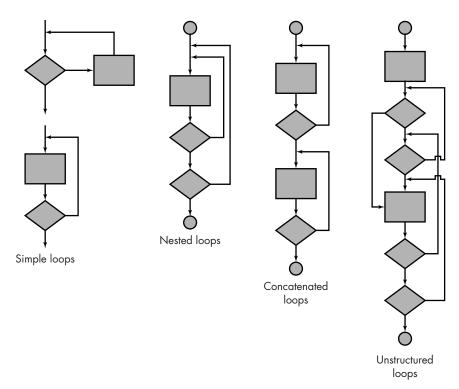
- 1. Start at the innermost loop. Set all other loops to minimum values.
- **2.** Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.



Complex loop structures are another hiding place for bugs. It's well worth spending time designing tests that fully exercise loop structures.

FIGURE 17.8 Classes of

loops



- **3.** Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
- **4.** Continue until all loops have been tested.

**Concatenated loops.** Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

**Unstructured loops.** Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs (Chapter 16).



You can't test unstructured loops effectively. Redesign them.

#### 17.6 BLACK-BOX TESTING

*Black-box testing,* also called *behavioral testing,* focuses on the functional requirements of the software. That is, black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing (see Chapter 18). Because black-box testing purposely disregards control structure, attention is focused on the information domain. Tests are designed to answer the following questions:

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

By applying black-box techniques, we derive a set of test cases that satisfy the following criteria [MYE79]: (1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing and (2) test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

#### 17.6.1 Graph-Based Testing Methods

The first step in black-box testing is to understand the objects<sup>6</sup> that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another [BEI95]." Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

To accomplish these steps, the software engineer begins by creating a *graph*—a collection of *nodes* that represent objects; *links* that represent the relationships between objects; *node weights* that describe the properties of a node (e.g., a specific data value or state behavior); and *link weights* that describe some characteristic of a link.<sup>7</sup>

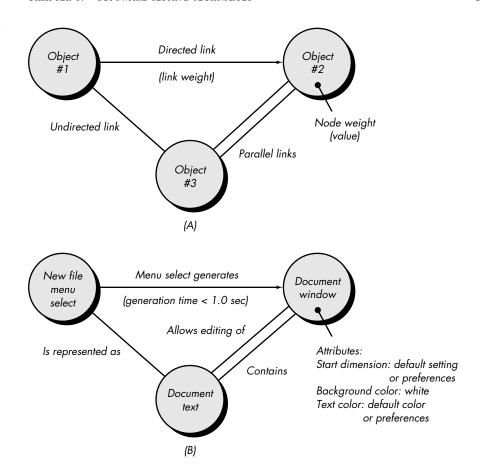


A graph represents the relationships between data objects and program objects, enabling us to derive test cases that search for errors associated with these relationships.

<sup>6</sup> In this context, the term *object* encompasses the data objects that we discussed in Chapters 11 and 12 as well as program objects such as modules or collections of programming language statements.

<sup>7</sup> If these concepts seem vaguely familiar, recall that graphs were also used in Section 17.4.1 to create a program graph for the basis path testing method. The nodes of the program graph contained instructions (program objects) characterized as either procedural design representations or source code, and the directed links indicated the control flow between these program objects. Here, the use of graphs is extended to encompass black-box testing as well.

## (A) Graph notation (B) Simple example



The symbolic representation of a graph is shown in Figure 17.9A. Nodes are represented as circles connected by links that take a number of different forms. A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction. A *bidirectional link*, also called a *symmetric link*, implies that the relationship applies in both directions. *Parallel links* are used when a number of different relationships are established between graph nodes.

As a simple example, consider a portion of a graph for a word-processing application (Figure 17.9B) where

Object #1 = new file menu select Object #2 = document window Object #3 = document text

Referring to the figure, a menu select on **new file** generates a **document window**. The node weight of **document window** provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a

symmetric relationship between the **new file menu select** and **document text,** and parallel links indicate relationships between **document window** and **document text.** In reality, a far more detailed graph would have to be generated as a precursor to test case design. The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

Beizer [BEI95] describes a number of behavioral testing methods that can make use of graphs:

**Transaction flow modeling.** The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps (e.g., **flight.information.input** is followed by *validation/availability.processing*). The data flow diagram (Chapter 12) can be used to assist in creating graphs of this type.

**Finite state modeling.** The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **order-information** is verified during *inventory-availability look-up* and is followed by **customer-billing-information input**). The state transition diagram (Chapter 12) can be used to assist in creating graphs of this type.

**Data flow modeling.** The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node **FICA.tax.withheld (FTW)** is computed from **gross.wages (GW)** using the relationship, FTW =  $0.62 \times GW$ .

**Timing modeling.** The nodes are program objects and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

A detailed discussion of each of these graph-based testing methods is beyond the scope of this book. The interested reader should see [BEI95] for a comprehensive discussion. It is worthwhile, however, to provide a generic outline of the graph-based testing approach.

Graph-based testing begins with the definition of all nodes and node weights. That is, objects and attributes are identified. The data model (Chapter 12) can be used as a starting point, but it is important to note that many nodes may be program objects (not explicitly represented in the data model). To provide an indication of the start and stop points for the graph, it is useful to define entry and exit nodes.

Once nodes have been identified, links and link weights should be established. In general, links should be named, although links that represent control flow between program objects need not be named.

What generic activities are required during graph-based testing?

In many cases, the graph model may have loops (i.e., a path through the graph in which one or more nodes is encountered more than one time). Loop testing (Section 17.5.3) can also be applied at the behavioral (black-box) level. The graph will assist in identifying those loops that need to be tested.

Each relationship is studied separately so that test cases can be derived. The *transitivity* of sequential relationships is studied to determine how the impact of relationships propagates across objects defined in a graph. Transitivity can be illustrated by considering three objects, **X**, **Y**, and **Z**. Consider the following relationships:

X is required to compute Y

**Y** is required to compute **Z** 

Therefore, a transitive relationship has been established between **X** and **Z**:

X is required to compute Z

Based on this transitive relationship, tests to find errors in the calculation of  ${\bf Z}$  must consider a variety of values for both  ${\bf X}$  and  ${\bf Y}$ .

The *symmetry* of a relationship (graph link) is also an important guide to the design of test cases. If a link is indeed bidirectional (symmetric), it is important to test this feature. The UNDO feature [BEI95] in many personal computer applications implements limited symmetry. That is, UNDO allows an action to be negated after it has been completed. This should be thoroughly tested and all exceptions (i.e., places where UNDO cannot be used) should be noted. Finally, every node in the graph should have a relationship that leads back to itself; in essence, a "no action" or "null action" loop. These *reflexive* relationships should also be tested.

As test case design begins, the first objective is to achieve *node coverage*. By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.

Next, *link coverage* is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present. A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked (Section 17.5.3).

#### 17.6.2 Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.



Input classes are known relatively early in the software process. For this reason, begin thinking about equivalence partitioning as the design is created. Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, if a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an equivalence class is present [BEI95]. An *equivalence class* represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

- **1.** If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined.
- **2.** If an input condition requires a specific *value*, one valid and two invalid equivalence classes are defined.
- **3.** If an input condition specifies a member of a *set*, one valid and one invalid equivalence class are defined.
- **4.** If an input condition is *Boolean*, one valid and one invalid class are defined.

As an example, consider data maintained as part of an automated banking application. The user can access the bank using a personal computer, provide a six-digit password, and follow with a series of typed commands that trigger various banking functions. During the log-on sequence, the software supplied for the banking application accepts data in the form

```
area code—blank or three-digit number prefix—three-digit number not beginning with 0 or 1 suffix—four-digit number password—six digit alphanumeric string commands—check, deposit, bill pay, and the like
```

The input conditions associated with each data element for the banking application can be specified as

area code: Input condition, Boolean—the area code may or may not be

present.

Input condition, range—values defined between 200 and 999, with

specific exceptions.

prefix: Input condition, range—specified value >200

Input condition, value—four-digit length

password: Input condition, *Boolean*—a password may or may not be present.

Input condition, *value*—six-character string.

command: Input condition, set—containing commands noted previously.

Applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

# 17.6.3 Boundary Value Analysis



BVA extends equivalence partitioning by focusing on data at the "edges" of an equivalence class. For reasons that are not completely clear, a greater number of errors tends to occur at the boundaries of the input domain rather than in the "center." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique. Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well [MYE79].

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

- **1.** If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*
- 2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
- **3.** Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.
- **4.** If internal program data structures have prescribed boundaries (e.g., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection.

#### 16.6.4 Comparison Testing

There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical. In such applications redundant hardware and software are often used to minimize the possibility of error. When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification. In such situations, each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.



Using lessons learned from redundant systems, researchers (e.g., [BRI87]) have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer-based system. These independent versions form the basis of a black-box testing technique called *comparison testing* or *back-to-back testing* [KNI89].

When multiple implementations of the same specification have been produced, test cases designed using other black-box techniques (e.g., equivalence partitioning) are provided as input to each version of the software. If the output from each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference. In most cases, the comparison of outputs can be performed by an automated tool.

Comparison testing is not foolproof. If the specification from which all versions have been developed is in error, all versions will likely reflect the error. In addition, if each of the independent versions produces identical but incorrect results, condition testing will fail to detect the error.

#### 17.6.5 Orthogonal Array Testing

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test processing of the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing become impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding errors associated with region faults—an error category associated with faulty logic within a software component.

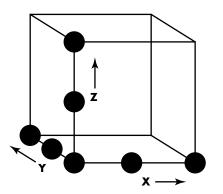
To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are  $3^3 = 27$  possible test cases. Phadke [PHA97] suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure 17.10. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

When orthogonal array testing occurs, an *L9 orthogonal array* of test cases is created. The *L9* orthogonal array has a "balancing property [PHA97]." That is, test cases (represented by black dots in the figure) are "dispersed uniformly throughout the test

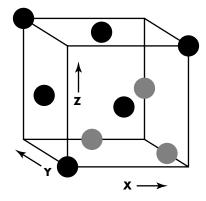


Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases.

A geometric view of test cases [PHA97]







L9 orthogonal array

domain," as illustrated in the right-hand cube in Figure 17.10. Test coverage across the input domain is more complete.

To illustrate the use of the L9 orthogonal array, consider the *send* function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now

P1 = 2, send it one hour later

P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other *send* functions. If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3). Phadke [PHA97] assesses these test cases in the following manner:

Such test cases are useful only when one is certain that these test parameters do not interact. They can detect logic faults where a single parameter value makes the software malfunction. These faults are called *single mode faults*. This method cannot detect logic faults that cause malfunction when two or more parameters simultaneously take certain values; that is, it cannot detect any interactions. Thus its ability to detect faults is limited.

Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is  $3^4 = 81$ , large, but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables us to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure 17.11.

FIGURE 17.11
An L9
orthogonal
array

Test case	Test parameters			
	$P_1$	$P_2$	$P_3$	$P_4$
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Phadke [PHA97] assesses the result of tests using the L9 orthogonal array in the following manner:

**Detect and isolate all single mode faults.** A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 [Figure 17.11] will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with "send it now" (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.

**Detect all double mode faults.** If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault*. Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.

**Multimode faults.** Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.

A detailed discussion of orthogonal array testing can be found in [PHA89].

# 17.7 TESTING FOR SPECIALIZED ENVIRONMENTS, ARCHITECTURES, AND APPLICATIONS

As computer software has become more complex, the need for specialized testing approaches has also grown. The white-box and black-box testing methods discussed in Sections 17.5 and 17.6 are applicable across all environments, architectures, and

applications, but unique guidelines and approaches to testing are sometimes warranted. In this section we consider testing guidelines for specialized environments, architectures, and applications that are commonly encountered by software engineers.

#### 17.7.1 Testing GUIs

**XRef**Guidelines for the design of GUIs are presented in Chapter



Testing GUIs

Graphical user interfaces (GUIs) present interesting challenges for software engineers. Because of reusable components provided as part of GUI development environments, the creation of the user interface has become less time consuming and more precise. But, at the same time, the complexity of GUIs has grown, leading to more difficulty in the design and execution of test cases.

Because many modern GUIs have the same look and feel, a series of standard tests can be derived. Finite state modeling graphs may be used to derive a series of tests that address specific data and program objects that are relevant to the GUI.

Due to the large number of permutations associated with GUI operations, testing should be approached using automated tools. A wide array of GUI testing tools has appeared on the market over the past few years. For further discussion, see Chapter 31.

#### 17.7.2 Testing of Client/Server Architectures

Client/server (C/S) architectures represent a significant challenge for software testers. The distributed nature of client/server environments, the performance issues associated with transaction processing, the potential presence of a number of different hardware platforms, the complexities of network communication, the need to service multiple clients from a centralized (or in some cases, distributed) database, and the coordination requirements imposed on the server all combine to make testing of C/S architectures and the software that reside within them considerably more difficult than stand-alone applications. In fact, recent industry studies indicate a significant increase in testing time and cost when C/S environments are developed.

#### 17.7.3 Testing Documentation and Help Facilities

The term *software testing* conjures images of large numbers of test cases prepared to exercise computer programs and the data that they manipulate. Recalling the definition of *software* presented in the first chapter of this book, it is important to note that testing must also extend to the third element of the software configuration—documentation.

Errors in documentation can be as devastating to the acceptance of the program as errors in data or source code. Nothing is more frustrating than following a user guide or an on-line help facility exactly and getting results or behaviors that do not coincide with those predicted by the documentation. It is for this reason that that documentation testing should be a meaningful part of every software test plan.

# XRef

Client/server software engineering is presented in Chapter 28.

Documentation testing can be approached in two phases. The first phase, *review and inspection* (Chapter 8), examines the document for editorial clarity. The second phase, *live test,* uses the documentation in conjunction with the use of the actual program.

Surprisingly, a live test for documentation can be approached using techniques that are analogous to many of the black-box testing methods discussed in Section 17.6. Graph-based testing can be used to describe the use of the program; equivalence partitioning and boundary value analysis can be used to define various classes of input and associated interactions. Program usage is then tracked through the documentation. The following questions should be answered during both phases:

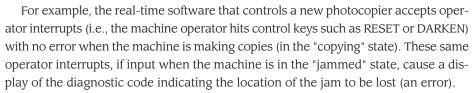
- What questions should be addressed as we test documentation?
- Does the documentation accurately describe how to accomplish each mode of use?
- Is the description of each interaction sequence accurate?
- Are examples accurate?
- Are terminology, menu descriptions, and system responses consistent with the actual program?
- Is it relatively easy to locate guidance within the documentation?
- Can troubleshooting be accomplished easily with the documentation?
- Are the document table of contents and index accurate and complete?
- Is the design of the document (layout, typefaces, indentation, graphics) conducive to understanding and quick assimilation of information?
- Are all software error messages displayed for the user described in more detail in the document? Are actions to be taken as a consequence of an error message clearly delineated?
- If hypertext links are used, are they accurate and complete?
- If hypertext is used, is the navigation design appropriate for the information required?

The only viable way to answer these questions is to have an independent third party (e.g., selected users) test the documentation in the context of program usage. All discrepancies are noted and areas of document ambiguity or weakness are defined for potential rewrite.

#### 17.7.4 Testing for Real-Time Systems

The time-dependent, asynchronous nature of many real-time applications adds a new and potentially difficult element to the testing mix—time. Not only does the test case designer have to consider white- and black-box test cases but also event handling (i.e., interrupt processing), the timing of the data, and the parallelism of the tasks (processes) that handle the data. In many situations, test data provided when a real-

time system is in one state will result in proper processing, while the same data provided when the system is in a different state may lead to error.



In addition, the intimate relationship that exists between real-time software and its hardware environment can also cause testing problems. Software tests must consider the impact of hardware faults on software processing. Such faults can be extremely difficult to simulate realistically.

Comprehensive test case design methods for real-time systems have yet to evolve. However, an overall four-step strategy can be proposed:

**Task testing.** The first step in the testing of real-time software is to test each task independently. That is, white-box and black-box tests are designed and executed for each task. Each task is executed independently during these tests. Task testing uncovers errors in logic and function but not timing or behavior

Behavioral testing. Using system models created with CASE tools, it is possible to simulate the behavior of a real-time system and examine its behavior as a consequence of external events. These analysis activities can serve as the basis for the design of test cases that are conducted when the real-time software has been built. Using a technique that is similar to equivalence partitioning (Section 17.6.1), events (e.g., interrupts, control signals) are categorized for testing. For example, events for the photocopier might be user interrupts (e.g., reset counter), mechanical interrupts (e.g., paper jammed), system interrupts (e.g., toner low), and failure modes (e.g., roller overheated). Each of these events is tested individually and the behavior of the executable system is examined to detect errors that occur as a consequence of processing associated with these events. The behavior of the system model (developed during the analysis activity) and the executable software can be compared for conformance. Once each class of events has been tested, events are presented to the system in random order and with random frequency. The behavior of the software is examined to detect behavior errors.

**Intertask testing.** Once errors in individual tasks and in system behavior have been isolated, testing shifts to time-related errors. Asynchronous tasks that are known to communicate with one another are tested with different data rates and processing load to determine if intertask synchronization errors will occur. In addition, tasks that communicate via a message queue or data store are tested to uncover errors in the sizing of these data storage areas.





The Software Testing
Discussion Forum
presents topics of interest
to testing professionals:
www.ondaweb.com
/HyperNews/
get.cgi/forums/
sti.html

**System testing.** Software and hardware are integrated and a full range of system tests (Chapter 18) are conducted in an attempt to uncover errors at the software/hardware interface. Most real-time systems process interrupts. Therefore, testing the handling of these Boolean events is essential. Using the state transition diagram and the control specification (Chapter 12), the tester develops a list of all possible interrupts and the processing that occurs as a consequence of the interrupts. Tests are then designed to assess the following system characteristics:

- Are interrupt priorities properly assigned and properly handled?
- · Is processing for each interrupt handled correctly?
- Does the performance (e.g., processing time) of each interrupt-handling procedure conform to requirements?
- Does a high volume of interrupts arriving at critical times create problems in function or performance?

In addition, global data areas that are used to transfer information as part of interrupt processing should be tested to assess the potential for the generation of side effects.

#### 17.8 SUMMARY

The primary objective for test case design is to derive a set of tests that have the highest likelihood for uncovering errors in the software. To accomplish this objective, two different categories of test case design techniques are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.

Hetzel [HET84] describes white-box testing as "testing in the small." His implication is that the white-box tests that we have considered in this chapter are typically applied to small program components (e.g., modules or small groups of modules). Black-box testing, on the other hand, broadens our focus and might be called "testing in the large."

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage. Equiv-

alence partitioning divides the input domain into classes of data that are likely to exercise specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems will small numbers of input parameters.

Specialized testing methods encompass a broad array of software capabilities and application areas. Testing for graphical user interfaces, client/server architectures, documentation and help facilities, and real-time systems each require specialized guidelines and techniques.

Experienced software developers often say, "Testing never ends, it just gets transferred from you [the software engineer] to your customer. Every time your customer uses the program, a test is being conducted." By applying test case design, the software engineer can achieve more complete testing and thereby uncover and correct the highest number of errors before the "customer's tests" begin.

#### REFERENCES

[BEI90] Beizer, B., *Software Testing Techniques*, 2nd ed., Van Nostrand-Reinhold, 1990.

[BEI95] Beizer, B., Black-Box Testing, Wiley, 1995.

[BRI87] Brilliant, S.S., J.C. Knight, and N.G. Levenson, "The Consistent Comparison Problem in N-Version Software," *ACM Software Engineering Notes*, vol. 12, no. 1, January 1987, pp. 29–34.

[DAV95] Davis, A., 201 Principles of Software Development, McGraw-Hill, 1995.

[DEU79] Deutsch, M., "Verification and Validation," in *Software Engineering* (R. Jensen and C. Tonies, eds.), Prentice-Hall, 1979, pp. 329–408.

[FOS84] Foster, K.A., "Sensitive Test Data for Boolean Expressions," *ACM Software Engineering Notes*, vol. 9, no. 2, April 1984, pp. 120–125.

[FRA88] Frankl, P.G. and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Engineering*, vol. SE-14, no. 10, October 1988, pp. 1483–1498.

[FRA93] Frankl, P.G. and S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow," *IEEE Trans. Software Engineering,* vol. SE-19, no. 8, August 1993, pp. 770–787.

[HET84] Hetzel, W., The Complete Guide to Software Testing, QED Information Sciences, 1984.

[HOW82] Howden, W.E., "Weak Mutation Testing and the Completeness of Test Cases," *IEEE Trans. Software Engineering*, vol. SE-8, no. 4, July 1982, pp. 371–379.

[JON81] Jones, T.C., *Programming Productivity: Issues for the 80s,* IEEE Computer Society Press, 1981.

[KAN93] Kaner, C., J. Falk, and H.Q. Nguyen, *Testing Computer Software,* 2nd ed., Van Nostrand-Reinhold, 1993.

[KNI89] Knight, J. and P. Ammann, "Testing Software Using Multiple Versions," Software Productivity Consortium, Report No. 89029N, Reston, VA, June 1989.

[MCC76] McCabe, T., "A Software Complexity Measure," *IEEE Trans. Software Engineering*, vol. SE-2, December 1976, pp. 308–320.

[MYE79] Myers, G., The Art of Software Testing, Wiley, 1979.

[NTA88] Ntafos, S.C., A Comparison of Some Structural Testing Strategies," *IEEE Trans. Software Engineering*, vol. SE-14, no. 6, June 1988, pp. 868–874.

[PHA89] Phadke, M.S., Quality Engineering Using Robust Design, Prentice-Hall, 1989.

[PHA97] Phadke, M.S., "Planning Efficient Software Tests," *Crosstalk,* vol. 10, no. 10, October 1997, pp. 11–15.

[TAI87] Tai, K.C. and H.K. Su, "Test Generation for Boolean Expressions," *Proc. COMPSAC '87*, October 1987, pp. 278–283.

[TAI89] Tai, K.C., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, no. 2, April 1989, pp. 58–61.

[WHI80] White, L.J. and E.I. Cohen, "A Domain Strategy for Program Testing," *IEEE Trans. Software Engineering*, vol. SE-6, no. 5, May 1980, pp. 247–257.

#### PROBLEMS AND POINTS TO PONDER

- **17.1.** Myers [MYE79] uses the following program as a self-assessment for your ability to specify adequate testing: A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral. Develop a set of test cases that you feel will adequately test this program.
- **17.2.** Design and implement the program (with error handling where appropriate) specified in Problem 1. Derive a flow graph for the program and apply basis path testing to develop test cases that will guarantee that all statements in the program have been tested. Execute the cases and show your results.
- **17.3.** Can you think of any additional testing objectives that are not discussed in Section 17.1.1?
- **17.4.** Apply the basis path testing technique to any one of the programs that you have implemented in Problems 16.4 through 16.11.
- **17.5.** Specify, design, and implement a software tool that will compute the cyclomatic complexity for the programming language of your choice. Use the graph matrix as the operative data structure in your design.
- **17.6.** Read Beizer [BEI95] and determine how the program you have developed in Problem 17.5 can be extended to accommodate various link weights. Extend your tool to process execution probabilities or link processing times.

- **17.7.** Use the condition testing approach described in Section 17.5.1 to design a set of test cases for the program you created in Problem 17.2.
- **17.8.** Using the data flow testing approach described in Section 17.5.2, make a list of definition-use chains for the program you created in Problem 17.2.
- **17.9.** Design an automated tool that will recognize loops and categorize them as indicated in Section 17.5.3.
- **17.10.** Extend the tool described in Problem 17.9 to generate test cases for each loop category, once encountered. It will be necessary to perform this function interactively with the tester.
- **17.11.** Give at least three examples in which black-box testing might give the impression that "everything's OK," while white-box tests might uncover an error. Give at least three examples in which white-box testing might give the impression that "everything's OK," while black-box tests might uncover an error.
- **17.12.** Will exhaustive testing (even if it is possible for very small programs) guarantee that the program is 100 percent correct?
- **17.13.** Using the equivalence partitioning method, derive a set of test cases for *Safe-Home* system described earlier in this book.
- **17.14.** Using boundary value analysis, derive a set of test cases for the PHTRS system described in Problem 12.13.
- **17.15.** Do a bit of outside research and write a brief paper that discusses the mechanics for generating orthogonal arrays for test data.
- **17.16.** Select a specific GUI for a program with which you are familiar and design a series of tests to exercise the GUI.
- **17.17.** Do some research on a client/server system with which you are familiar. Develop a set of user scenarios and then create an operational profile for the system.
- **17.18.** Test a user manual (or help facility) for an application that you use frequently. Find at least one error in the documentation.

#### FURTHER READINGS AND INFORMATION SOURCES

Software engineering presents both technical and management challenges. Books by Black (*Managing the Testing Process*, Microsoft Press, 1999); Dustin, Rashka, and Paul (*Test Process Improvement: Step-by-Step Guide to Structured Testing*, Addison-Wesley, 1999); Perry (*Surviving the Top Ten Challenges of Software Testing: A People-Oriented Approach*, Dorset House, 1997); and Kit and Finzi (*Software Testing in the Real World: Improving the Process*, Addison-Wesley, 1995) address management and process issues.

A number of excellent books are now available for those readers who desire additional information on software testing technology. Kaner, Nguyen, and Falk (*Testing Computer Software*, Wiley, 1999); Hutcheson (*Software Testing Methods and Metrics: The Most Important Tests, McGraw-Hill, 1997*); Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing, Prentice-Hall, 1995*); Jorgensen (*Software Testing : A Craftsman's Approach, CRC Press, 1995*) present treatments of the subject that consider testing methods and strategies.

Myers [MYE79] remains a classic text, covering black-box techniques in considerable detail. Beizer [BEI90] provides comprehensive coverage of white-box techniques, introducing a level of mathematical rigor that has often been missing in other treatments of testing. His later book [BEI95] presents a concise treatment of important methods. Perry (*Effective Methods for Software Testing, Wiley-QED, 1995*) and Friedman and Voas (*Software Assessment: Reliability, Safety, Testability, Wiley, 1995*) present good introductions to testing strategies and tactics. Mosley (*The Handbook of MIS Application Software Testing, Prentice-Hall, 1993*) discusses testing issues for large information systems, and Marks (*Testing Very Big Systems, McGraw-Hill, 1992*) discusses the special issues that must be considered when testing major programming systems.

Software testing is a resource-intensive activity. It is for this reason that many organizations automate parts of the testing process. Books by Dustin, Rashka, and Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999) and Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) discuss tools, strategies, and methods for automated testing. An excellent source of information on automated tools for software testing is the *Testing Tools Reference Guide* (Software Quality Engineering, Jacksonville, FL, updated yearly). This directory contains descriptions of hundreds of testing tools, categorized by testing activity, hardware platform, and software support.

A number of books consider testing methods and strategies in specialized application areas. Gardiner (*Testing Safety-Related Software: A Practical Handbook,* Springer-Verlag, 1999) has edited a book that addresses testing of safety-critical systems. Mosley (*Client/Server Software Testing on the Desk Top and the Web,* Prentice-Hall, 1999) discusses the test process for clients, servers, and network components. Rubin (*Handbook of Usability Testing,* Wiley, 1994) has written a useful guide for those who must exercise human interfaces.

A wide variety of information sources on software testing and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to testing concepts, methods, and strategies can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/sepa/resources/test-techniques.mhtml