# Chapter 16

## Software Testing

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1</td>
<td>Fundamentals of Software Testing</td>
<td>678</td>
</tr>
<tr>
<td>16.1.1</td>
<td>Basic Terminology</td>
<td>678</td>
</tr>
<tr>
<td>16.1.2</td>
<td>Basic Notions of Testing</td>
<td>679</td>
</tr>
<tr>
<td>16.1.3</td>
<td>Determining Test Cases</td>
<td>680</td>
</tr>
<tr>
<td>16.1.4</td>
<td>Levels of Testing</td>
<td>681</td>
</tr>
<tr>
<td>16.1.5</td>
<td>Psychology of Testing</td>
<td>681</td>
</tr>
<tr>
<td>16.1.6</td>
<td>Testing Principles</td>
<td>683</td>
</tr>
<tr>
<td>16.2</td>
<td>Human Testing</td>
<td>683</td>
</tr>
<tr>
<td>16.2.1</td>
<td>Code Readings</td>
<td>683</td>
</tr>
<tr>
<td>16.2.2</td>
<td>Structured Walk-Throughs</td>
<td>684</td>
</tr>
<tr>
<td>16.3</td>
<td>Black-Box Testing</td>
<td>684</td>
</tr>
<tr>
<td>16.3.1</td>
<td>Boundary Value Testing</td>
<td>685</td>
</tr>
<tr>
<td>16.3.2</td>
<td>Equivalence Class Testing</td>
<td>687</td>
</tr>
<tr>
<td>16.4</td>
<td>White-Box (Program-Based) Testing</td>
<td>691</td>
</tr>
<tr>
<td>16.5</td>
<td>Object-Oriented Testing</td>
<td>696</td>
</tr>
<tr>
<td>16.5.1</td>
<td>Issues in Testing Object-Oriented Software</td>
<td>696</td>
</tr>
<tr>
<td>16.5.2</td>
<td>Method Testing</td>
<td>697</td>
</tr>
<tr>
<td>16.5.3</td>
<td>Testing Recursive Methods</td>
<td>700</td>
</tr>
<tr>
<td>16.5.4</td>
<td>State-Based Class Testing</td>
<td>702</td>
</tr>
<tr>
<td>16.5.5</td>
<td>The Effect of Inheritance on Testing</td>
<td>706</td>
</tr>
<tr>
<td>16.5.6</td>
<td>Object-Oriented Integration Testing</td>
<td>708</td>
</tr>
<tr>
<td>16.5.7</td>
<td>Object-Oriented System Testing</td>
<td>709</td>
</tr>
<tr>
<td>16.6</td>
<td>Locating and Repairing Dynamic Faults</td>
<td>710</td>
</tr>
<tr>
<td>16.6.1</td>
<td>Planning for Debugging</td>
<td>711</td>
</tr>
<tr>
<td>16.6.2</td>
<td>Debugging by Brute Force</td>
<td>711</td>
</tr>
<tr>
<td>16.6.3</td>
<td>Debugging by Backtracking</td>
<td>712</td>
</tr>
<tr>
<td>16.6.4</td>
<td>Debugging by Induction</td>
<td>712</td>
</tr>
<tr>
<td>16.6.5</td>
<td>Debugging by Deduction</td>
<td>714</td>
</tr>
<tr>
<td>16.6.6</td>
<td>Debugging Example</td>
<td>714</td>
</tr>
<tr>
<td>16.7</td>
<td>Concluding Remarks</td>
<td>716</td>
</tr>
</tbody>
</table>
During the software development process, there are many opportunities for errors or human mistakes to be made in its various development phases. In the early phases, there exists some automated tool support for the detection of errors; however, several of the techniques used are still manual. Although these techniques detect and eliminate many early errors, the remaining errors are reflected in the code. This results in code that has errors because of specification and design errors. Also, additional errors usually are introduced during coding. This chapter is devoted to techniques for finding errors throughout the development and implementation phases.

The testing of software is concerned with uncovering errors. This activity can be done separately or in conjunction with the coding activity. The activity is not a small task, as it is widely acknowledged that approximately 40% of total elapsed development time and over 50% of total resources can be expended on testing. In some safety-critical systems, such as those developed for air-traffic and nuclear-reactor applications, testing costs can be several times the total cost of the other development phases.

Testing is never easy; Edsger Dijkstra has written that whereas testing effectively shows the presence of errors, it can never show their absence. The success of uncovering errors during testing depends on the test cases used to exercise the code. The main focus of this chapter is to examine test-generation techniques (methods) and their associated criteria for selecting test cases.

In addition to the design and generation of test cases, the testing phase consists of other tasks, including test planning, test execution, and the collection and evaluation of test results. All of these tasks are incorporated in a software testing strategy. There are two general strategies for testing: bottom-up and top-down. In a bottom-up strategy, we first test each module, component, or class in isolation. Then the components are repeatedly combined and tested as subsystems, with the main objective of uncovering interface errors, until all components are included in the system. On the completion of unit and integration testing, the entire system is tested to ensure that it behaves as required by its users. A top-down approach is in many ways the reverse of bottom-up. The top-level component is first tested by itself. Then all components that collaborate directly with tested component(s) are combined and tested until all components of the entire system are incorporated.

16.1 Fundamentals of Software Testing

16.1.1 Basic Terminology

Testing terminology has evolved over the last quarter century. Numerous papers and books have been written by scores of authors; the resulting terminology has often been confusing and at times inconsistent. For example, a well-known author on testing uses the term “bug” to mean many things depending on the context. According to that author, a “bug” can be a mistake in the interpretation of a specific requirement in the requirements specification for a particular problem, a design error in going from requirements (what the system is to do) to design (how the system does it), a syntax error in some code fragment, or the cause of some system crash. In this chapter, we will use the terminology from the standards developed by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society [26].

Software developers make errors or mistakes that result in software systems containing defects or faults. For example, a misunderstood requirement in the requirements specification invariably results in a design that does not do what the users want. Errors tend to propagate and be magnified in the downstream phases of the development process.
A fault is the manifestation or form of an error in an artifact produced. The artifact can be a narrative text document, interaction diagram, object diagram, or code. Faults can occur in both software and hardware. The terms “defect” and “bug” have also been used instead of faults. Although the synonym “bug” has been used frequently in the past instead of software fault, its use is now discouraged because calling a fault a bug seems to imply that the fault somehow wandered into the software from somewhere and that software developers are powerless to control it.

We can classify faults as those of omission or commission. When a software developer makes an error of omission, the resulting fault is one where something is missing in the artifact— for example, a fault of omission may occur when an assignment statement that performs some computation is not in the code. Faults of omission are very important. Studies have found them to be the most common type of fault in certain application areas. By contrast, when a software developer makes an error of commission, the resulting fault is one with something incorrect in the artifact—for example, an assignment statement that performs the wrong calculation in the code. Faults of omission are more difficult to detect and resolve than those of commission.

A failure is the inability of a system to perform as required. Although the system may behave as specified in the requirements, it may not perform as required. Failures can be detected in any phase of software development or during operation and maintenance.

A fault can lead to zero or more failures. Observing a failure implies the existence of a fault in the system. However, the presence of a fault does not imply the occurrence of a failure. For example, if a code fragment containing a fault is never executed, the fault will never cause that fragment to fail. In summary, an error can lead to a fault and a fault in turn can lead to a failure.

Failures are observed during the testing phase. These failures indicate the presence of faults, which are identified in a separate “debugging” activity. If failures are not observed during some period of time, this does not imply the absence of faults. Failures only imply the presence of faults, which makes it difficult to decide when to terminate the testing process.

In this chapter and in the remainder of the book, we will use the terms “error” and its manifestation, “fault,” interchangeably. Unless necessary, we do not distinguish between the two terms.

### 16.1.2 Basic Notions of Testing

Recall from Section 15.1 that there are two broad classes of techniques that can be used to verify a program. One class of techniques is based on providing a formal proof for a program. The proof is usually deduced directly from the program. Formal proofs tend to be long and complicated, even for short programs. Furthermore, a proof can easily be wrong. Finally, proof techniques do not take into consideration a programming language’s environment, such as hardware, operating system, and users.

The second class of verification techniques is based on testing. Testing is the activity of executing a program with the purpose of finding errors. A program is executed for a set of input values, which is called a test point or test case. The program’s observed outputs for each test case are used to find and repair errors. A test is a finite collection of test cases. Testing, as opposed to proving, does yield concrete information about a program’s behavior, through its execution in a real environment. Testing by itself, however, is usually not sufficient to verify a program. This is because to completely verify a program would require (for any but the simplest programs) an astronomical number of test cases. In practice, only a small subset of possible test cases is used. A test case may result in exposing an error, but does not show the absence of errors, except in the case of trivial programs.
The documentation of a test case should include, in addition to the set of inputs identified by some testing method, a list of expected outputs, any circumstances that hold before a test case executes, and a list of actual outputs. A test oracle generates the list of expected outputs for a test case. There are two main types of test oracles: automated and human. An automated oracle that always produces the expected outputs is obviously preferable. In many situations, however, the oracle is a person who determines, mostly by hand, the expected outputs of a program for some test case. Since humans are prone to making errors, the outputs produced by a human oracle must also be verified. This makes testing tedious, expensive, and time consuming. Throughout the remainder of the chapter, we assume the existence of a test oracle that computes the expected outputs for all test cases in a test.

In particular, note that human oracles often use the specifications of a program to determine its expected behavior. Consequently, it is important to have specifications against which the software is tested. However, it should be noted that using specifications to generate expected outputs may produce outputs that are different from the required outputs.

16.1.3 Determining Test Cases

The most important aspect of testing deals with the determination of a finite set of test cases that will expose as many errors as possible. Several testing techniques or methods exist that assist testers in identifying and choosing test cases. There are two broad approaches to identifying test cases. These approaches are known as white-box (structural) testing and black-box (functional) testing. In white-box (also called clear-box, program-based, or logic-driven) testing, the tester uses the internal structure of the program in the formulation of test cases. In black-box (also called specification-based, data-driven, or input-driven) testing, a program is viewed as a black box or function that transforms the inputs (the domain) to outputs (the range). The details of how the transformation is done is not used in designing the test cases.

In white-box testing, one obvious approach, in attempting to exhaustively (or completely) test a program, is to cause every statement in the given program to execute at least once. As we shall see later, however, this approach is greatly inadequate. Another approach is to execute, by generating a suitable set of test cases, all possible control paths through a program. By a control path, we mean a sequence of statements that can be executed as a result of the branching of conditional and loop statements. The number of distinct control paths in most programs tends to be extremely large. The testing of all such paths is quite often impractical. Even if we could test all possible paths, there are several reasons that the path approach to testing is not sufficient to completely test a program. For one, an exhaustive path test is not the same as checking the specifications for the program. For example, a program might perform a sort instead of a search. Unless we have the program’s specifications, exhaustive path testing will not detect a program that correctly does a task but it is the wrong task. Another reason is that the program may contain several missing paths. Path testing does not expose the absence of necessary paths. Furthermore, an inappropriate or incorrect decision may not be exposed when a program is path-tested. Finally, an incorrect calculation may not be detected when the appropriate path is tested.

In black-box testing, the tester views the program as a black box whose internal structure is unknown. Test cases are generated solely from the specifications of the program and not its internal structure. Such test cases are usually extended to include invalid data so that the program can detect erroneous inputs and output appropriate error messages. A program that handles invalid as well as valid input data is said to be robust.
The exhaustive testing of a program by black-box testing requires exhaustive input testing. This requires that all possible inputs be included in a set of test cases. Also, the verification of a program requires that a program be tested for what it is not supposed to do as well as what it is supposed to do. Thus, a tester must include test cases that cover not only all valid inputs, but also all invalid inputs. Consequently, even for small programs, exhaustive testing would require a tester to essentially produce an infinite number of test cases. Therefore, exhaustive testing is not practical.

The testing of large programs, such as a Java or Eiffel compiler, is even more difficult. Such compilers must detect valid as well as invalid programs. The number of such programs is clearly infinite. Programs such as operating systems, database systems, and banking systems have memory. The result of one test case depends on the results of previous test cases. Therefore, exhaustive sequences of test cases must be devised to test such programs.

From the previous discussion, we see that exhaustive input testing is not practical or possible. Therefore, we cannot guarantee that all errors in a program will be found. However, since it is not economically feasible to exhaustively test a given program, we want to maximize the number of errors found by a finite set of test cases.

In summary, white-box and black-box testing strategies complement each other. The use of black-box tests, augmented by white-box tests, may in many situations be the best way to find errors.

Since the testing of a program is always incomplete—that is, we can never guarantee that all errors have been found—a reasonable objective is to find as many errors as possible. Thus, it is desirable to choose test cases that have a high degree of success in exposing errors. Clearly, because testing is an expensive activity, we want to find as many errors as possible with as few test cases as possible. Specific techniques for test-case generation will be discussed throughout this chapter.

16.1.4 Levels of Testing

Another important aspect of testing concerns the various testing levels that may be encountered during the testing process. Recall from Chapter 1, that the waterfall model is an early example of a software development life cycle. Although this model has drawbacks, as pointed out earlier in the text, it is convenient here to use the model to identify distinct levels of testing. Using a “V” diagram, Figure 16.1 illustrates the waterfall model and the correspondence between development and testing levels. The development levels consisting of specification, architectural (system) design, and detailed design correspond to the system, integration, and unit testing levels, respectively.

These three levels of testing can be done with different approaches such as bottom-up, top-down, or some combination of the two. In particular, unit testing can involve both white-box and black-box testing methods. Traditional white-box testing methods have been much more appropriate at the unit level than at the integration and system levels. However, it should be emphasized that newer white-box methods have recently become available that can be used in the two higher levels, especially for testing object-oriented software. Although black-box methods are used at the unit level, their use is also relevant at the integration level and, especially, at the system level.

16.1.5 Psychology of Testing

Testing can be done from several viewpoints. One of its most important viewpoints, however, concerns issues of human psychology and economics. Considerations such as having the
A proper attitude toward testing and the feasibility of completely testing a program appear to be as important as, or maybe even more important than, purely technical issues.

Since people are usually goal-oriented, it is vitally important from a psychological viewpoint that a proper goal for testing be established. If the stated goal in testing is to show that a program contains no errors, individuals will probably select test data that do not expose errors in the program. One reason for this is that such a goal is impossible or infeasible to achieve for all but the most trivial programs. Alternatively, if the stated goal is to show that a program has errors, the test data will usually have a higher probability of exposing errors. The latter approach will make the program more reliable than the former.

Earlier we defined testing as the activity of executing a program with the purpose of finding errors. This definition implies that testing is not a constructive process, but a destructive process. Since most people view things in a constructive positive manner rather than a destructive negative manner, this explains why many people find testing to be a difficult task. In fact, the proper testing of a program is often more difficult than its design.

It is unfortunate that in many instances testing activities are not seriously considered until software has been shown not to work properly. In designing and implementing a system, it is easy to become convinced that the solution is correct and therefore extensive testing is unwarranted. This attitude is based on viewing software development as a creative, innovative, challenging, constructive, and optimistic activity.

Thus, a proper testing attitude is to view a test case as being successful if it exposes or detects a new error. In contrast, a test case is unsuccessful if it fails to find a new error. Observe that the meaning of the words “successful” and “unsuccessful” in testing is the opposite of the common usage of these words by project managers and programmers. As a result, the testing attitude is opposite to the design and programming attitude, since testing involves trying to destroy what a programmer has built. For this reason, it is difficult for a programmer who has created a program to then attempt to destroy it. Consequently, many programmers cannot effectively test their own programs, because they do not have the necessary mental attitude of wanting to find errors. Therefore, the verification of large systems is now often performed by independent testing teams.

Figure 16.1. Levels of testing in the waterfall model
16.1.6 Testing Principles

The following is a list of testing guidelines that is important in fostering a proper attitude to testing a program:

1. A program is assumed to contain errors.
2. Testing is performed so that errors are exposed.
3. Each test case should have its associated expected result.
4. A test case is considered to be successful if it exposes a new error; otherwise, it is unsuccessful.
5. Test cases should include both valid and invalid data.
6. Programmers should not test their own work.
7. The number of new errors still to be found in a program is directly proportional to the number already found.
8. The results of each test case must be examined carefully so that errors are not overlooked.
9. All test cases should be kept for possible future use.

16.2 Human Testing

This section briefly describes two approaches to using humans to find faults. The first has the developer read and mentally trace the code looking for faults. This works best after the developer has worked on some other task for a period of time. The second approach uses a group of people.

16.2.1 Code Readings

In the early years of computing, it was generally believed that the only way to test a program was to execute it on a computer. In recent years, however, the reading of documents, diagrams, and programs by people has proved to be an effective way to find errors. Human-based testing methods should be applied before program testing on computers. Although the approach of critical artifact reading is simple and informal, it can contribute significantly to the productivity and reliability of a system. First, it can be done early in the development stages, and errors found at an early stage generally are cheaper and easier to correct than those found later. Also, at the computer-based testing stage, programmers are under greater stress than at the human-testing stage. The result is that more mistakes are usually made in attempting to correct an error at the former stage than at the latter. In addition, certain types of errors are more easily found by human testing than by computer-based testing techniques.

Although this section is called code readings, more than just code can be read. In particular, specification documents, analysis documents and diagrams, and design documents can be read. They need to be read for completeness, special cases, and overall quality. If the developer is doing the reading, there should be at least a couple of days between work on an artifact and critically reading it.
Obviously, the code is read for logical correctness and clarity. In addition, there are some common errors that should be guarded against. We now present a short checklist of common errors in Java code—a partial list of things to look for when reading code:

1. Check for different identifiers with similar names. Such identifiers (e.g., `total` and `totals`) are a frequent source of errors. Also, 1, I, and l are often confused. Also, because Java is case-sensitive, great care must be taken in entering identifiers.

2. Check that every entity is initialized with the proper value. If the default initialization is used, check that it is the proper value.

3. Check each loop for off-by-one errors. Pay special attention to what happens the first time and last time through the loop, and if the loop is never executed.

4. Check boolean expressions for correctness. Many errors are made in using logical operators `||`, `&&`, and `!`.

### 16.2.2 Structured Walk-Throughs

Another approach to human testing of designs and programs is the structured walk-through approach. Here, one person, often the developer, leads a group of people through the design/program. The leader attempts to explain and justify the design. The members of the group point out flaws in the design and suggest methods to improve or simplify the design.

In preparation for a walk-through, the leader must view the system from a different perspective—a high-level view that is simple enough to explain to others. As a result, the leader may see simplifications or cases that were missed when working at the detailed development level.

This approach to finding faults is particularly relevant for the system design level of development, before moving on to detailed design. Here, the leader presents his or her view on the overall organization of the system. The idea is to have a simple overall structure that is comprehensive enough to include all the required functionality. If the leader cannot present a simple enough view for others to follow, that is a sign of difficulties to come and the need for a better design. Also, this is the time for better ideas to be suggested by the group, as useful ideas can easily be accommodated before detailed design.

### 16.3 Black-Box Testing

Our first broad approach to deriving test cases for a program, black-box testing, is to ignore its structure and concentrate only on its specifications. Thus, this section focuses on specification-based testing techniques. Throughout the discussion, we assume that each test case is considered to be independent of every other test case. In effect, this assumption means that no values are saved in any variables from one test case to the next one. In particular, no state changes are brought forward from one test case to the next one (i.e., the program is said to not have memory).

It has already been stated that an exhaustive black-box test of a program is usually impossible. Consequently, we want to devise a small subset of all the possible test cases that will expose as many errors as possible.

The specifications for a program consist of a description of the possible inputs and the corresponding correct outputs of the program. Recall that in giving these specifications we
are concerned with what a program does, not how it does it. Therefore, the program can be viewed as if it were a black box (i.e., we cannot see inside it) that maps possible inputs into possible outputs, as shown in Figure 16.2.

This section examines two black-box testing methods. The first method, boundary value testing, is the simplest, and probably the oldest, testing method. The second method, equivalence class testing, partitions a program’s possible input space into disjoint subsets in an attempt to achieve some form of complete testing while avoiding redundancy. Other black-box methods—in particular, functional testing methods based on decision tables—are not covered.

### 16.3.1 Boundary Value Testing

Historically, testers have observed that, for a given input-to-output mapping, more errors occur at or near the boundaries of the input domain rather than in the “middle” of the domain. These observations have led testers to develop boundary value analysis techniques to select test cases that exercise input values at or near the boundaries of input variables.

Consider some mapping (function) that has an input variable with the interval of values

\[ a \leq x \leq b, \]

where the boundary values for \( x \) are \( a \) and \( b \). One basic boundary value analysis approach is to select test values for an input variable, such as \( x \) above, as follows: \( a, a + \epsilon \), nominal, \( b - \epsilon \), and \( b \), where “nominal” represents some “middle” or typical value within \( x \)’s range, and \( \epsilon \) denotes some small deviation (e.g., one for ranges of integer values).

This basic approach can be generalized in two ways: by the number of variables and by the types of ranges. Generalizing the approach to deal with more than one variable can be straightforward. Consider, as an example, a mapping that involves two input variables with the following ranges:

\[ a \leq x \leq b \]

\[ c \leq y \leq d. \]

A generalization of the boundary value analysis approach to handling this example is easy if we assume that failures are seldom the result of simultaneous faults in the input variables. In other words, we assume that the two variables are independent. Such an assumption is called a single-fault assumption. What this means for testing in the current example of two variables is to hold one of them at its nominal or “middle” value and let the other variable assume its set of five values described earlier. Using this approach with \( x \) as the first variable and \( y \) as the second variable yields the set of cases

\[ \{(x_*, c), (x_*, c + \epsilon), (x_*, y_*), (x_*, d - \epsilon), (a, y_*), (a + \epsilon, y_*), (x_*, y_*), (b - \epsilon, y_*), (b, y_*)\}, \]

where \( x_* \) and \( y_* \) denote the nominal values for \( x \) and \( y \), respectively. Note that this test-generation approach yields nine distinct test cases, since the test case \((x_*, y_*)\) occurs twice in
the enumeration. More generally, for a mapping of $n$ variables, the generalization approach generates $4n + 1$ test cases.

Boundary value analysis produces good test cases when the program to be tested is the implementation of a mapping with independent input variables that denote physical bounded values. In this case, the generalization of the test cases can be done mechanically, without any consideration being given to the nature of the mapping or the meaning of each input variable.

The boundary values of a variable that represent some physical attribute, such as speed, weight, height, pressure, and temperature, can be of vital importance. For example, pressure or temperature values beyond some maximum value may be extremely important. By contrast, nonphysical attributes, such as student numbers, telephone numbers, and credit card numbers, are not likely to have test cases based on their boundary values, which successfully detect faults.

A simple extension to boundary value analysis involves the incorporation of two additional test values for each variable: the first value slightly less than a variable’s minimum value and a second value that slightly exceeds the variable’s maximum value. This results in the following seven test cases for a mapping having one input variable, such as $a \leq x \leq b$:

$$a - \epsilon, a, a + \epsilon, \text{ nominal, } b - \epsilon, b, b + \epsilon.$$ 

This extension of the basic approach is called robustness testing.

Robustness testing can also be extended to generate test cases for several independent variables that represent physical attributes with the same limitations of the basic approach. The main advantage of robustness testing is the focus it places on exception handling. The approach is also applicable to a mapping’s output variables. Attempting to force output variables to have values outside their permitted ranges can lead to the detection of interesting faults.

Recall that the basic approach assumes that the input variables are independent. If this is not the case, however, the Cartesian product of each set of five or seven values for each variable should be used as test cases. For example, if our mapping involves two input variables $x$ and $y$, the Cartesian product of two sets, each consisting of five values, yields $5^2$ or 25 test cases. With robustness testing, the approach yields $7^2$ or 49 test cases. This type of boundary analysis testing method is called worst-case testing. The number of test cases when using this approach increases dramatically with the number of variables. Of course, some of the test cases may not be very effective at detecting faults. However, worst-case testing is useful in testing a mapping that has several input variables which model physical attributes when system failures can be catastrophic or costly.

A familiar variation of boundary value analysis involves special-value testing, where a tester’s experience with similar software, domain knowledge, and specific information about trouble spots is used to create test cases. The success of this approach depends exclusively on the abilities of the tester. The approach is highly subjective, but in many instances it can lead to more effective test cases than basic black-box testing.

In this section, we have investigated boundary value analysis to devise test cases based on a program’s input variables. However, the approach can also be applied to a program’s output variables. We can design test cases for programs that generate a variety of error messages. Test cases that exercise both valid and invalid error messages are desirable. Finally, it is important to keep in mind that, because of its simplicity and the fact that it usually assumes independent input variables, the basic boundary value analysis testing approach may generate poor test cases.
16.3.2 Equivalence Class Testing

For an int variable in some program, it might be possible to test the project when every program int value is input for the variable. This is true because, on any specific machine, only a finite number of values can be assigned to an int variable. However, the number of values is large, and the testing would be very time consuming and not likely worthwhile. The number of possible values is much larger for variables of type float or String. Thus, for almost every program, it is impossible to test all possible input values.

To get around the impossibility of testing for every possible input value, the possible input values for a variable are normally divided into categories, usually called blocks or equivalence classes. The objective is to put values into the same equivalence class if the project should have similar (equivalent) behavior for each value of the equivalence class. Now, rather than testing the project for all possible input values, the project is tested for an input value from each equivalent class. The rationale for defining an equivalence class is as follows: If one test case for a particular equivalence class exposes an error, all other test cases in that equivalence class will likely expose the same error.

Using standard notation from discrete mathematics, the objective is to partition the input values for each variable, where a partition is defined as follows:

Definition 16.1: A partition of a set \( A \) is the division of the set into subsets \( A_i, i = 1, 2, \ldots, m \), called blocks or equivalence classes, such that each element of \( A \) is in exactly one of the equivalence classes.

Often the behavior of a program is a function of the relative values of several variables. In this case, it is necessary for the partition to reflect the values of all the variables involved.

As an example, consider the following informal specification of a program:

Given the three sides of a triangle as integers \( x \), \( y \), and \( z \), it is desired to have a program to determine the type of the triangle: equilateral, isosceles, or scalene.

The behavior (i.e., output) of the program depends on the values of the three integers. However, as previously remarked, it is infeasible to try all possible combinations of the possible integer values.

Traditional equivalence class testing simply partitions the input values into valid and nonvalid values, with one equivalence class for valid values and another for each type of invalid values. Note that this implies an individual test case to cover each invalid equivalence class. The rationale for doing this is that if invalid inputs can contain multiple errors, the detection of one error may result in other error checks not being made.

For the triangle example, there are several types of invalid values. The constraints can be divided into the following categories:

\( C_1 \). The values of \( x \), \( y \), and \( z \) are integers.

\( C_2 \). Each input contains exactly three values: \( x \), \( y \), and \( z \).

\( C_3 \). The values of \( x \), \( y \), and \( z \) are greater than zero.

\( C_4 \). The length of the longest side is less than the sum of the lengths of the other two sides.

Of these categories, the first two categories are more difficult to handle. If a decimal value or a string value is entered when an integer value is needed, it will normally give rise to an exception resulting in the termination of the program. If the program is to be crash-proof,
each line of input should be read as a string and processed in a manner to handle every possible input. For our analysis here, we will not take this approach, and we ignore the possibility of such input values. Also category two, where the wrong number of inputs is provided, is not easily handled and will be ignored.

Although the third category refers to all three input variables, they are really independent situations. To guarantee that each invalid situation is checked independently, an invalid equivalence class should be set up for each of the variables having a nonpositive value:

1. \{ (x, y, z) \mid x \leq 0, y, z > 0 \}
2. \{ (x, y, z) \mid y \leq 0, x, z > 0 \}
3. \{ (x, y, z) \mid z \leq 0, x, y > 0 \}

Note that each of the three sets is very large, but each triple in the same set corresponds to the same invalid situation.

For the fourth category, the relative sizes of the values are important. However, each of the variables can be the one that has the largest value (i.e., corresponds to the longest side). Thus, three more invalid equivalence classes are needed:

4. \{ (x, y, z) \mid x \geq y, x \geq z, x \geq y + z \}
5. \{ (x, y, z) \mid y \geq x, y \geq z, y \geq x + z \}
6. \{ (x, y, z) \mid z \geq x, z \geq y, z \geq x + y \}

Although traditional black-box testing collects all valid data sets into one equivalence class, this is often not desirable. When the program behaves differently for different valid data values, these data values should be partitioned into equivalence classes that do behave similarly. In the triangle example, the program behaves differently (produces different output) for each of the three different types of triangles. Recall that a valid triangle is called **equilateral** if all three sides is equal, **isosceles** if exactly one pair of sides are equal, and **scalene** if none of the sides are equal. This yields five valid equivalence classes for valid data — one for equilateral; three for isosceles, depending on which pairs of sides are equal; and one for scalene:

7. \{ (x, y, z) \mid x = y = z \}
8. \{ (x, y, z) \mid x = y, z \neq x \}
9. \{ (x, y, z) \mid y = z, x \neq y \}
10. \{ (x, y, z) \mid x = z, y \neq x \}
11. \{ (x, y, z) \mid x \neq y, y \neq z, x \neq z \}

Thus, we have 11 equivalence classes that represent 11 different scenarios. For each of these equivalence classes, we need to generate at least one test case to ensure that the project handles the data values from the equivalence class correctly. In formulating the test cases, it is desirable to analyze the boundary of each equivalence class. Thus, rather than selecting any test case in an equivalence class as a representative, we should select test cases that are on the class boundary or at least “close” to the boundary.

In the current example, possible test cases for each equivalence class are the following:

1. (−1, 2, 3), (0, 2, 3)
2. (2, −1, 3), (2, 0, 3)
3. (2, 3, −1), (2, 3, 0)
4. (5, 2, 3), (5, 1, 2)
5. (2, 5, 3), (1, 5, 2)
6. (2, 3, 5), (1, 2, 5)
In some cases, two test cases are included for an equivalence class to have a boundary case and a case just beyond the boundary.

As a second example, consider the task of testing a procedure to sort three integer values \(x, y, \) and \(z\). Obviously, it is impractical to test all possible input values, so the equivalence class approach should be used. The equivalence classes should reflect the different relationships that can exist between the sizes of the values. Thus, they should include the cases where each value is the smallest of the values, the largest, and in the middle. As there are 3! or 6 permutations of three values, six equivalence classes are required. The most difficult part of obtaining the equivalence classes is ensuring that they form a partition (i.e., each triple occurs in only one equivalence class). Thus, each equivalence class must have a constraint that is disjoint from the others. The following equivalence classes are one possibility:

\[
\begin{align*}
&x \leq y \leq z \quad x \leq z < y \quad y < x \leq z \\
&y \leq z < x \quad z < x \leq y \quad z < y < x
\end{align*}
\]

Each equivalence class was designed to be disjoint from the previous classes. For example, the fourth class is disjoint from the previous three as it requires \(x\) to be larger than \(z\). The last class is also disjoint from the first three as it also requires \(x\) to be larger than \(z\). It is disjoint from the fourth as the last class requires \(z\) to be less than \(y\). Finally, the last class is disjoint from the fifth as it requires \(y\) to be less than \(x\).

Usually, the interesting cases are the nonequality cases. If they are separated from the equality ones, the expanded partition becomes

\[
\begin{array}{cccccccc}
&x \leq y \leq z & x \leq z < y & y < x \leq z & y \leq z < x & z < x \leq y & z < y < x \\
\hline
x = y = z & x = z < y & y < x = z & y = z < x & z = x = y & z < x = y & z < y < x \\
\hline
x = y < z & x < z < y & y < x < z & y < z < x & z < x < y \\
x < y = z \\
x < y < z
\end{array}
\]

Each equivalence class is listed below the class from which it came. This collection of test cases should show any faults in the procedure. However, such an exhaustive collection of test cases is only possible when there are few values to sort, as even the smaller partition above has \(n!\) test cases, where \(n\) is the number of values to be sorted.

In the triangle and sorting examples, the interaction of the variables was quite obvious. This is not always the case, so a distinction is made between weak and strong equivalence class testing. As a motivating example, assume that a program implements a mapping of three variables \(x, y, \) and \(z,\) wherein their corresponding input variable domains are \(X, Y, \) and \(Z,\) respectively. Further, assume that each domain is partitioned by a suitable equivalence relation such that

- \(X\) is partitioned into equivalence classes \(X_1, X_2,\) and \(X_3.\)
- \(Y\) is partitioned into equivalence classes \(Y_1\) and \(Y_2.\)
- \(Z\) is partitioned into equivalence classes \(Z_1, Z_2, Z_3,\) and \(Z_4.\)

Now choose \(x_i, y_i,\) and \(z_i\) to be representative elements from the equivalence classes \(X_i, Y_i,\) and \(Z_i,\) respectively. Thus, there are three, two, and four representatives that cover \(X, Y,\)
Table 16.1. Weak Equivalence Class Testing

<table>
<thead>
<tr>
<th>Test Case</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_1$</td>
<td>$y_1$</td>
<td>$z_1$</td>
</tr>
<tr>
<td>2</td>
<td>$x_2$</td>
<td>$y_2$</td>
<td>$z_2$</td>
</tr>
<tr>
<td>3</td>
<td>$x_3$</td>
<td>$y_1$</td>
<td>$z_3$</td>
</tr>
<tr>
<td>4</td>
<td>$x_1$</td>
<td>$y_2$</td>
<td>$z_4$</td>
</tr>
</tbody>
</table>

and $Z$, respectively. A set of test cases for this program is given in Table 16.1. The number of test cases in the table is determined by the equivalence relation with the largest number of equivalence classes (i.e., equivalence relation for $Z$ having four classes). Note that the test cases have been identified in a systematic way, and it assumes that the input variables are independent of each other. This form of testing is called weak equivalence class testing.

A stronger form of equivalence class testing is based on the Cartesian product of the blocks induced by each equivalence relation (i.e., each representative for $X$ is combined with each representative for $Y$ and $Z$). In the current example, a table of $3 \times 2 \times 4 = 24$ test cases would result. This form of testing is called strong equivalence class testing. This approach covers all equivalence classes, and there is a test case for each combination of inputs. Therefore, there is a “completeness” aspect to this approach.

We conclude the discussion of equivalence class testing with the following observations:

- For programs whose inputs are described by ranges and sets of discrete values, equivalence class testing can be effective in detecting faults.
- The success of equivalence class testing depends on the identification of appropriate equivalence classes. Sometimes the equivalence classes are obvious, yet at other times the appropriate classes may be elusive.
- Weak equivalence class testing does not take into consideration any interdependencies among input variables.
- Boundary value analysis is complementary to equivalence class testing.

Problems 16.3

1. Prove that for a mapping of $n$ independent input variables $4n + 1$ test cases are required.

2. For a function of $n$ variables, obtain a formula that gives the number of robust test cases.

3. Extend the triangle problem specification so that right triangles are also recognized. A right triangle with sides $x$, $y$, and $z$ satisfies $z^2 = x^2 + y^2$. Revise the set of equivalence classes and associated test cases for this extended triangle problem.

4. Let $\text{Tomorrow}$ be a function of three variables: $\text{month}$, $\text{day}$, and $\text{year}$. It returns the date of the next day that follows the given input date. The $\text{month}$, $\text{day}$, and $\text{year}$ variables have the following ranges:

   $1 \leq \text{month} \leq 12$
   $1 \leq \text{day} \leq 31$
   $1801 \leq \text{year} \leq 2001$

Using a combination of equivalence-partitioning and boundary-value approaches:
/** The purpose of this method is to demonstrate program-based testing. */
int testExample(int x, int y, int z)
{
    if ((x > 10) | (y != 0))
        z = 1 + z / x;
    if ((x != 10) & (z >= 2))
        z = z - 1;
    return z;
}

Figure 16.3. Simple method to illustrate white-box testing

(a) Identify equivalence classes for this problem.
(b) Derive a set of test cases based on the equivalence classes found in part (a).

5. Let Yesterday be a function similar to that of Tomorrow in the preceding problem. It returns the date of the “day before” the given input date. Using the same ranges and testing approaches from the preceding problem, obtain the following:
   (a) Identify equivalence classes for this problem.
   (b) Derive a set of test cases based on the equivalence classes found in part (a).

6. The real roots of a quadratic equation \( ax^2 + bx + c = 0 \) are given by the formula
   \[
   x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},
   \]
   where \( b^2 \geq 4ac \); \( a, b, \) and \( c \) are real values; and \( a \neq 0 \). For this problem:
   (a) Develop an appropriate set of input conditions.
   (b) On the basis of the input conditions in part (a), identify the equivalence classes.
   (c) Formulate the test cases to cover as many equivalence classes as possible.

7. Given the three sides of a triangle \( x, y, \) and \( z \), it is desired to compute the area of the triangle by the function
   \[
   \text{Area} = \sqrt{p(p-x)(p-y)(p-z)},
   \]
   where \( p = (x + y + z)/2 \). The lengths of the sides are to be positive integers.
   Using a combination of equivalence-partitioning and boundary-value approaches:
   (a) Identify equivalence classes for this problem.
   (b) Devise a set of test cases based on the equivalence classes of the problem.

16.4 White-Box (Program-Based) Testing

It is desirable to derive test cases using specification-based (black-box) techniques and obtain additional test cases by applying program-based (white-box) techniques. The basic idea of white-box testing is to look at the code and design test cases to test specific parts of the code. Ideally, the collection of all white-box test cases would test all possible executions of the code. However, for most programs, this would entail too many test cases. Instead, on the basis of the program structure, we want to devise test cases that will “exercise” or “cover” the various parts of a program.

The simplest criterion for covering a program, called statement coverage testing, is to require that each statement/instruction in a program be executed at least once. As an
example of this approach, consider the simple method in Figure 16.3. Graphically, this method can be represented by a control-flow graph as shown in Figure 16.4. The two diamond-shaped icons represent decision nodes, with T indicating the true branch and F indicating the false branch. Rectangular icons denote assignment statements. The directed lines that connect nodes indicate the various possible branches/paths that can be taken in executing this program. Several of the lines are labeled with a single lowercase letter for path identification purposes. Observe that every statement in the program can be executed by using a single test case. The test case $x = 20$, $y = 5$, and $z = 20$ causes the statements on path $abe$ to be executed.

The statement coverage criterion is very weak. For example, the logical operation in the first statement could have been & instead of |. Such an error would not be found. As another example, if the condition in the second statement should have read $z <= 2$ rather than $z >= 2$, this error would also not be detected. Furthermore, if the value of $z$ was supposed to be changed when path $acd$ was followed, such an error would not be exposed because there are no statements at all on this path. In this example, statement coverage testing did not generate a test case to cover this path. Clearly, the statement coverage approach is virtually useless in this example, because it fails to expose many possible errors.

A stronger criterion for testing success is branch testing or decision testing. Using this criterion, each branch of each decision in a program must be traversed at least once. The decision statements include if statements, switch statements, and the various loop statements.

Figure 16.4. A sample function and its path structure
For the path structure of Figure 16.4, branch testing can be realized by using two test cases:

\[
\begin{align*}
  x &= 20 \quad y = 5 \quad z = 20 \quad (\text{path } abe) \\
  x &= 10 \quad y = 0 \quad z = 10 \quad (\text{path } acd)
\end{align*}
\]

Alternatively, two other test cases can cover \textit{ace} and \textit{abd}.

Branch testing success usually implies that the statement covering criterion has also been satisfied. However, there are exceptions. For example, if there are no branching decisions in a program, no statement will be executed.

This approach can be used to obtain a complete set of test cases that tests all branches. First, a complete set of execution paths required for branch testing is obtained directly from the program. Next, for each execution path, a set of constraints on the input data values must be obtained that implies that the path will be executed. The constraints are obtained by tracing the path backwards in the program to determine the logical conditions that must be true as a result of the statements that have been done. Finally, values are selected that fit the constraints.

For example, suppose that we wish to test the path \textit{abe}. Starting at the last condition, since the second \textbf{if} condition must be true to test edge \textit{e}, we must have

\[
x \neq 10 \land z \geq 2
\]

when it is tested. As path \it b is taken, the value of \it z for this condition is given by \(1 + z / x\), where the value for \it z in this expression is the original value of \it z. Thus, in terms of the original value for \it z, the condition to cause branch \textit{e} of the second \textbf{if} to be executed is

\[
x \neq 10 \land (1 + z / x) \geq 2
\]

The first \textbf{if} condition must also be true. Putting them together yields

\[
(x \neq 10 \land (1 + z / x) \geq 2) \land (x > 10 \lor y \neq 0)
\]

\[
= (x > 10 \lor (1 + z / x) \geq 2 \land x > 10) \lor (x \neq 10 \land (1 + z / x) \geq 2 \land y \neq 0)
\]

\[
= (x > 10 \lor (1 + z / x) \geq 2) \lor (x \neq 10 \land y \neq 0 \land (1 + z / x) \geq 2)
\]

The expression \(1 + z / x \geq 2\) is equivalent to

\[
(x > 0 \land z \geq x) \lor (x < 0 \land z \leq x)
\]

Putting this condition into the preceding one, we obtain

\[
(x > 10 \lor z \geq x) \lor (x \neq 10 \land y \neq 0 \land x > 0 \land z \geq x)
\]

\[
\lor (y \neq 0 \land x < 0 \land z \leq x)
\]

From this expression, it is easy to obtain input values that cause the path \textit{abe} to be executed. This can also be repeated for the other paths to cover all the branches. Note that in some cases it might be impossible to execute a path; such paths are called \textit{infeasible}. In such a case, the condition simplifies to the value \textbf{false}, and no test case can be generated for it.

Although branch testing is superior to statement coverage, branch testing is still inadequate. For example, if an error in the decision of the second \textbf{if} statement caused it to be

\[
x \neq 10 \land z \leq 2
\]

instead of

\[
x \neq 10 \land z \geq 2,
\]
this error would go undetected if the two previous test cases were used. This occurs because the difference between \((z \leq 2)\) and \((z \geq 2)\) is not tested by either test case (for the first test case, both \(z \geq 2\) and \(z \leq 2\) are true, and for the second test case, the condition is false, since \(x = 10\) is independent of the value of \(z\)).

On the other hand,

\[
\begin{align*}
x &= 20 & y &= 0 & z &= 20 & \text{(path } abc) \\
x &= 10 & y &= 5 & z &= 0 & \text{(path } abd) \\
\end{align*}
\]

tests both conditions in each if statement, but fails to achieve branch coverage because the false branch of the first if statement, containing edge \(c\), is not followed. To completely check the logic of a program, all branches and all possible outcomes of multiple conditions in each conditional construct in the program must be tested.

A more thorough approach to testing a program is to ensure that every distinct path through a program is executed at least once. The distinct paths of the example in Figure 16.4 are \(abe\), \(ace\), \(abd\), and \(acd\). The following test cases ensure path coverage for the current example:

\[
\begin{align*}
x &= 20 & y &= 5 & z &= 20 & \text{(path } abc) \\
x &= 5 & y &= 0 & z &= 10 & \text{(path } ace) \\
x &= 10 & y &= 5 & z &= 0 & \text{(path } abd) \\
x &= 10 & y &= 0 & z &= 10 & \text{(path } acd) \\
\end{align*}
\]

In general, the number of distinct paths in a program may be very large. As an example, consider the graphic representation in Figure 16.5 of a simple program with one loop that is executed 10 times. There are four possible paths within the body of the loop. Since the loop has 10 repetitions, there are \(4^{10}\) or approximately 1 million distinct execution paths within this loop. However, if the loop is executed 20 times, there are over a trillion distinct execution paths. Clearly, it is not possible to test all these paths. This is true for most programs, so branch testing or a variation of it is usually used.

Additional testing techniques based on control-flow and data-flow graph representations will be discussed in Section 19.8.5. Basis path testing derives a set of linear independent basis paths from a program’s control flow graph. Data-flow testing involves first obtaining a graph from a program based on where variables are defined and subsequently used. Test cases can then be formulated from these defined uses paths.

Testing the assertions of the precondition and the postcondition of a method should also be considered. These assertions are part of the definition (in an ADT sense) of a method, so it might make sense to test them during black-box testing. However, the type of testing generated for the assertions is generally code specific, so their analysis often fits better under white-box testing. Also, note that the precondition and postcondition implicitly include the invariant of the class containing the method.

If a method has a precondition, test cases should be considered that attempt to violate each branch of its assertions. Also, this is an appropriate time to determine any cases that the method can handle, but are precluded by the precondition. For the postcondition, test cases should be considered that attempt to violate the postcondition. Such test cases should all be infeasible, but it can be worthwhile to attempt to generate them. Further, in considering such cases, it is a good idea to see whether the postcondition can be strengthened to catch more faults.

If the method has loops, for each loop tests should be developed so that the method is tested when the loop body is not executed, is done exactly once, and is done many times. If the loop condition is a compound condition, there should be tests to check each part.
### Sec. 16.4. White-Box (Program-Based) Testing

#### Problems 16.4

1. Given the method

   ```
   int someMethod(int a, int b) {
   int c = 0;
   if (a > 10 && b == 5) {
       c = 1;
   }
   if (a == 5 || b > 7) {
       c += 2;
   }
   return c;
   }
   ```

   devise suitable test cases to ensure
   
   (a) statement coverage
   (b) branch coverage
   (c) path coverage
2. Given the method

```java
/** This method contains an infeasible path. */
void processGrade(int grade)
{
    if (grade < 49)
        System.out.println("fail");
    else
        System.out.println("pass");
    if (grade >= 80)
        System.out.println(" with honors");
}
```

(a) Construct a control flow graph for this method.
(b) Identify the infeasible path (i.e., the path that cannot be executed).
(c) Devise test cases to give statement, branch, and path coverage.

### 16.5 Object-Oriented Testing

Since most object-oriented development approaches, including the one proposed in Section 12.4, usually do not follow a waterfall-like development cycle such as the one shown in Figure 16.1, there appears to be some confusion concerning the levels in object-oriented testing. Recall from Section 16.1 that in traditional testing there are three levels of testing: unit, integration, and system. In traditional testing, the smallest unit of testing is the smallest compilable unit, such as a procedure, function, or module. In an object-oriented context, the unit to be tested is usually a class. This results from the encapsulation of attributes and methods (procedures or functions) in the definition of classes. At the overall system level, object-oriented development does not appear to affect testing. Since testing at this stage is based on black-box techniques that use the requirements specification, system testing should be independent of the way a system is developed.

However, integration testing in an object-oriented context is a different matter. Since in this context we do not usually have a decomposition structure, which is always available when following a functional approach to software development, some other approach must be used. Because objects communicate by sending and receiving messages, composition can replace decomposition. A major difference between object-oriented and traditional development is that the former uses composition whereas the latter uses decomposition.

#### 16.5.1 Issues in Testing Object-Oriented Software

In traditional unit testing, the unit under test is a procedure or function. Although in testing object-oriented software we also test individual methods (procedures or functions) within a class, we are primarily concerned with the unit testing of classes. Since methods in a class may collaborate to deliver some accessible services provided by a class, we need to test how these methods work together with class attributes to provide such services. This type of testing is called integration testing. Although we refer to testing a class as unit testing, it is really a combination of traditional unit testing and integration testing.

Class testing involves testing the methods of an object with the goal of exposing errors in the methods’ implementation or the object’s state. Since an object’s state is a function of its methods, the key is to test all its methods. Each method of a class, when the class represents a good abstraction, will perform a single cohesive operation. The specification of a method is defined by
• The method’s name

• The method’s parameters (type and order).

Since such a method directly corresponds to the traditional unit of testing, black-box, using its specification, and white-box testing are applicable to testing class methods. This is certainly true for a method that does not call any other method. The only complication is that the result of the method may depend on the state (field values) of the object. Thus, we assume the existence of a “testing oracle” that has access to the state information of the object.

Class testing also involves testing methods that interact. As we have seen, there are two kinds of interaction: intraclass and interclass. An intraclass interaction occurs when a method calls another method within the same class. Such interaction involves little or no flow of data, since both methods access the object’s fields. An interclass interaction, by contrast, involves the sending of a message from one instance of a class to another instance of some (likely different) class. The testing of intraclass interaction is usually less tedious than the testing of interclass interaction.

If the method called, whether intraclass or interclass, is already implemented and tested, then there are no complications. Otherwise the method is often replaced by a dummy method or stub. Frequently, the stub or dummy method is simply a “shell” or skeleton that does very little work. In particular, a stub might simply return a particular value, output a message that indicates that the method was invoked as expected, or provide for entering the result in an interactive and debugging mode.

In summary, new issues that arise in object-oriented testing that are not present in traditional testing include the following:

• Classes are tested indirectly by testing its instances (i.e., its objects).

• Objects tend to be small, but interfacing is more complicated.

• Requirements specified in a requirements document are not likely expressed in terms of objects and methods.

• The state of an object may influence an execution path, and a class’ methods may communicate through this state.

• The effect of structural inheritance (class and interface) and dynamic binding may complicate testing.

Several of the new issues make the testing of object-oriented software more difficult.

### 16.5.2 Method Testing

As just noted, we test a class indirectly by testing its instances (i.e., objects). Testing the behavior of an object requires some knowledge about its internal structure or state. The state of an object was introduced in Section 3.3.2 as the values of the object’s fields. Section 3.3.3 introduced three main categories of operations/methods that can be performed on an object: constructor, accessor, and modifier. A constructor creates an object and initializes that object to some initial state. An accessor is an operation that queries an object’s state without changing it. A modifier is a state-changing operation.

In this subsection, we look at testing an individual method of a class. In the next subsection, the discussion will consider sequences of methods.
There is a fundamental problem that often occurs in testing a method. To know whether the result of applying the method to a specific object is correct, it is necessary to know the resulting state of the object. To know the state of the object, it is necessary to use access methods. But how does one know the access methods, which definitely depend on the unknown state, are correct? If the state is determined by the values of a small number of fields, the state can easily be determined from the values of these fields. However, the state often depends on the field values in some complex fashion. In this case, it is not easy to determine the state of the object from the field values. The problem is further compounded for black-box testing, as the whole object is regarded as a black box, and even its fields are unknown. In this case, rather than testing an individual method, it is necessary to test the method in combination with an appropriate query method or methods. If the expected answers are obtained by applying the query method(s) to the object after applying the method to be tested, it is assumed that the method is correct. Once some methods have been tested, they can be used in the verification of other methods.

As an example, consider the black-box testing of the \texttt{insert()} method for an ordered binary tree. Before the \texttt{insert()} method can be applied, the tree object has to be created. Thus, the constructor is needed, as is a query method to determine the current state of the object. The \texttt{toString()} method is designed to form a \texttt{String} representation of the object, so it is the natural query method to use. The result is the standard means to test the method—create the object via the constructor, check the state using \texttt{toString()}, and do a number of \texttt{insert()} operations using the \texttt{toString()} method to check the state of the tree after every insertion.

We still need to decide what values to insert into the tree. Random values could be used, but a better approach is to use the boundary-value and equivalence-class techniques. One boundary is determined by the size of the tree. Thus, the first three test cases are as follows:

1. Insertion into an empty tree
2. Insertion into a size one tree
3. Insertion into a tree of size greater than 1

Since these insertions are being done into an ordered binary tree, the items should be kept in lexical order; this ordering should be checked. The method \texttt{toString()} yields a list of the items in order, so that it can be used as the query method. The selection of values to be used can be done by the boundary-value approach. Therefore, five further test cases are the following:

4. Insertion of a new minimum value
5. Insertion of a new second smallest value
6. Insertion of a new maximum value
7. Insertion of a new second largest value
8. Insertion of a new value that is approximately the median value

Note that we have implicitly used equivalence classes to specify these test cases. All new values that are smaller than the values already in the tree are equivalent in terms of inserting a new minimum value. Thus, this is one equivalence class. If there are \( n \) values in the tree,
/** Insert x into the tree.
 Analysis : Time = O(h) worst case, where h = height of the tree */
 public void insert(I x)
 {
   if (isEmpty())
     rootNode = new BinaryNodeUos<>(x);
   else if (x.compareTo(rootItem()) < 0)
   {
     OrderedSimpleTreeUos<I> leftTree = rootLeftSubtree();
     leftTree.insert(x);
     setRootLeftSubtree(leftTree);
   }
   else
   {
     OrderedSimpleTreeUos<I> rightTree = rootRightSubtree();
     rightTree.insert(x);
     setRootRightSubtree(rightTree);
   }
 }

Figure 16.6. Method insert() from OrderedSimpleTreeUos

then any new value fits into one of \( n + 1 \) equivalence classes according to where it fits in the ordering of the values. The preceding test cases do not test all equivalence classes, only the ones specified by boundary value testing.

These tests should be sufficient to show that the items are being kept in order. But since the items are to be kept in an ordered binary tree, we should check that the items are in a binary tree. Other than using query method toString(), there is no way of doing this. However, the LinkedSimpleTreeUos class has function toStringByLevel(), which shows the complete structure of the tree. Thus, to see the tree structure, this method should be used.

The boundary value tests for the correct tree structure overlap with previous test cases: insert into an empty tree, insert into a one item tree, insert leftmost in the tree (a new minimum value), insert rightmost in the tree (a new maximum value), and insert in the middle of the tree (a new value near the median). The fact that a new value is always inserted at the leaf level suggests some new equivalence-class test cases:

9. Insertion to the left of a leaf node
10. Insertion to the right of a leaf node
11. Insertion to the left of a node with only a right subtree
12. Insertion to the right of a node with only a left subtree

Note that most of these cases might be covered by the previous cases. However, the way to ensure that none of these cases is omitted is to have a special test case for each one.

All these test cases have been developed without looking at the implementation of the insert() method. To complete the development of test cases, we should look at the code to develop white-box test cases. The method for insert() is given in Figure 16.6. The method has three cases, so it would seem that at least three test cases are needed just to do statement coverage. However, the method is recursive, so two of the cases call the method again. Hence, in a general tree, it is possible to design one test case that does both left and right branching before inserting at the leaf level:
13. Insert a value in a location that involves both left and right branching

Thus, statement coverage is easily achieved. This last test case also does branch coverage, since there is at least one statement in every branch of the code. Path coverage is harder to achieve. To cover all possible paths through the code, it would be necessary to test all possible paths in all possible binary trees. Clearly, this is impossible to do. Hence, the tester will need to be content to test a variety of different paths. This should be accomplished by the foregoing test cases, so no more cases are added.

16.5.3 Testing Recursive Methods

We have made liberal use of recursive methods in the discussions of various data structures. Although any recursive algorithm can be rewritten into an equivalent iterative counterpart with loops, the approach used in testing the former is different from that used in testing the latter.

Recursive methods often fail when

- Incorrect argument values are used in the main or recursive calls.
- A base case is not detected or reached.
- A method that recursively traverses a data structure may not traverse the entire structure.
- The method’s time performance may be unsuitable for certain performance-sensitive applications.
- Recursion depth causes the run-time system to fail to provide the memory needed in the form of activation records for the method to continue its recursive descent.

Any recursive method definition should include one or more base cases and one or more recursive cases. Recall that the precondition and postcondition of a method implicitly include the class invariant.

A test suite for a recursive method should include at least one test case for each of the following (some of which have been discussed before):

- Base case(s)—involving no recursion.
- Single recursive call case(s)—involving a depth of one recursion.
- Many recursive calls case—recursion to a large depth, maybe even the greatest allowable or feasible level that makes sense.
- Boundary values defined for the method’s parameters or states of data structures traversed by the method.
- Worst-case time/space performance for the method to a maximum-reasonable depth of recursion.

The \texttt{insert()} example of this subsection is recursive, so let us see what test cases are implied by the preceding specification. Testing the basic case without recursion is the same as insertion into an empty tree (i.e., test case 1). There are two ways to reach the base case after one recursive call. They are insertion immediately to the left of the root and insertion immediately to the right of the root. They are special cases of test cases 1, 9, and 10, but should be added to the list to ensure that they are both included. Accordingly, we have
/** The number of leaves in a tree.
 Analysis: Time = O(size of the tree) */

int numberOfLeaves()
{
    if (isEmpty())
        return 0;
    if (rootLeftSubtree.isEmpty() && rootRightSubtree.isEmpty())
        return 1;
    return rootLeftSubtree.numberOfLeaves() + rootRightSubtree.numberOfLeaves();
}

Figure 16.7. Function numberOfLeaves()
2. Design a set of test cases to test search() of class OrderedSimpleTreeUos on page 486.

3. Design a set of test cases to test toString() of class BinaryNodeUos on page 463.

4. Design a set of test cases to test toStringByLevel(int i) of class LinkedSimpleTreeUos on page 467.

5. Design a set of test cases to test deleteItem() of class OrderedSimpleTreeUos on page 486.

16.5.4 State-Based Class Testing

The application of a sequence of modifier operations to an object in some initial state will result in a corresponding sequence of new states. The state-based approach to testing emphasizes the state changes as a sequence of operations is done.

As an example, consider a simple-bounded counter object whose values vary from some initial value, say 0, to some maximum value denoted by upper. In addition to a counter constructor, the following three methods can be performed on an object:

- inc() — increments the counter by 1
- dec() — decrements the counter by 1
- value() — returns the contents of the counter

Methods inc() and dec() are modifiers that change an object’s state. The value() method does not change an object’s state. The state space of a counter object consists of the integers between 0 and upper, inclusive. The state of a counter is recorded in the single field ctr.

The equivalence class testing approach can be used to reduce the state space. The general approach is to first examine the state space of a field for specific state values (e.g., boundary values). The values of interest are those that are handled differently in some way. The specific field values for a class are obtained by examining the specification and code for that class. Second, the state values for the field are grouped together so that all values in a group are handled in the same manner. Consequently, there is no need to distinguish between them when performing state-based testing. Generally, this results in the specific state values being placed in different groups. Each group forms an equivalence class for testing purposes.

As an example of using this equivalence partitioning approach, consider the bounded-counter example again. The specific values of interest are when a counter has values of 0 and upper. They are each in their own groups. A group is also formed for all the intermediate values

\[0 < \text{ctr} < \text{upper}\]

as each of these values behaves in much the same way. Thus, the three states for a bounded counter are as follows:

1: \(\text{ctr} = 0\)
2: \(\text{ctr} = \text{upper}\)
3: \(0 < \text{ctr} < \text{upper}\)

Now that the original state space has been reduced to a manageable size, we can determine the effect of applying each operation on each of these three possible states. Given an object’s current state, the application of an operation to this state produces a new state. This new state can be the same as the current state or a different state.
Table 16.2. Transition Matrix for a Bounded Counter

<table>
<thead>
<tr>
<th>Current State</th>
<th>Method</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>value</td>
<td>inc</td>
<td>dec</td>
</tr>
<tr>
<td>1: ctr = 0</td>
<td>1</td>
<td>2</td>
<td>error</td>
</tr>
<tr>
<td>2: 0 &lt; ctr &lt; upper</td>
<td>2</td>
<td>{2, 3}</td>
<td>{1, 2}</td>
</tr>
<tr>
<td>3: ctr = upper</td>
<td>3</td>
<td>error</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 16.2 contains a *transition matrix* for the bounded-counter example. We assume that a counter object has been created by a constructor that initializes the counter to contain a value of 0. The leftmost column in the table denotes the current state of the object. Each of the three remaining columns denotes the new state produced when applying a particular operation to an object’s current state. For example, method *value()* does not change an object’s current state. However, the two remaining methods do change the object’s current state. An *inc()* method changes the object’s state from state 1 to state 2. The application of an *inc()* method to an object in state 2 either leaves the state unchanged or changes the object’s state to state 3. When an *inc()* method is applied to an object in state 3, an illegal state denoted by *error* occurs. Since the counter is already at its maximum allowed value, the operation cannot be performed. Usually, a precondition stating that the object’s value must be less than *upper* is specified in the code for operation *inc()*.

The column for the *dec()* method is interpreted in the same manner. Trying to decrement a counter while it is in state 1 results in an illegal or *error* state. Decrementing a counter in state 2 results in a transition to either state 2 or state 1. Finally, decrementing a counter in state 3 changes the state to state 2.

A minimum test coverage for an object is *all state coverage*. To attain such coverage, operation sequences should cause an object to make transitions through all allowable states. State coverage is analogous to statement coverage in traditional unit-level testing. Coverage of all transitions from one state to another is a more acceptable standard. From the transition matrix for the bounded counter, we can generate a test case for each of the transitions. In general, however, this approach may not give adequate test coverage. Also, it can be difficult to obtain a suitable number of equivalence classes.

As a more complicated example of state-based testing, consider a class representing accounts in a banking application, such as the one discussed in Chapter 12. Let us assume that the class has the instance variables

```java
String owner;
int accID;
float balance;
```

and the associated methods are *open()*, *deposit()*, *withdraw()*, and *close()*. The state space of an object is the Cartesian product of the three fields *owner*, *accID*, and *balance*. This state space is huge. We can use the specific value and general state group approach to reduce the domain of each of the three fields and, in so doing, reduce the overall state space of an account object. The specific values of interest for the instance variables of the *Account* class are:

```java
owner = null;
accID = 0;
balance = 0.0;
```

For each of the fields, the other (nonspecific) values can be grouped as follows:
Using the specific values and the groups, we obtain a Cartesian product consisting of 18 triples of the form \((x, y, z)\), where 

\[
\begin{align*}
ox & \in \{/D2/D9/D0/D0, \text{nonnull}\}, \\
y & \in \{\text{accID} < 0, \text{accID} = 0, \text{accID} > 0\}, \\
z & \in \{\text{balance} < 0.0, \text{balance} = 0.0, \text{balance} > 0.0\}.
\end{align*}
\]

The original state space has been reduced to a manageable size.

A state transition diagram is a graphic representation of an object transition matrix. Such a diagram (see Figure 16.8) consists of a set of nodes or vertices where each node represents a state, or equivalence class of states, of an object. Each state node is identified by a unique label. The directed arcs leading from one state to another indicate the state transitions. Each label immediately above or below an arc denotes the operation that causes that transition. In Figure 16.8, the state diagram contains five states: start, emptyAccount, normalAccount, nullAccount, and closedAccount. The directed arcs with the labels open, deposit, withdraw, and close denote the transitions that are allowed on an open account object. An open() operation creates an empty account with a nonnull owner, a positive account number, and a zero balance. An initial deposit operation produces a normal account with a positive balance. For a normal account, we can perform many queries, deposits, or withdrawals. A final withdrawal operation would cause a normal account to become a null account with a balance of zero. A close operation on a null account results in a closed account. We have ignored showing the invalid or illegal states that may arise owing to bad data, such as a negative account number or a null owner. Also, we have kept it simple by assuming that there is no line of credit and the balance is always nonnegative. The start and closedAccount states are special states to start and end the diagram. Although the emptyAccount and nullAccount states will have the same attribute values (i.e., they correspond to the same triple), two distinct states are used, as they represent different situations in the evolution of an account. The operations for querying the field values of an object have been omitted from the state transition diagram.

The focus in state-based testing is on designing appropriate sequences of operations to exercise all the states of a class. A sequence of operations (separated by commas) on an object is read from left to right, so that, for example,
open(), deposit(), withdraw(), close()

is the minimum test sequence of operations that exercises all the states. A better test sequence is one that exercises all the transitions. There are many other valid operation sequences. A compact description of all valid sequences is given by the expression

open(), deposit(), ExtraMethods, withdraw(), close()

The term ExtraMethods, which denotes possible operation sequences, is defined by

(deposit() | withdraw() | balance() | owner | accID)*

where | separates alternative operations that can be placed in the sequence and * denotes 0 or more operations selected for the sequence. The following are valid test sequences:

open(), deposit(), owner, accID, balance(), withdraw(), close()
open(), deposit(), withdraw(), deposit(), deposit(), withdraw(), close()

The first sequence exercises the query operations that do not change state, other than those in the minimum test sequences given earlier. The second sequence exercises the state changing operations.

State-based testing can also be used to check the construction of dynamic data structures, such as stacks, queues, and trees. We can analyze a data structure with respect to its possible significant structural changes. The result of the analysis is a set of situations called data scenarios, which can be used to test objects. These data scenarios are essentially additional specific values of interest for some field. For example, consider a data structure such as a bounded queue that can contain a maximum of \( n \) elements. Data scenarios that are important in the correct construction of such a data structure would include the following:

- An empty queue
- A queue of one element
- A partially filled queue
- A full queue containing \( n \) elements

These scenarios can be used to formulate additional test cases to those obtained by applying the specific value and general state group approach to reducing the state space of an object.

It is quite possible that, in attempting to reduce an object’s state space, there will still be too many states and transitions to test. In such cases, state-based testing becomes impractical. An alternate and more practical approach is to use the notion of identity transitions. An identity transition consists of a sequence of method invocations that begin and end at the same state. For example, in the bank account class introduced earlier, consider the two sequences (columns) of method invocations:

\[
\begin{align*}
\text{b1} = \text{a.balance()} \\
\text{a.deposit}(100) & \quad \text{b1} = \text{a.balance()} \\
\text{a.withdraw}(100) & \quad \text{a.deposit}(100) \\
\text{b2} = \text{a.balance()} & \quad \text{a.withdraw}(200) \\
\text{b2} := \text{a.balance()} & \\
\end{align*}
\]

where \( \text{b1} \) and \( \text{b2} \) denote a begin and end balance for account \( \text{a} \), respectively. The values of \( \text{b1} \) and \( \text{b2} \) in each sequence should be identical; otherwise a fault has been detected. Many such sequences of method invocations exist.
The idea of identity transitions comes from the axiomatic description of ADTs in Chapter 6. For example, in the axiomatic specification of a stack in Section 6.3.2, the axiom

\[ s\text{.push}(x)\text{.pop} = s \]

was used to express the behavior that pushing \( x \) on a stack followed by the popping of the stack results in an identical stack \( s \).

An advantage of this approach to testing is that the individual states and transitions associated with each method in the sequence need not be explicitly specified. Another advantage is that transitions can be checked by an automated tool.

For an object whose state consists of more than one attribute, testing each individual state is impractical. State-based testing of a class is feasible only if the state space can be reduced in such a way as to still get good test coverage.

Problems 16.5.4

1. Obtain a state transition table for an unbounded positive counter with an initial value of 1 and the following methods: \texttt{value()}, \texttt{inc1()}, and \texttt{dec1()}.

2. Consider a hospital patient system where a patient is admitted initially to the hospital. Once admitted, the patient is assigned to a bed within a specified ward. An operation, which is performed by a surgeon, is scheduled for the patient in some operating theater. The operation is performed on the patient in the theater. After the operation, the patient goes to a recovery ward for some period of time. After recovery, the patient is returned to her or his assigned bed. After full recovery, the patient is discharged from the hospital. Obtain a state transition table for the patient.

3. (a) Assume that, in the object-oriented modeling of a university student registration application, a student entity is represented by a class. Based on your own experience as a student, formulate a state diagram representation for a student object for the registration system to be developed. The state diagram should show what you consider possible states and associated transitions from the time you were admitted to the university to the time you leave the university.

   (b) Based on your state diagram in part (a), formulate test cases for a student object.

16.5.5 The Effect of Inheritance on Testing

When a subclass inherits from a base class, some methods may need testing, whereas others may not. The complete testing of each class individually and separately in a given inheritance taxonomy can be wasteful. Yet some methods may need to be retested. One reason for having to retest code is that a subclass may affect inherited fields. In an attempt to reduce testing effort, it becomes necessary to identify which methods need to be tested and which methods need not be retested. The overall goal here is to inherit some of the test cases from a subclass’ ancestor. Such an approach to testing a subclass is sometimes referred to as \textit{incrementing testing}, because we inherit some test cases from its ancestor and create new test cases for that subclass.

There is one case, sometimes called \textit{strict inheritance}, wherein the addition of a new subclass does not require the retesting of inherited members from the superclass. In this case, a subclass contains new members (fields or methods) that do not interact with inherited members. As an example, consider the simple taxonomy of Figure 16.9 where class \( B \) inherits from the superclass \( A \). Class \( B \), in addition to inheriting instance variable \( x \) and method \( f() \) from \( A \), has a new instance variable \( y \) and a new method \( g() \). If \( g() \) does not interact with
the inherited members \( x \) and \( f() \), we can reuse the test cases for \( A \) and devise new test cases that test members \( y \) and \( g() \) in \( B \).

However, assume that methods \( f() \) and \( g() \) interact; for example, with \( f() \) and \( g() \) setting \( x \) to 2.5 and 5.5, respectively. Without the retesting of \( f() \) in \( A \), the conflicting assumptions about \( x \) would not be detected.

A more complex situation arises in the case of nonstrict inheritance, where, for example, some method may be overridden. In overriding an inherited method, the overridden method’s signature remains the same, but the implementation of the overridden method can be much different from its inherited counterpart. White-box-derived test cases for the overridden method are usually quite different from those of the ancestor method. For example, two implementations for sorting items could be very different with respect to their code structures. It is also likely that the overridden method will behave differently from its inherited counterpart. Therefore, the black-box test cases for the original and overridden methods will likely be different. New black-box test cases must be created for the overridden method.

As another example, consider the simple inheritance taxonomy consisting of three classes in Figure 16.10, where \( A \) has methods \( m1() \) and \( m2() \), \( B \) overrides method \( m1() \), and \( C \) overrides method \( m2() \). Assume that in \( A \), method \( m1() \) invokes \( m2() \); in \( B \), \( m1() \) invokes \( m2() \) from \( A \); and in \( C \), method \( m1() \) from \( B \) invokes \( m2() \) from \( C \). We need different test cases for method \( m1() \) in each of the classes \( A \), \( B \), and \( C \). Since the inherited method \( m1() \) from \( A \) is overridden in \( B \), it is clear that different sets of test cases are needed for classes \( A \) and \( B \). With respect to \( B \) and \( C \), observe that method \( m1() \) is the same in \( B \) and \( C \). However, \( m1() \) in \( B \) invokes \( m2() \) from \( A \), and \( m1() \) in \( C \) invokes \( m2() \) from \( C \). Since the interaction of \( m1() \) in \( B \) is different from the interaction of \( m1() \) in \( C \), different test cases for \( m1() \) are required for classes \( B \) and \( C \).

### 16.5.6 Object-Oriented Integration Testing

Because object-oriented software development emphasizes composition rather than traditional functional decomposition, traditional integration testing strategies have little relevance in testing object-oriented software. The overall algorithm in object-oriented software is distributed among several methods in collaborating classes/objects; however, in a functionally decomposed system, the structure of the code directly reflects the overall structure.
of the algorithm. Also, an object-oriented system consists of collaboration packages (sub-systems) each of which contains classes and packages as components.

One strategy for integration testing object-oriented software, use-based testing, uses a layered approach. The independent classes in the system are tested first; these classes have little or no dependence on other supplier classes. The next layer of classes to be tested, called dependent classes, are those that are clients of the independent classes. Successive layers of dependent classes are tested until all components of the system are present.

Another integration testing approach, thread-based testing, is based on the notion of a thread (not to be confused with Java’s Thread capabilities). A thread is a sequence of instructions that might be consecutively executed. They are not necessarily consecutive statements of a method. Instead, they might be in several methods of several classes, but there is a set of data values that will cause them to be executed consecutively. In this approach to testing, we select and execute threads. Threads exist at different levels (e.g., at unit, integration, and system levels). At these levels, threads correspond to their respective level behavior. Several views of threads include the following:

- The execution of a sequence of source statements
- The sequence of activities involved with a specific instance of a use case (scenario)
- A sequence of transitions in a state transition diagram
- An interleaved sequence of input and output events

Several other views of threads could be given.

The analysis and design models for a system are useful in devising test cases at the integration level. These models reveal the nature of the collaborations, dependencies, behavior,
and communication specifics that are used in developing the system and should hold in
the resulting system. In particular, use cases and the various UML diagrams for a system
can be used as a basis for generating test cases. Such tests tend to exercise more than
one subsystem within the system; their main goal is to check the behavior of interacting
subsystems. Incorrect or incomplete specifications can be detected by these tests.

It is convenient to view a system as having a port boundary, which is the location of
system-level inputs and outputs. A port is defined as a point where an input/output device
is attached to a system. Examples of port devices include a display screen on a terminal and
a printer. If a test case requires an input or an output that is not visible at a port boundary,
the test case is not considered to be a system-level test case. Such a test is considered to be
an integration-level test.

16.5.7 Object-Oriented System Testing

At the system level, testers of object-oriented software focus on the use of the system by
its various actors. Test cases at this level should be based on the use cases that have been
formulated for the system. Such test cases are likely to uncover errors in the system’s user
interaction requirements. Black-box testing techniques are also applicable to formulating
test cases for object-oriented software. Additional test cases can be derived from the system’s
context model and the analysis model of the problem domain. The testing at this stage
should concentrate on what the system is supposed to do, not on what the software does.

Object-oriented systems are often viewed as being event driven. Each port device or
port is where system events occur. We associate an input system event with an input port
and an output system event with an output port. Events at system ports are visible to the
users of the system. By observing sequences of port events, users gain an understanding
of system behavior. This idea was used by Jorgensen [29] to define an Atomic System
Function (ASF). An ASF consists of an input port event and a collection of threads. Each
thread of an ASF starts with the ASF’s input port event and follows a sequence of method
invocations until it is terminated at an output port event. Several threads are possible for
one input event as the reaction of the system can depend on the data input. An ASF thread
is the simplest thread at the system level as a thread that does not involve the I/O ports
is considered to be at the unit (class) or integration level. In general, we want to formulate
test cases that reflect ASFs.

Test coverage, as a minimum, should test every thread for every ASF. To find all ASFs,
for each port the events need to be identified that can occur at that port. A more adequate
coverage involves having threads for the occurrence of common sequences of input port
events (i.e., sequences of ASFs). Furthermore, threads can be identified, in some contexts,
for inappropriate input events. Such threads are used by testers who attempt to cause the
system to fail.

Systems with interesting object models can lead to additional threads. For example,
in the student registration system in Section 14.4, there are relationships among object
instances of Student, College, Course, CourseSection, and Instructor. Various trans-
actions that are performed by the registration system include the following:

- Enrolling a student in a course section
- Deleting a student from a course section
- Assigning an instructor to a course section
- Adding a course registration to a course section
• Adding an instructor to a department’s roster

Such transactions are the main threads of the system. Often these transactions involve abnormal or illegal situations. For example, an enroll transaction may try to enroll a student in a course section that is full. A course drop transaction may attempt to delete the registration of a student in some course section for which the student was never registered. These examples and many more can be obtained from the problem domain object model (see Figure 14.31). Relationship instances often lead to the identification of system-level threads.

As was done in event-based testing, we can seek threads in terms of data-based coverage measures. Test cases can be designed to check the cardinality of each relationship, participation in each relationship, and functional dependencies among relationships. Recall from Section 3.2 that relationships can be one-to-one, one-to-many, many-to-one, and many-to-many. A student can enroll in no more than some maximum number of courses, say six. A course section may have some minimum enrollment to be given. Relationship participation deals with whether every object of a class must participate in a relationship. For example, every student must belong to a college and every instructor must belong to a department. A course section must be taught by at least one instructor. An example of a functional dependency among relationships is that a student that does not belong to a college might not be allowed to register in courses offered by departments within that college. These cardinality and participation relationships should be tested in the system developed.

16.6 Locating and Repairing Dynamic Faults

Two important factors in producing high-quality software are correctness and robustness (see Section 1.3). As we have seen in Section 15.2, with some work Java’s exception handling supports the Design-by-Contract approach to developing systems. Robustness is supported by Java’s exception mechanism, which is described in Section 15.3.

So far, this chapter has discussed test planning and test case development, which is an organized approach that tries to expose the presence of faults in a given program. We now focus on the other aspects of testing: running test cases and evaluating test results.

In the discussion of Design by Contract in the preceding chapter, we saw that the components of a contract consist of typed arguments associated with method calls and assertions. In a strongly typed language like Java, any fault in a method’s types of parameters is detected by the compiler. Programs that fail because of this type of fault are not allowed to execute.

The assertion component of a contract, however, cannot be checked so easily. Unlike typed method parameters, assertions, such as preconditions, postconditions, and class invariants, can only be checked dynamically at run time because they involve values computed during a program’s execution.

Once the presence of a fault has been established, its nature and location must be determined first and the fault must be fixed or repaired. This process has been called debugging. This section describes various approaches to debugging. We progress from brute-force debugging approaches to more sophisticated ones.

16.6.1 Planning for Debugging

We can prepare for debugging by first assuming that the program will contain faults. Therefore, the design and programming phases should view the easy detection of faults as an important objective.
Recall from Section 2.14.3 that a common approach is to insert diagnostic printouts throughout a program from the start. Frequently such printouts are made conditional on special input debugging flags. It is far easier to insert these special print statements when the program is first written than at a later time. Conditional print statements often affect the structure (e.g., nesting and alignment) of a program. The placing of diagnostic printouts is easy at the initial writing of the program, since the details of its strong and weak points are fresh in the programmer’s mind, and it is usually a simple matter to insert a few additional statements. After the program is written, the programmer may forget some of the details, thereby making the insertion of appropriate printout statements more difficult.

16.6.2 Debugging by Brute Force

A popular method of program debugging is the brute-force approach. The popularity of this rather inefficient method is perhaps due to the fact that it is the least mentally taxing of the various approaches. This method is often time consuming and not very successful in medium-sized and larger systems.

The brute-force approach to debugging includes the following strategies:

1. Scattering print statements randomly throughout the program
2. Using automated debugging tools

The technique of scattering print statements throughout a program displays the contents of selected variables and facilitates the study of the program’s dynamics. Some drawbacks of this technique include the following:

1. It can be a hit-or-miss approach.
2. It can be used on small or large programs, but the cost of using this technique can be prohibitive.
3. Its use can produce large volumes of data.

The technique of using an automated debugging aid is another brute-force approach that is better than the technique just described. A programmer can analyze the dynamics of his or her program by using a special interactive debugging aid or taking advantage of certain language debugging constructs that are part of the programming language. Examples of such language constructs include those that change the values of specified variables, the printed tracing of procedures, functions, and so on, and printed traces of certain executed statements. Some system environments provide automated debugging aids that offer to a programmer a variety of commands that facilitates the following:

1. The examination and changing of variables
2. The setting and removing of breakpoints
3. A variety of query and search commands
4. A go command

The setting of a breakpoint (or a “stop”) allows a programmer to stop the program during its execution when a specified statement or location is reached. Once a breakpoint has been reached, a programmer can, through the use of the automated aid, examine the contents
of certain variables and make alterations to a program. A go command is used to resume
the execution of a program after a breakpoint has been encountered. Some automated
debugging aids permit the specification of conditional breakpoints and go commands. For
example, a stop at a break point may occur only after control has passed through a specified
number of times.

The use of an automated debugging aid can produce an excessive amount of data, much
of which is useless. Also, for the most part, this approach ignores the process of thinking.
A fault can be located and repaired by first carefully analyzing the clues, or symptoms, and
then piecing together the details. We now turn to strategies based on this kind of thinking.

16.6.3 Debugging by Backtracking

Thus far, we have discussed two brute-force approaches to debugging that require little
thinking. We now examine a debugging method that requires some degree of thinking. Al-
though this technique is of a general nature, it can be effective in locating faults, particularly
in small programs.

The basic approach to backtracking is to start at the point in the program where a failure
has been observed from displayed or printed results. At this point, we can deduce from the
observed output the values of the program’s variables—that is, the program’s state. From
this point, we can mentally back up the execution of the program to a previous state; that
is, to a previous location from the initial point in the program where a new set of values
for the program’s variables is obtained. These new program values are inferred from the
program values at the previously considered (initial) point. By repeating this basic backup
process, we can often pinpoint the fault easily. That is, we can isolate a fault to be between
the last two successive points in the program. At this point in the sequence, the state of the
program, as defined by the program’s variable values, is different from what was expected.
Sometimes it may be necessary to collect more information by running additional test cases.

Rather than backtrack from the point at which the failure was observed, we can instead
track forward from the program’s inputs. We wish to reemphasize that the tracking of faults
(either backward or forward) often works well on small programs. However, if the tracking
of a fault becomes difficult, a more systematic and rigorous approach is required. The next
subsection examines such an approach.

16.6.4 Debugging by Induction

A more formal approach to locating faults is based on the principle of induction. This
approach begins with a set of clues or symptoms of the failure. These clues are then analyzed
to produce a hypothesis of the most probable cause for the failure. If the hypothesis can be
verified—that is, if the hypothesized cause of the fault accounts for all the clues—the fault
is located; otherwise the hypothesis must be changed. The procedure is as follows:
Sec. 16.6. Locating and Repairing Dynamic Faults

Table 16.3. A Table for Organizing Clues

<table>
<thead>
<tr>
<th>Question</th>
<th>Is</th>
<th>Is Not</th>
</tr>
</thead>
<tbody>
<tr>
<td>What</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Where</td>
<td></td>
<td></td>
</tr>
<tr>
<td>When</td>
<td></td>
<td></td>
</tr>
<tr>
<td>To what degree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Collect the relevant data
2. Organize the data
3. Study the relationships
4. Attempt to devise a hypothesis
   If the attempt is not successful, go to step 1
5. Attempt to prove the hypothesis
   If the proof attempt is not successful, go to step 4
6. Repair the fault that has caused failure
7. Verify the repair

We now elaborate on each step of this algorithm.

In the first step, it is important to take into account all available data or symptoms about the problem. Test data indicating correct operation as well as data indicating incorrect operation are collected.

The second step organizes the clues or symptoms given by data so that patterns emerge. This is the inductive step, which permits us to progress from particulars to the general; finding contradictions is an important goal in this step. A useful way to organize and display data is shown in Table 16.3. The “What” question concerns the general symptoms of the fault. The “Where” question concerns the location(s) where the symptoms were observed. The “When” question relates to the times that the symptoms occurred. Finally, the “To what degree” question concerns the scope and magnitude of the symptoms. The “Is” and “Is Not” columns of the table each contain the circumstances and conditions under which the failure occurred or did not occur, respectively. These two columns describe potential contradictions that may lead to a hypothesis concerning the failure. To obtain contradictions, it is sometimes required to use additional test points.

In steps 3 and 4, we study the relationships among the clues so that a hypothesis of most probable cause is postulated. If a hypothesis cannot be formulated, additional data are required. Additional test cases may be required at this point. At any rate, the investigation must continue.

Step 5 attempts to prove the hypothesis. Failure to prove the hypothesis and attempting to repair the fault may result in only a portion of the problem being corrected. The proving of the hypothesis involves verifying that it completely accounts for all clues.

In step 6, the faulty code fragment that caused the failure is repaired.

Step 7 involves rerunning the case at which a fault was exposed to verify that the repair has corrected the observed problem. It may be advisable to run additional test cases to test the repair. Also, it is wise to rerun previously successful test cases to ensure that the repair has not created new problems through a ripple effect. If the repair is not successful, then return to step 1.
16.6.5 Debugging by Deduction

As opposed to debugging by induction, where one arrives at a hypothesis of the most probable cause for a failure by examining a set of clues or symptoms for that failure, debugging by deduction starts with a number of causes and by the process of elimination and refinement arrives at the most probable cause.

The following general algorithm outlines the steps in debugging by deduction:

1. List all possible causes for the observed failure
2. Use the available data to eliminate various causes
3. Refine the remaining hypotheses
4. Prove (or disprove) each remaining hypothesis
5. Repair the fault that caused the failure
6. Verify the repair

We now elaborate on each step of this algorithm.

The first step is to compile a list of all possible causes for the failure. This cause list can then be used to structure and analyze the available information.

The second step involves the analysis of the available data. Finding contradictions is an important goal in this step. Table 16.3 can again be used for this purpose. To obtain contradictions, additional test points may have to be used. In the case where more than one possible cause remains, the most likely cause is used first.

If necessary, the third step involves using the available clues to refine the hypothesis to something more specific.

Steps 4, 5, and 6 are similar to steps 5, 6, and 7 in the debugging by induction approach of the previous subsection.

16.6.6 Debugging Example

We consider now a very simple example. Figure 16.11 presents a Java program to compute the average of a set of marks. The program should then display the average, the number of marks greater or equal to the average, and the number of marks below the average. A fault has been deliberately introduced into this program. Although the fault may be immediately obvious to some (probably not all) readers, we will use the formal techniques for finding it. A programmer must develop these techniques, because not all faults are obvious.

The requirements for this program specified that it was to accept up to 100 integer marks, each of which was in the range 0 through 100 inclusive. The marks are to be supplied in the input stream, followed by a sentinel value of minus one. Table 16.4 presents several test cases suggested by the requirements along with the expected and actual results.

We will apply the method of debugging by induction. The pertinent data have already been presented in Table 16.4. Organizing these data exposes several symptoms of the fault. The average displayed is always less than the correct result. Further, either the number of marks above or the number of marks below the average is one too great. These observations suggest a possible hypothesis: The count of marks is one too high (an off-by-one fault). This would account both for the number above or below being one too large and also for the average being too low. For example, in the first test case, an average computed as \((10 + 40 + 60 + 90)/5\) would give \(200/5 = 40\). However, the actual average is 39.8. We
import java.io.*;

public class FaultExample
{
    public static void main(String[ ] args) throws IOException
    {
        int[ ] marks = new int[100]; // the marks
        int numMarks; // actual number of marks
        int sum = 0; // sum of the marks
        float average; // average of the marks
        int numAbove; // number of marks above average
        int numBelow; // number of marks below average
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        /* Step 1: Read the marks. */
        numMarks = 0;
        do
        {
            marks[numMarks++] = Integer.parseInt(br.readLine());
        } while (marks[numMarks - 1] != -1);
        /* Step 2: Compute and display average. */
        for (int i = 0; i < numMarks; i++)
        {
            sum += marks[i];
            average = (float) sum / numMarks;
        }
        System.out.println("The average mark is " + average + ".");
        /* Step 3: Count and display the number of marks above and below average. */
        numAbove = 0;
        numBelow = 0;
        for (int i = 0; i < numMarks; i++)
        {
            if (marks[i] >= average)
            {
                numAbove++;
            }
            else
            {
                numBelow++;
            }
            System.out.println("There were "+ numAbove + " marks above or the same as "+ "the average.");
            System.out.println("There were "+ numBelow + " marks above or the same as "+ "the average.");
        }
    }
}

Figure 16.11. Erroneous program to compute the average of a set of marks

<table>
<thead>
<tr>
<th>Table 16.4. Expected and Actual Results for Debugging Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Values</td>
</tr>
<tr>
<td>Avg.</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>10 40 60 90</td>
</tr>
<tr>
<td>90 100</td>
</tr>
<tr>
<td>0 10</td>
</tr>
<tr>
<td>50</td>
</tr>
</tbody>
</table>
observe that $5 \times 39.8 = 199$ and conclude that the sum must have been computed incorrectly as well. The combination of the number of marks being one too high and the sum being one too low leads to our final hypothesis: The sentinel value of minus one is included in the computations as an actual mark.

Our final hypothesis can be proven either by tracing the program or by running it with some additional output statements to specifically identify the marks included in the computation of the average. Proof of the hypothesis and repair of the error are left as exercises.

**Problems 16.6**

1. Prove the proposed hypothesis for the example in this section by tracing the program.

2. Repair the fault in the example in this section. Validate the repaired program by running it with a well-chosen set of test cases.

### 16.7 Concluding Remarks

We have emphasized in this chapter that the testing of software is concerned with uncovering errors. This phase can consume a disproportionate percentage of total development resources when compared with other development phases. Testing is never easy; whereas testing shows the presence of errors, it usually never shows their absence.

The fundamental notions of testing were discussed. The IEEE terminology of error, fault, and failure was introduced. Software testers use techniques for generating good test cases whose execution will cause a system to fail because it contains faults. These failures can then be traced back to errors made in the software development process. It is important to understand the psychology of testing. A test case is successful if it exposes or detects an error. Alternatively, a test case is unsuccessful if it fails to find a new error. The main goal in successful testing is to attempt to break or destroy a system. This is the opposite of other developers who develop or create systems. Three levels of testing were identified: unit, integration, and system.

Several approaches to testing were explored. Human testing has developers of a system read and logically trace the code looking for faults. This approach to testing has been found to be successful in industry, with some firms reporting the discovery of as much as half of all errors made during the entire development process.

Two broad approaches to identifying test cases were presented. In black-box testing, the tester views the program as a black box whose internal structure is unknown. Test cases are generated solely from the specification of the program. Two black-box testing techniques were presented: boundary value testing and equivalence class testing. In white-box testing, a tester uses the internal structure of the program in the formulation of suitable test cases. Test cases are designed to “exercise” or “cover” various parts of a program. Coverage criteria, from the weakest to the more desirable, include statement coverage, decision coverage, and some form of path coverage. In general, it is impossible to test all possible paths in a program. Some specific path coverage approaches will be given in Chapter 19.

Several issues in testing object-oriented software were raised. New issues in object-oriented testing that are not present in traditional testing include the following:

- Classes are tested indirectly by testing their instances (i.e., their objects).
- Objects tend to be small, but interfacing is more complicated.
• Requirements specified in a requirements document are not likely expressed in terms of objects and methods.

• The state of an object may influence an execution path, and a class’ methods may communicate through this state.

• The effect of structural inheritance (class and interface) and dynamic binding may complicate testing.

We have seen that several of these issues make the testing of object-oriented software more difficult.

Once the presence of a fault has been established, its nature and location must be detected first and then it must be fixed or repaired. Several approaches to locating and repairing faults were outlined.