SOFTWARE TESTING TECHNIQUES

The 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-

<table>
<tr><td>

QUICK LOOK

</td><td>

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-

</td><td>

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.

</td></tr>
</table>

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

Quote:

"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:

> **?** **What are primary objectives when we test software?**

1. 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.

> **Quote:**
>
> "Errors are more common, more pervasive, and more troublesome in software than with other technologies."
>
> **David Parnas**

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[1] 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

---

1   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 Bach[2] 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

---

2 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).

KEY POINT

"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

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

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.[3] 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 representation in Figure 17.2A. Here, a flowchart is used to depict program control structure.
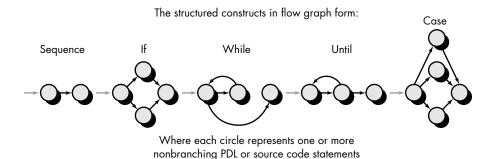
---

3   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

The structured constructs in flow graph form:

Sequence   If   While   Until   Case

Where each circle represents one or more
nonbranching PDL or source code statements

ADVICE

*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). Refer-
ring 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,* repre-
sent 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.[4]

When compound conditions are encountered in a procedural design, the genera-
tion 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 *pred-
icate node* and is characterized by two or more edges emanating from it.
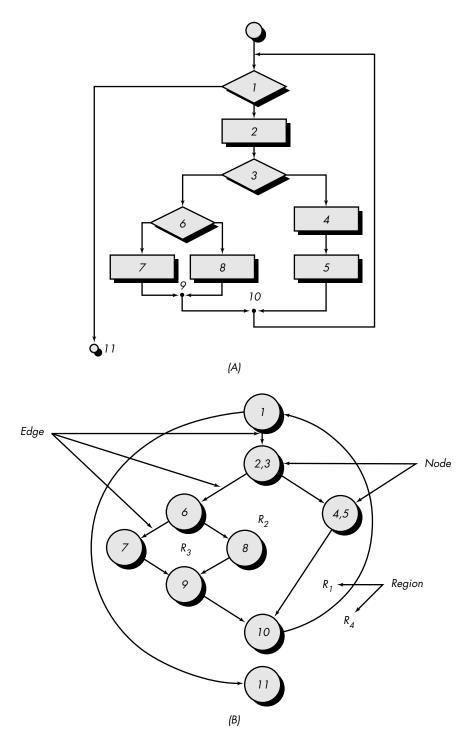
17.4.2 Cyclomatic Complexity

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 inde-
pendent 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

---

4   A more-detailed discussion of graphs and their use in testing is contained in Section 17.6.1.
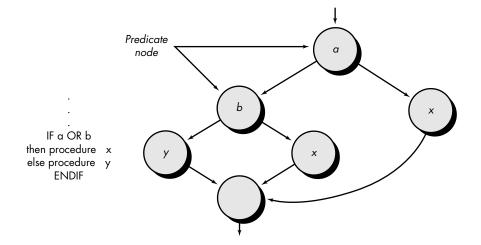
FIGURE 17.2

Flowchart, (A)
and flow
graph (B)

(A)

(B)

FIGURE 17.3
Compound
logic

graph, <mark>an independent path must move along at least one edge that has not been traversed before the path is defined.</mark> 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

ADVICE

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.

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

    1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

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 <mark>*basis set*</mark> 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:

? How is cyclomatic complexity computed?

  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

$$\boxed{V(G) = E - N + 2}$$

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
   $V(G) = 17$ edges $- 13$ nodes $+ 2 = 6$
   $V(G) = 5$ predicate nodes $+ 1 = 6$

FIGURE 17.4

PDL for test case design with nodes identified

```
PROCEDURE 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;        ②
    sum = 0;
①   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
      ⑤      THEN increment total.valid by 1;
         ⑦         sum = s sum + value[i]
                ELSE skip
             ENDIF
      ⑧   increment i by 1;
      ⑨ ENDDO
         IF total.valid > 0    ⑩
          ⑪   THEN average = sum / total.valid;
      ⑫       ELSE average = –999;
          ⑬ ENDIF
    END average
```

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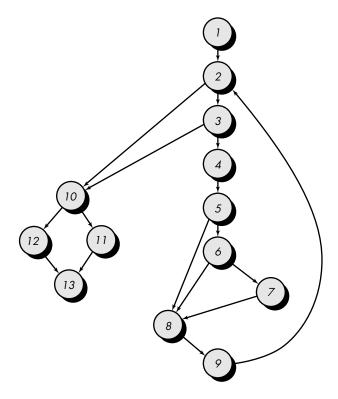
    path 5:   1-2-3-4-5-6-8-9-2-. . .

    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

Path 1 test case:
value(k) = valid input, where k < i for 2 ≤ i ≤ 100
value(i) = −999 where 2 ≤ i ≤ 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.

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:

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

- 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.

FIGURE 17.6

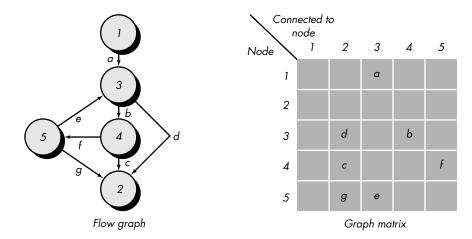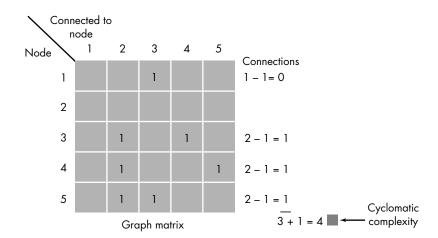Graph matrix

Flow graph

Graph matrix

FIGURE 17.7

Connection
matrix

Graph 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.

17.5.1   Condition Testing[5]

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 (¬) 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 (¬). 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

---

5   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

$$E_1 \text{ <relational-operator> } E_2$$

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 $(0 < i \leq n)$ 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.

As an example, consider the condition

C_1: B_1 & B_2

where B_1 and B_2 are Boolean variables. The condition constraint for C_1 is of the form (D_1, D_2), where each of D_1 and D_2 is t or f. The value (t, f) is a condition constraint for C_1 and is covered by the test that makes the value of B_1 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_1. If C_1 is incorrect due to one or more Boolean operator errors, at least one of the constraint set will force C_1 to fail.

As a second example, a condition of the form

C_2: B_1 & $(E_3 = E_4)$

where B_1 is a Boolean expression and E_3 and E_4 are arithmetic expressions. A condition constraint for C_2 is of the form (D_1, D_2), where each of D_1 is t or f and D_2 is >, =, <. Since C_2 is the same as C_1 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_1. 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 | statement S contains a definition of X}

USE(S) = {X | 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, $0 < i \leq 5$,

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:

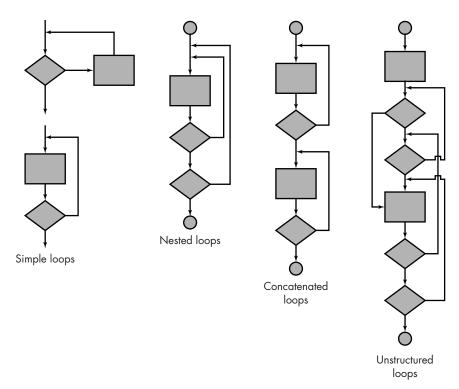1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

ADVICE

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

Simple loops

Nested loops

Concatenated
loops

Unstructured
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.

> **ADVICE**
>
> *You can't test unstructured loops effectively. Redesign them.*

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

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[6] 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.[7]
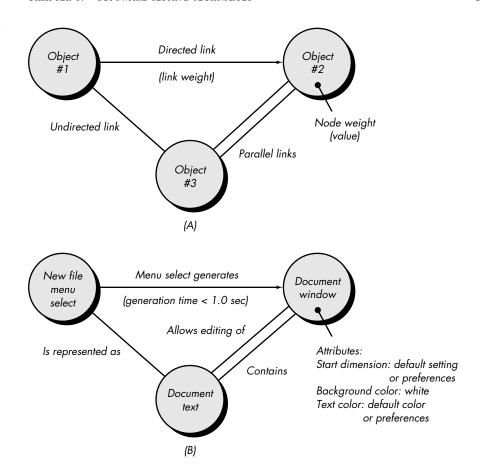
---

6 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.

7 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.

FIGURE 17.9
(A) Graph
notation
(B) Simple
example

FIGURE 17.9 (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.

What generic activities are required during graph-based testing?

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.

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 **Z** must consider a variety of values for both **X** and **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.

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

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:

? How do I create BVA test cases?

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.
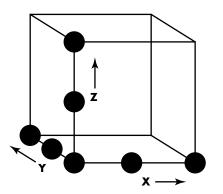
To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, *X, Y,* and *Z.* Each of these input items has three discrete values associated with it. There are $3^3 = 27$ possible test cases. Phadke [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
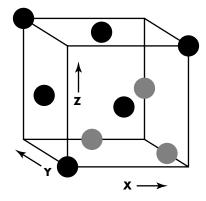
One input item at a time   **L9 orthogonal array**

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

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.

Testing GUIs

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.

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:
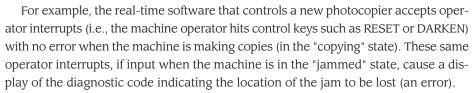
- Does the documentation accurately describe how to accomplish each mode of use?

- 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.

For example, the real-time software that controls a new photocopier accepts operator interrupts (i.e., the machine operator hits control keys such as RESET or DARKEN) with no error when the machine is making copies (in the "copying" state). These same operator interrupts, if input when the machine is in the "jammed" state, cause a display of the diagnostic code indicating the location of the jam to be lost (an 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:

Real-Time Systems

> **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.

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 *SafeHome* 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

A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in the successful construction of software. The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to promote reasonable planning and management tracking as the project progresses. Shooman [SHO83] discusses these issues:

In many ways, testing is an individualistic process, and the number of different types of tests varies as much as the different development approaches. For many years, our only defense against programming errors was careful design and the native intelligence of the programmer. We are now in an era in which modern design techniques [and formal technical reviews] are helping us to reduce the number of initial errors that are inherent in the code. Similarly, different test methods are beginning to cluster themselves into several distinct approaches and philosophies.

QUICK LOOK

What is it? Designing effective test cases (Chapter 17) is important, but so is the strategy you use to execute them. Should you develop a formal plan for your tests? Should you test the entire program as a whole or run tests only on a small part of it? Should you rerun tests you've already conducted as you add new components to a large system? When should you involve the customer? These and many other questions are answered when you develop a software testing strategy.

Who does it? A strategy for software testing is developed by the project manager, software engineers, and testing specialists.

Why is it important? Testing often accounts for more project effort than any other software engineering activity. If it is conducted haphazardly, time is wasted, unnecessary effort is expended, and even worse, errors sneak through undetected. It would therefore seem reasonable to establish a systematic strategy for testing software.

What are the steps? Testing begins "in the small" and progresses "to the large." By this we mean that early testing focuses on a single component and applies white- and black-box tests to uncover errors in program logic and function. After individual components are tested they must be integrated. Testing continues as the software is constructed. Finally, a series of high-order tests are executed once the full program is operational.

<table>
<tr>
<td>

**QUICK
LOOK**

</td>
<td>

These tests are designed to
uncover errors in requirements.
What is the work product? A
Test Specification documents the software team's
approach to testing by defining a plan that
describes an overall strategy and a procedure
that defines specific testing steps and the tests that
will be conducted.

</td>
<td>

How do I ensure that I've done it right? By reviewing
the Test Specification prior to testing, you can
assess the completeness of test cases and testing
tasks. An effective test plan and procedure will
lead to the orderly construction of the software
and the discovery of errors at each stage in the
construction process.

</td>
</tr>
</table>

These "approaches and philosophies" are what we shall call *strategy*. In Chapter
17, the technology of software testing was presented.[1] In this chapter, we focus our
attention on the strategy for software testing.

18.1   A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted system-
atically. For this reason a template for software testing—a set of steps into which we
can place specific test case design techniques and testing methods—should be defined
for the software process.

A number of software testing strategies have been proposed in the literature. All
provide the software developer with a template for testing and all have the follow-
ing generic characteristics:

- Testing begins at the component level[2] and works "outward" toward the inte-
  gration of the entire computer-based system.

- Different testing techniques are appropriate at different points in time.

- Testing is conducted by the developer of the software and (for large projects)
  an independent test group.

- Testing and debugging are different activities, but debugging must be accom-
  modated in any testing strategy.

A strategy for software testing must accommodate low-level tests that are necessary
to verify that a small source code segment has been correctly implemented as well
as high-level tests that validate major system functions against customer require-
ments. A strategy must provide guidance for the practitioner and a set of milestones
for the manager. Because the steps of the test strategy occur at a time when dead-

---

1   Testing for object-oriented systems is discussed in Chapter 23.
2   For object-oriented systems, testing begins at the class or object level. See Chapter 23 for details.

line pressure begins to rise, progress must be measurable and problems must surface as early as possible.

18.1.1 Verification and Validation

Software testing is one element of a broader topic that is often referred to as *verification and validation* (V&V). *Verification* refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm [BOE81] states this another way:

> *Verification:*   "Are we building the product right?"
> *Validation:*   "Are we building the right product?"

The definition of V&V encompasses many of the activities that we have referred to as *software quality assurance* (SQA).

Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing [WAL89]. Although testing plays an extremely important role in V&V, many other activities are also necessary.

Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective formal technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

Miller [MIL77] relates software testing to quality assurance by stating that "the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems."

18.1.2 Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests mitigate against thorough testing.

From a psychological point of view, software analysis and design (along with coding) are *constructive* tasks. The software engineer creates a computer program, its documentation, and related data structures. Like any builder, the software engineer is proud of the edifice that has been built and looks askance at anyone who attempts to tear it down. When testing commences, there is a subtle, yet definite, attempt to "break" the thing that the software engineer has built. From the point of view of the builder, testing can be considered to be (psychologically) *destructive.* So the builder treads lightly, designing and executing tests that will demonstrate that the program works, rather than uncovering errors. Unfortunately, errors will be present. And, if the software engineer doesn't find them, the customer will!

There are often a number of misconceptions that can be erroneously inferred from the preceeding discussion: (1) that the developer of software should do no testing at all, (2) that the software should be "tossed over the wall" to strangers who will test it mercilessly, (3) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete program structure. Only after the software architecture is complete does an independent test group become involved.

The role of an *independent test group* (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. After all, personnel in the independent group team are paid to find errors.

However, the software engineer doesn't turn the program over to ITG and walk away. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

The ITG is part of the software development project team in the sense that it becomes involved during the specification activity and stays involved (planning and specifying test procedures) throughout a large project. However, in many cases the ITG reports to the software quality assurance organization, thereby achieving a degree of independence that might not be possible if it were a part of the software engineering organization.

18.1.3 A Software Testing Strategy

The software engineering process may be viewed as the spiral illustrated in Figure 18.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving
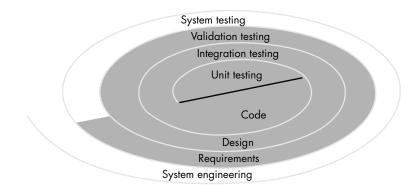
FIGURE 18.1

Testing
strategy

inward along the spiral, we come to design and finally to coding. To develop com-
puter software, we spiral inward along streamlines that decrease the level of abstrac-
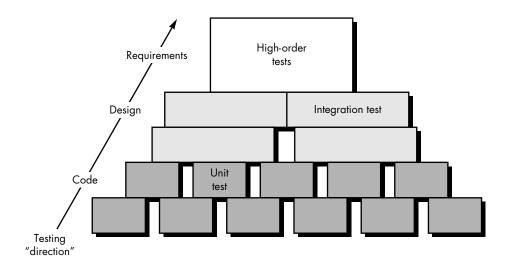tion on each turn.

A strategy for software testing may also be viewed in the context of the spiral (Fig-
ure 18.1). *Unit testing* begins at the vortex of the spiral and concentrates on each unit
(i.e., component) of the software as implemented in source code. Testing progresses
by moving outward along the spiral to *integration testing,* where the focus is on design
and the construction of the software architecture. Taking another turn outward on
the spiral, we encounter *validation testing,* where requirements established as part of
software requirements analysis are validated against the software that has been con-
structed. Finally, we arrive at *system testing,* where the software and other system
elements are tested as a whole. To test computer software, we spiral out along stream-
lines that broaden the scope of testing with each turn.

Considering the process from a procedural point of view, testing within the con-
text of software engineering is actually a series of four steps that are implemented
sequentially. The steps are shown in Figure 18.2. Initially, tests focus on each com-
ponent individually, ensuring that it functions properly as a unit. Hence, the name
unit testing. Unit testing makes heavy use of white-box testing techniques, exercis-
ing specific paths in a module's control structure to ensure complete coverage and
maximum error detection. Next, components must be assembled or integrated to
form the complete software package. Integration testing addresses the issues asso-
ciated with the dual problems of verification and program construction. Black-box
test case design techniques are the most prevalent during integration, although a lim-
ited amount of white-box testing may be used to ensure coverage of major control
paths. After the software has been integrated (constructed), a set of high-order tests
are conducted. Validation criteria (established during requirements analysis) must be
tested. Validation testing provides final assurance that software meets all functional,
behavioral, and performance requirements. Black-box testing techniques are used
exclusively during validation.

? What is the overall strategy for software testing?

XRef

Black-box and white-
box testing techniques
are discussed in
Chapter 17.

FIGURE 18.2

Software
testing steps

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

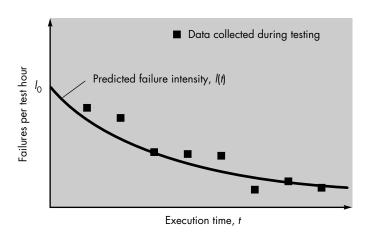### 18.1.4 Criteria for Completion of Testing

A classic question arises every time software testing is discussed: "When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

? **When are we done testing?**

One response to the question is: "You're never done testing, the burden simply shifts from you (the software engineer) to your customer." Every time the customer/user executes a computer program, the program is being tested. This sobering fact underlines the importance of other software quality assurance activities. Another response (somewhat cynical but nonetheless accurate) is: "You're done testing when you run out of time or you run out of money."

Although few practitioners would argue with these responses, a software engineer needs more rigorous criteria for determining when sufficient testing has been conducted. Musa and Ackerman [MUS89] suggest a response that is based on statistical criteria: "No, we cannot be absolutely certain that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95 percent confidence that the probability of 1000 CPU hours of failure free operation in a probabilistically defined environment is at least 0.995."

FIGURE 18.3

Failure
intensity as a
function of
execution time

    Using statistical modeling and software reliability theory, models of software fail-
ures (uncovered during testing) as a function of execution time can be developed
[MUS89]. A version of the failure model, called a *logarithmic Poisson execution-time
model,* takes the form

$$f(t) = (1/p) \ln [l_0 \, pt + 1] \tag{18-1}$$

where $f(t)$ = cumulative number of failures that are expected to occur once the
            software has been tested for a certain amount of execution time, t,
        l_0 = the initial software failure intensity (failures per time unit) at the begin-
            ning of testing,
        p = the exponential reduction in failure intensity as errors are uncovered
            and repairs are made.

    The instantaneous failure intensity, $l(t)$ can be derived by taking the derivative of
$f(t)$

$$l(t) = l_0 \, / \, (l_0 \, pt + 1) \tag{18-2}$$

Using the relationship noted in Equation (18-2), testers can predict the drop-off of
errors as testing progresses. The actual error intensity can be plotted against the pre-
dicted curve (Figure 18.3). If the actual data gathered during testing and the loga-
rithmic Poisson execution time model are reasonably close to one another over a
number of data points, the model can be used to predict total testing time required
to achieve an acceptably low failure intensity.

    By collecting metrics during software testing and making use of existing software
reliability models, it is possible to develop meaningful guidelines for answering the
question: "When are we done testing?" There is little debate that further work remains
to be done before quantitative rules for testing can be established, but the empirical
approaches that currently exist are considerably better than raw intuition.

18.2 STRATEGIC ISSUES

Later in this chapter, we explore a systematic strategy for software testing. But even the best strategy will fail if a series of overriding issues are not addressed. Tom Gilb [GIL95] argues that the following issues must be addressed if a successful software testing strategy is to be implemented:

? **What guidelines lead to a successful testing strategy?**

Specify product requirements in a quantifiable manner long before testing commences. Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability (Chapter 19). These should be specified in a way that is measurable so that testing results are unambiguous.

State testing objectives explicitly. The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, mean time to failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours per regression test all should be stated within the test plan [GIL95].

XRef

Use-cases describe a scenario for software use and are discussed in Chapter 11.

Understand the users of the software and develop a profile for each user category. Use-cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.

Develop a testing plan that emphasizes "rapid cycle testing." Gilb [GIL95] recommends that a software engineering team "learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field 'trialable,' increments of functionality and/or quality improvement." The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.

Quote:

"Testing only to end-user perceived requirements is like inspecting a building based on the work done by the interior decorator at the expense of the foundations, girders, and plumbing."

Boris Beizer

Build "robust" software that is designed to test itself. Software should be designed in a manner that uses antibugging (Section 18.3.1) techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.

Use effective formal technical reviews as a filter prior to testing. Formal technical reviews (Chapter 8) can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high-quality software.

Conduct formal technical reviews to assess the test strategy and test cases themselves. Formal technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.

Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

18.3   UNIT TESTING

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered errors is limited by the constrained scope established for unit testing. The unit test is white-box oriented, and the step can be conducted in parallel for multiple components.

18.3.1 Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure 18.4. The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.
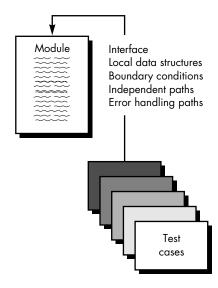
Unit Testing

Module

Interface
Local data structures
Boundary conditions
Independent paths
Error handling paths

Test cases

FIGURE 18.4
Unit test

Tests of data flow across a module interface are required before any other test is initiated. If data do not enter and exit properly, all other tests are moot. In addition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors.

Among the more common errors in computation are (1) misunderstood or incorrect arithmetic precedence, (2) mixed mode operations, (3) incorrect initialization, (4) precision inaccuracy, (5) incorrect symbolic representation of an expression. Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison). Test cases should uncover errors such as (1) comparison of different data types, (2) incorrect logical operators or precedence, (3) expectation of equality when precision error makes equality unlikely, (4) incorrect comparison of variables, (5) improper or nonexistent loop termination, (6) failure to exit when divergent iteration is encountered, and (7) improperly modified loop variables.

Good design dictates that error conditions be anticipated and error-handling paths set up to reroute or cleanly terminate processing when an error does occur. Yourdon [YOU75] calls this approach *antibugging*. Unfortunately, there is a tendency to incorporate error handling into software and then never test it. A true story may serve to illustrate:

A major interactive design system was developed under contract. In one transaction processing module, a practical joker placed the following error handling message after a series of conditional tests that invoked various control flow branches: ERROR! THERE IS NO WAY YOU CAN GET HERE. This "error message" was uncovered by a customer during user training!

Among the potential errors that should be tested when error handling is evaluated are

1. Error description is unintelligible.
2. Error noted does not correspond to error encountered.
3. Error condition causes system intervention prior to error handling.
4. Exception-condition processing is incorrect.
5. Error description does not provide enough information to assist in the location of the cause of the error.
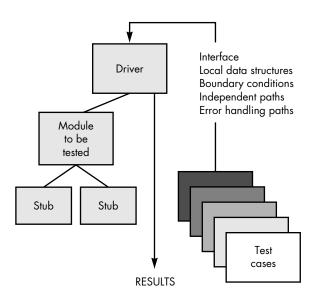
Boundary testing is the last (and probably most important) task of the unit test step. Software often fails at its boundaries. That is, errors often occur when the nth element of an n-dimensional array is processed, when the ith repetition of a loop with

FIGURE 18.5
Unit test
environment

i passes is invoked, when the maximum or minimum allowable value is encountered. Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

18.3.2 Unit Test Procedures

ADVICE

There are some situations in which you will not have the resources to do comprehensive unit testing. Select critical modules and those with high cyclomatic complexity and unit test only them.

Unit testing is normally considered as an adjunct to the coding step. After source level code has been developed, reviewed, and verified for correspondence to component-level design, unit test case design begins. A review of design information provides guidance for establishing test cases that are likely to uncover errors in each of the categories discussed earlier. Each test case should be coupled with a set of expected results.

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure 18.5. In most applications a *driver* is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. *Stubs* serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

18.4 INTEGRATION TESTING[3]

A neophyte in the software world might ask a seemingly legitimate question once all modules have been unit tested: "If they all work individually, why do you doubt that they'll work when we put them together?" The problem, of course, is "putting them together"—interfacing. Data can be lost across an interface; one module can have an inadvertent, adverse affect on another; subfunctions, when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems. Sadly, the list goes on and on.

Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dictated by design.

> **ADVICE**
>
> *Taking the big bang approach to integration is a lazy strategy that is doomed to failure. Integration testing should be conducted incrementally.*

There is often a tendency to attempt nonincremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. In the sections that follow, a number of different incremental integration strategies are discussed.

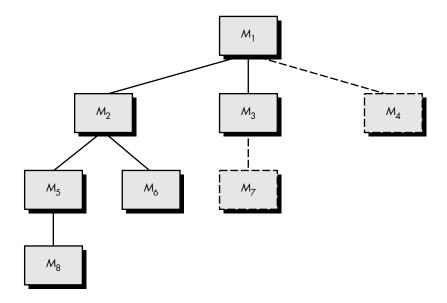### 18.4.1 Top-down Integration

> **ADVICE**
>
> *When you develop a detailed project schedule you have to consider the manner in which integration will occur so that components will be available when needed.*

Top-down integration testing is an incremental approach to construction of program structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Referring to Figure 18.6, *depth-first integration* would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M_1, M_2, M_5 would be integrated first. Next, M_8 or (if neces-

---

3   Integration strategies for object-oriented systems are discussed in Chapter 23.

FIGURE 18.6

Top-down
integration

sary for proper functioning of M_2) M_6 would be integrated. Then, the central and right-hand control paths are built. *Breadth-first integration* incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M_2, M_3, and M_4 (a replacement for stub S_4) would be integrated first. The next control level, M_5, M_6, and so on, follows.

The integration process is performed in a series of five steps:

What are the steps for top-down integration?

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing (Section 18.4.3) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated. For example, consider a classic transaction structure (Chapter 14) in which a complex series of interactive inputs is requested, acquired, and validated via an incoming path. The

XRef

Factoring is important for certain architectural styles. See Chapter 14 for additional details.

incoming path may be integrated in a top-down manner. All input processing (for subsequent transaction dispatching) may be demonstrated before other elements of the structure have been integrated. Early demonstration of functional capability is a confidence builder for both the developer and the customer.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure. The tester is left with three choices: (1) delay many tests until stubs are replaced with actual modules, (2) develop stubs that perform limited functions that simulate the actual module, or (3) integrate the software from the bottom of the hierarchy upward.

The first approach (delay tests until stubs are replaced by actual modules) causes us to loose some control over correspondence between specific tests and incorporation of specific modules. This can lead to difficulty in determining the cause of errors and tends to violate the highly constrained nature of the top-down approach. The second approach is workable but can lead to significant overhead, as stubs become more and more complex. The third approach, called *bottom-up testing*, is discussed in the next section.

18.4.2 Bottom-up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

? What are the steps for bottom-up integration?

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

KEY POINT

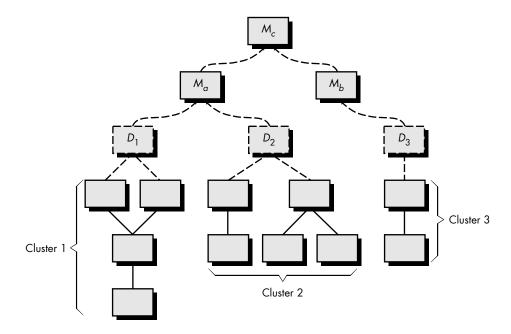Bottom-up integration eliminates the need for complex stubs.

Integration follows the pattern illustrated in Figure 18.7. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to M_a. Drivers D_1 and D_2 are removed and the clusters are interfaced directly to M_a. Similarly, driver D_3 for cluster 3 is removed prior to integration with module M_b. Both M_a and M_b will ultimately be integrated with component M_c, and so forth.

FIGURE 18.7
Bottom-up
integration

As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

18.4.3 Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, *regression testing* is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools.* Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

Regression testing is an important strategy for reducing "side effects." Run regression tests every time a major change is made to the software (including the integration of new modules).

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions. It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

18.4.4 Smoke Testing

Smoke testing is an integration testing approach that is commonly used when "shrink-wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. In essence, the smoke testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

The daily frequency of testing the entire product may surprise some readers. However, frequent tests give both managers and practitioners a realistic assessment of integration testing progress. McConnell [MCO96] describes the smoke test in the following manner:

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

Smoke testing provides a number of benefits when it is applied on complex, time-critical software engineering projects:

- *Integration risk is minimized.* Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.

- *The quality of the end-product is improved.* Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design defects. If these defects are corrected early, better product quality will result.

- *Error diagnosis and correction are simplified.* Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.

- *Progress is easier to assess.* With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

18.4.5 Comments on Integration Testing

There has been much discussion (e.g., [BEI84]) of the relative advantages and disadvantages of top-down versus bottom-up integration testing. In general, the advantages of one strategy tend to result in disadvantages for the other strategy. The major disadvantage of the top-down approach is the need for stubs and the attendant testing difficulties that can be associated with them. Problems associated with stubs may be offset by the advantage of testing major control functions early. The major disadvantage of bottom-up integration is that "the program as an entity does not exist until the last module is added" [MYE79]. This drawback is tempered by easier test case design and a lack of stubs.

Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes called *sandwich testing*) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

? What is a critical module and why should we identify it?

As integration testing is conducted, the tester should identify *critical modules.* A critical module has one or more of the following characteristics: (1) addresses several software requirements, (2) has a high level of control (resides relatively high in the program structure), (3) is complex or error prone (cyclomatic complexity may be used as an indicator), or (4) has definite performance requirements. Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

18.4.6 Integration Test Documentation

An overall plan for integration of the software and a description of specific tests are documented in a *Test Specification.* This document contains a test plan, and a test procedure, is a work product of the software process, and becomes part of the software configuration.

The test plan describes the overall strategy for integration. Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software. For example, integration testing for a CAD system might be divided into the following test phases:

- User interaction (command selection, drawing creation, display representation, error processing and representation).

- Data manipulation and analysis (symbol creation, dimensioning; rotation, computation of physical properties).

- Display processing and generation (two-dimensional displays, three-dimensional displays, graphs and charts).

- Database management (access, update, integrity, performance).

Each of these phases and subphases (denoted in parentheses) delineates a broad functional category within the software and can generally be related to a specific domain of the program structure. Therefore, program builds (groups of modules) are created to correspond to each phase. The following criteria and corresponding tests are applied for all test phases:

Interface integrity. Internal and external interfaces are tested as each module (or cluster) is incorporated into the structure.

Functional validity. Tests designed to uncover functional errors are conducted.

Information content. Tests designed to uncover errors associated with local or global data structures are conducted.

Performance. Tests designed to verify performance bounds established during software design are conducted.

A schedule for integration, the development of overhead software, and related topics is also discussed as part of the test plan. Start and end dates for each phase are established and "availability windows" for unit tested modules are defined. A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort. Finally, test environment and resources are described. Unusual hardware configurations, exotic simulators, and special test tools or techniques are a few of many topics that may also be discussed.

The detailed testing procedure that is required to accomplish the test plan is described next. The order of integration and corresponding tests at each integration

step are described. A listing of all test cases (annotated for subsequent reference) and expected results is also included.

A history of actual test results, problems, or peculiarities is recorded in the *Test Specification.* Information contained in this section can be vital during software maintenance. Appropriate references and appendixes are also presented.

Like all other elements of a software configuration, the test specification format may be tailored to the local needs of a software engineering organization. It is important to note, however, that an integration strategy (contained in a test plan) and testing details (described in a test procedure) are essential ingredients and must appear.

18.5 VALIDATION TESTING

At the culmination of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests—*validation testing*—may begin. Validation can be defined in many ways, but a simple (albeit harsh) definition is that validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

Reasonable expectations are defined in the *Software Requirements Specification*—a document (Chapter 11) that describes all user-visible attributes of the software. The specification contains a section called *Validation Criteria.* Information contained in that section forms the basis for a validation testing approach.

KEY POINT

Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level—on things that will be immediately apparent to the end-user.

18.5.1 Validation Test Criteria

Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct, and human-engineered and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability).

After each validation test case has been conducted, one of two possible conditions exist: (1) The function or performance characteristics conform to specification and are accepted or (2) a deviation from specification is uncovered and a *deficiency list* is created. Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery. It is often necessary to negotiate with the customer to establish a method for resolving deficiencies.

18.5.2 Configuration Review

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometimes called an *audit,* has been discussed in more detail in Chapter 9.

18.5.3 Alpha and Beta Testing

It is virtually impossible for a software developer to foresee how the customer will really use a program. Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field.

When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find.

The *alpha test* is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The *beta test* is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

18.6 SYSTEM TESTING

At the beginning of this book, we stressed the fact that software is only one element of a larger computer-based system. Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the

software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem. Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems and (1) design error-handling paths that test all information coming from other elements of the system, (2) conduct a series of tests that simulate bad data or other potential errors at the software interface, (3) record the results of tests to use as "evidence" if finger-pointing does occur, and (4) participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests [BEI84] that are worthwhile for software-based systems.

18.6.1 Recovery Testing

Many computer based systems must recover from faults and resume processing within a prespecified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

18.6.2 Security Testing

Any computer-based system that manages sensitive information or causes actions that can improperly harm (or benefit) individuals is a target for improper or illegal penetration. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal gain.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. To quote Beizer [BEI84]: "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes! The tester may attempt to acquire passwords

through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the system, thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the key to system entry.

Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

18.6.3 Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: "How high can we crank this up before it fails?"

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called *sensitivity testing.* In some situations (the most common occur in mathematical algorithms), a very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound performance degradation. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

18.6.4 Performance Testing

For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable. *Performance testing* is designed to test the run-time performance of software within the context of an integrated system. Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system elements are fully integrated that the true performance of a system can be ascertained.
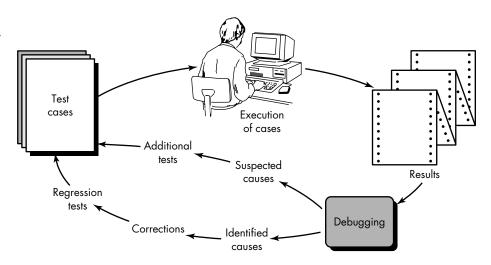
Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion. External instru-

mentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

18.7   THE ART OF DEBUGGING

Software testing is a process that can be systematically planned and specified. Test case design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. A software engineer, evaluating the results of a test, is often confronted with a "symptomatic" indication of a software problem. That is, the external manifestation of the error and the internal cause of the error may have no obvious relationship to one another. The poorly understood mental process that connects a symptom to a cause is debugging.

18.7.1 The Debugging Process

Debugging is not testing but always occurs as a consequence of testing.[4] Referring to Figure 18.8, the debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a symptom

---

4   In making this statement, we take the broadest possible view of testing. Not only does the developer test software prior to release, but the customer/user tests software every time it is used!

of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

Why is debugging so difficult? In all likelihood, human psychology (see the next section) has more to do with an answer than software technology. However, a few characteristics of bugs provide some clues:

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures (Chapter 13) exacerbate this situation.

2. The symptom may disappear (temporarily) when another error is corrected.

3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).

4. The symptom may be caused by human error that is not easily traced.

5. The symptom may be a result of timing problems, rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on different processors [CHE90].

During debugging, we encounter errors that range from mildly annoying (e.g., an incorrect output format) to catastrophic (e.g. the system fails, causing serious economic or physical damage). As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

18.7.2 Psychological Considerations

Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren't. Although experimental evidence on debugging is open to many interpretations, large variances in debugging ability have been reported for programmers with the same education and experience.

Commenting on the human aspects of debugging, Shneiderman [SHN80] states:

Debugging is one of the more frustrating parts of programming. It has elements of problem solving or brain teasers, coupled with the annoying recognition that you have made a mistake. Heightened anxiety and the unwillingness to accept the possibility of errors increases the task difficulty. Fortunately, there is a great sigh of relief and a lessening of tension when the bug is ultimately . . . corrected.

Although it may be difficult to "learn" debugging, a number of approaches to the problem can be proposed. We examine these in the next section.

18.7.3 Debugging Approaches

Regardless of the approach that is taken, debugging has one overriding objective: to find and correct the cause of a software error. The objective is realized by a combination of systematic evaluation, intuition, and luck. Bradley [BRA85] describes the debugging approach in this way:

Debugging is a straightforward application of the scientific method that has been developed over 2,500 years. The basis of debugging is to locate the problem's source [the cause] by binary partitioning, through working hypotheses that predict new values to be examined.

Take a simple non-software example: A lamp in my house does not work. If nothing in the house works, the cause must be in the main circuit breaker or outside; I look around to see whether the neighborhood is blacked out. I plug the suspect lamp into a working socket and a working appliance into the suspect circuit. So goes the alternation of hypothesis and test.

In general, three categories for debugging approaches may be proposed [MYE79]: (1) brute force, (2) backtracking, and (3) cause elimination.

Set a time limit, say two hours, on the amount of time you spend trying to debug a problem on your own. After that, get help!

The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error. Although the mass of information produced may ultimately lead to success, it more frequently leads to wasted effort and time. Thought must be expended first!

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

The third approach to debugging—*cause elimination*—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is

devised and the aforementioned data are used to prove or disprove the hypothesis. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each. If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug.

Each of these debugging approaches can be supplemented with debugging tools. We can apply a wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test case generators, memory dumps, and cross-reference maps. However, tools are not a substitute for careful evaluation based on a complete software design document and clear source code.

CASE Tools
Testing and Debugging

Any discussion of debugging approaches and tools is incomplete without mention of a powerful ally—other people! Each of us can recall puzzling for hours or days over a persistent bug. A colleague wanders by and in desperation we explain the problem and throw open the listing. Instantaneously (it seems), the cause of the error is uncovered. Smiling smugly, our colleague wanders off. A fresh viewpoint, unclouded by hours of frustration, can do wonders. A final maxim for debugging might be: "When all else fails, get help!"

Once a bug has been found, it must be corrected. But, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good. Van Vleck [VAN89] suggests three simple questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

When I correct an error, what questions should I ask myself?

1. **Is the cause of the bug reproduced in another part of the program?** In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.

2. **What "next bug" might be introduced by the fix I'm about to make?** Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.

3. **What could we have done to prevent this bug in the first place?** This question is the first step toward establishing a statistical software quality assurance approach (Chapter 8). If we correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

18.8 SUMMARY

Software testing accounts for the largest percentage of technical effort in the software process. Yet we are only beginning to understand the subtleties of systematic test planning, execution, and control.

The objective of software testing is to uncover errors. To fulfill this objective, a series of test steps—unit, integration, validation, and system tests—are planned and executed. Unit and integration tests concentrate on functional verification of a component and incorporation of components into a program structure. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system.

Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened.

Unlike testing (a systematic, planned activity), debugging must be viewed as an art. Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error. Of the many resources available during debugging, the most valuable is the counsel of other members of the software engineering staff.

The requirement for higher-quality software demands a more systematic approach to testing. To quote Dunn and Ullman [DUN82],

What is required is an overall strategy, spanning the strategic test space, quite as deliberate in its methodology as was the systematic development on which analysis, design and code were based.

In this chapter, we have examined the strategic test space, considering the steps that have the highest likelihood of meeting the overriding test objective: to find and remove errors in an orderly and effective manner.

REFERENCES

[BEI84]   Beizer, B., *Software System Testing and Quality Assurance,* Van Nostrand-Reinhold, 1984.

[BOE81]   Boehm, B., *Software Engineering Economics,* Prentice-Hall, 1981, p. 37.

[BRA85]   Bradley, J.H., "The Science and Art of Debugging," *Computerworld,* August 19, 1985, pp. 35–38.

[CHE90]   Cheung, W.H., J.P. Black, and E. Manning, "A Framework for Distributed Debugging," *IEEE Software,* January 1990, pp. 106–115.

[DUN82] Dunn, R. and R. Ullman, *Quality Assurance for Computer Software,* McGraw-Hill, 1982, p. 158.

[GIL95]   Gilb, T., "What We Fail to Do in Our Current Testing Culture," *Testing Techniques Newsletter,* (on-line edition, ttn@soft.com), Software Research, January 1995.

[MCO96] McConnell, S., "Best Practices: Daily Build and Smoke Test", *IEEE Software,* vol. 13, no. 4, July 1996, 143–144.

[MIL77]   Miller, E., "The Philosophy of Testing," in *Program Testing Techniques,* IEEE Computer Society Press, 1977, pp. 1–3.

[MUS89] Musa, J.D. and Ackerman, A.F., "Quantifying Software Validation: When to Stop Testing?" *IEEE Software,* May 1989, pp. 19–27.

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

[SHO83] Shooman, M.L., *Software Engineering,* McGraw-Hill, 1983.

[SHN80] Shneiderman, B., *Software Psychology,* Winthrop Publishers, 1980, p. 28.

[VAN89] Van Vleck, T., "Three Questions About Each Bug You Find," *ACM Software Engineering Notes,* vol. 14, no. 5, July 1989, pp. 62–63.

[WAL89] Wallace, D.R. and R.U. Fujii, "Software Verification and Validation: An Overview," *IEEE Software,* May 1989, pp. 10–17.

[YOU75] Yourdon, E., *Techniques of Program Structure and Design,* Prentice-Hall, 1975.

PROBLEMS AND POINTS TO PONDER

18.1. Using your own words, describe the difference between verification and validation. Do both make use of test case design methods and testing strategies?

18.2. List some problems that might be associated with the creation of an independent test group. Are an ITG and an SQA group made up of the same people?

18.3. Is it always possible to develop a strategy for testing software that uses the sequence of testing steps described in Section 18.1.3? What possible complications might arise for embedded systems?

18.4. If you could select only three test case design methods to apply during unit testing, what would they be and why?

18.5. Why is a highly coupled module difficult to unit test?

18.6. The concept of "antibugging" (Section 18.2.1) is an extremely effective way to provide built-in debugging assistance when an error is uncovered:
    a. Develop a set of guidelines for antibugging.
    b. Discuss advantages of using the technique.
    c. Discuss disadvantages.

18.7. Develop an integration testing strategy for the any one of the systems implemented in Problems 16.4 through 16.11. Define test phases, note the order of integration, specify additional test software, and justify your order of integration. Assume that all modules (or classes) have been unit tested and are available. Note: it may be necessary to do a bit of design work first.

18.8. How can project scheduling affect integration testing?

18.9. Is unit testing possible or even desirable in all circumstances? Provide examples to justify your answer.

18.10. Who should perform the validation test—the software developer or the software user? Justify your answer.