8
Testing

In a software development project, errors can be introduced at any stage during development. Though errors are detected after each phase by techniques like inspections, some errors remain undetected. Ultimately, these remaining errors are reflected in the code. Hence, the final code is likely to have some requirements errors and design errors, in addition to errors introduced during the coding activity. To ensure quality of the final delivered software, these defects will have to be removed.

There are two types of approaches for identifying defects in the software—static and dynamic. In static analysis, the code is not executed but is evaluated through some process or some tools for locating defects. Code inspections, which we discussed in the previous chapter, is one static approach. Another is static analysis of code through the use of tools. In dynamic analysis, code is executed, and the execution is used for determining defects. Testing is the most common dynamic technique that is employed. Indeed, testing is the most commonly used technique for detecting defects, and performs a very critical role for ensuring quality.

During testing, the software under test (SUT) is executed with a finite set of test cases, and the behavior of the system for these test cases is evaluated to determine if the system is performing as expected. The basic purpose of testing is to increase the confidence in the functioning of SUT. And as testing is extremely expensive and can consume unlimited amount of effort, an additional practical goal is to achieve the desired confidence as efficiently as possible. Clearly, the effectiveness and efficiency of testing depends critically on the test cases selected. Much of this chapter therefore is devoted to test case selection.

P. Jalote, *A Concise Introduction to Software Engineering*,
DOI: 10.1007/978-1-84800-302-6_8, © Springer-Verlag London Limited 2008

In this chapter we will discuss:

– Basic concepts and definitions relating to testing, like error, fault, failure, test case, test suite, test harness, etc.

– The testing process—how testing is planned and how testing of a unit is done.

– Test case selection using black-box testing approaches.

– Test case selection using white-box approaches.

– Some metrics like coverage and reliability that can be employed during testing.

8.1 Testing Concepts

In this section we will first define some of the terms that are commonly used when discussing testing. Then we will discuss some basic issues relating to how testing is performed, and the importance of psychology of the tester.

8.1.1 Error, Fault, and Failure

While discussing testing we commonly use terms like *error*, *fault*, *failure* etc. Let us start by defining these concepts clearly [52].

The term *error* is used in two different ways. It refers to the discrepancy between a computed, observed, or measured value and the true, specified, or theoretically correct value. That is, error refers to the difference between the actual output of a software and the correct output. In this interpretation, error is essentially a measure of the difference between the actual and the ideal. Error is also used to refer to human action that results in software containing a defect or fault. This definition is quite general and encompasses all the phases.

Fault is a condition that causes a system to fail in performing its required function. A fault is the basic reason for software malfunction and is practically synonymous with the commonly used term *bug*, or the somewhat more general term *defect*. The term *error* is also often used to refer to defects (taking a variation of the second definition of error). In this book we will continue to use the terms in the manner commonly used, and no explicit distinction will be made between errors and faults, unless necessary.

Failure is the inability of a system or component to perform a required function according to its specifications. A software failure occurs if the behavior of the software is different from the specified behavior. Failures may be caused by functional or performance factors. Note that the definition does not imply that a failure must be *observed.* It is possible that a failure may occur but not be detected.

There are some implications of these definitions. Presence of an error (in the state) implies that a failure must have occurred, and the observance of a failure implies that a fault must be present in the system. However, the presence of a fault does not imply that a failure must occur. The presence of a fault in a system only implies that the fault has a *potential* to cause a failure. Whether a fault actually manifests itself in a certain time duration depends on how the software is executed.

There are direct consequences of this on testing. If during testing we do not observe any errors, we cannot say anything about the presence or absence of faults in the system. If, on the other hand, we observe some failure, we can say that there are some faults in the system. This relationship of fault and failure makes the task of selecting test cases for testing very challenging—an objective while selecting test cases is to select those that will reveal the defect, if it exists. Ideally we would like the set of test cases to be such that if there are any defects in the system, some test case in the set will reveal it—something impossible to achieve in most situations.

It should be noted that during the testing process, only failures are observed, by which the presence of faults is deduced. That is, testing only reveals the presence of faults. The actual faults are identified by separate activities, commonly referred to as "debugging." In other words, for identifying faults, after testing has revealed the presence of faults, the expensive task of debugging has to be performed. This is one of the reasons why testing is an expensive method for identification of faults.

8.1.2 Test Case, Test Suite, and Test Harness

So far we have used the terms *test case* or *set of test cases* informally. Let us define them more precisely. A *test case* (often called a *test*) can be considered as comprising a set of test inputs and execution conditions, which are designed to exercise the SUT in a particular manner [52]. Generally, a test case also specifies the expected outcome from executing the SUT under the specified execution conditions and test inputs. A group of related test cases that are generally executed together to test some specific behavior or aspect of the SUT is often referred to as a *test suite.*

Note that in a test case, test inputs and execution conditions are mentioned separately. Test inputs are the specific values of parameters or other inputs that are given to the SUT either by the user or some other program. The execution conditions, on the other hand, reflect the state of the system and environment which also impact the behavior of the SUT. So, for example, while testing a function to add a record in the database if it does not already exist, the behavior of the function will depend both on the value of the input record as well as the state of the database. And a test case needs to specify both. For example, a test case for this function might specify a record r as input, and might specify that the state of the database be such that r already exists in it.

Testing can be done manually with the tester executing the test cases in the test suite and then checking if the behavior is as specified in the test cases. This is a very cumbersome process, particularly when the test suite contains a large number of test cases. It becomes even more cumbersome since the test suite often has to be executed every time the SUT is changed. Hence, the current trend is to automate testing.

With automated testing, a test case is typically a function call (or a method invocation), which does all the activities of a test case—it sets the test data and the test conditions, invokes the SUT as per the test case, compares the results returned with expected results, and declares to the tester whether the SUT failed or passed the test case. In other words, with automated testing, executing a test case essentially means executing this function. A test suite will then be a set of such functions, each representing a test case. To test a SUT with the test suite, generally an automated test script will be written which will invoke the test cases in the desired sequence.

To have a test suite executed automatically, we will need a framework in which test inputs can be defined, defined inputs can be used by functions representing test cases, the automated test script can be written, the SUT can be executed by this script, and the result of entire testing reported to the tester. Many *testing frameworks* now exist that permit all this to be done in a simple manner. A testing framework is also sometimes called a *test harness.* A test harness or a test framework makes the life of a tester simpler by providing easy means of defining a test suite, executing it, and reporting the results. With a test framework, a test suite is defined once, and then whenever needed, complete testing can be done by the click of a button or giving a command.

8.1.3 Psychology of Testing

As mentioned, in testing, the software under test (SUT) is executed with a set of test cases. As discussed, devising a set of test cases that will guarantee that all errors will be detected is not feasible. Moreover, there are no formal or precise methods for selecting test cases. Even though there are a number of heuristics and rules of thumb for deciding the test cases, selecting test cases is still a creative activity that relies on the ingenuity of the tester. Because of this, the psychology of the person performing the testing becomes important.

A basic purpose of testing is to detect the errors that may be present in the program. Hence, one should not start testing with the intent of showing that a program works; rather the intent should be to show that a program does not work; to reveal any defect that may exist. Due to this, testing has also been defined as the process of executing a program with the intent of finding errors [68].

This emphasis on proper intent of testing is not a trivial matter because test cases are designed by human beings, and human beings have a tendency to perform actions to achieve the goal they have in mind. So, if the goal is to demonstrate that a program works, we may consciously or subconsciously select test cases that will try to demonstrate that goal and that will beat the basic purpose of testing. On the other hand, if the intent is to show that the program does not work, we will challenge our intellect to find test cases toward that end, and we are likely to detect more errors. Testing is essentially a destructive process, where the tester has to treat the program as an adversary that must be beaten by the tester by showing the presence of errors. This is one of the reasons why many organizations employ *independent testing* in which testing is done by a team that was not involved in building the system.

8.1.4 Levels of Testing

Testing is usually relied upon to detect the faults remaining from earlier stages, in addition to the faults introduced during coding itself. Due to this, different levels of testing are used in the testing process; each level of testing aims to test different aspects of the system.

The basic levels are unit testing, integration testing, system testing, and acceptance testing. These different levels of testing attempt to detect different types of faults. The relation of the faults introduced in different phases, and the different levels of testing are shown in Figure 8.1.

The first level of testing is called *unit testing*, which we discussed in the previous chapter. Unit testing is essentially for verification of the code produced by individual programmers, and is typically done by the programmer of the module. Generally, a module is offered by a programmer for integration and use by others only after it has been unit tested satisfactorily.

The next level of testing is often called *integration testing.* In this, many unit tested modules are combined into subsystems, which are then tested. The goal here is to see if the modules can be integrated properly. Hence, the emphasis is

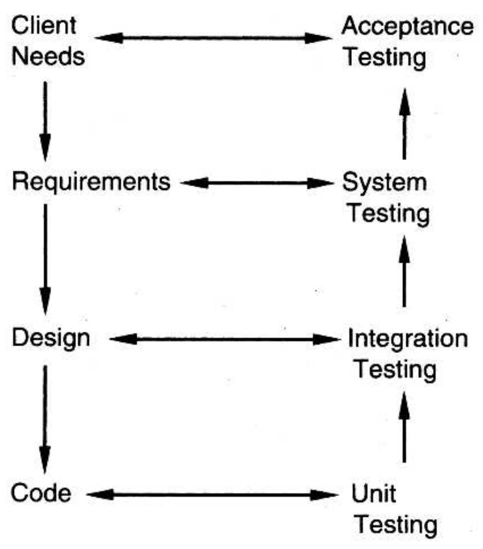

Figure 8.1: Levels of testing.

on testing interfaces between modules. This testing activity can be considered testing the design.

The next levels are *system testing* and *acceptance testing*. Here the entire software system is tested. The reference document for this process is the requirements document, and the goal is to see if the software meets its requirements. This is often a large exercise, which for large projects may last many weeks or months. This is essentially a validation exercise, and in many situations it is the only validation activity. Acceptance testing is often performed with realistic data of the client to demonstrate that the software is working satisfactorily. It may be done in the setting in which the software is to eventually function. Acceptance testing essentially tests if the system satisfactorily solves the problems for which it was commissioned.

These levels of testing are performed when a system is being built from the components that have been coded. There is another level of testing, called *regression testing*, that is performed when some changes are made to an existing system. We know that changes are fundamental to software; any software must undergo changes. However, when modifications are made to an existing system, testing also has to be done to make sure that the modification has not had any undesired side effect of making some of the earlier services faulty. That is, besides ensuring the desired behavior of the new services, testing has to ensure that the desired behavior of the old services is maintained. This is the task of regression testing.

For regression testing, some test cases that have been executed on the old system are maintained, along with the output produced by the old system. These test cases are executed again on the modified system and its output compared with the earlier output to make sure that the system is working as before on these test cases. This frequently is a major task when modifications are to be made to existing systems.

Complete regression testing of large systems can take a considerable amount of time, even if automation is used. If a small change is made to the system, often practical considerations require that the entire test suite not be executed, but regression testing be done with only a subset of test cases. This requires suitably selecting test cases from the suite which can test those parts of the system that could be affected by the change. Test case selection for regression testing is an active research area and many different approaches have been proposed in the literature for this. We will not discuss it any further.

8.2 Testing Process

The basic goal of the software development process is to produce software that has no errors or very few errors. Testing is a quality control activity which focuses on identifying defects (which are then removed). We have seen that different levels of testing are needed to detect the defects injected during the various tasks in the project. And at a level multiple SUTs may be tested. And for testing each SUT, test cases will have to be designed and then executed. Overall, testing in a project is a complex task which also consumes the maximum effort. Hence, testing has to be done properly in a project. The testing process for a project consists of three high-level tasks—test planning, test case design, and test execution. We will discuss these in the rest of this section.

8.2.1 Test Plan

In general, in a project, testing commences with a *test plan* and terminates with successful execution of acceptance testing. A test plan is a general document for the entire project that defines the scope, approach to be taken, and the schedule of testing, as well as identifies the test items for testing and the personnel responsible for the different activities of testing. The test planning can be done well before the actual testing commences and can be done in parallel with the coding and design activities. The inputs for forming the test plan are: (1) project plan, (2) requirements document, and (3) architecture or design document. The project plan is needed to make sure that the test plan is consistent with the overall quality plan for the project and the testing schedule matches that of the project plan. The requirements document and the design document are the basic documents used for selecting the test units and deciding the approaches to be used during testing. A test plan should contain the following:

– Test unit specification

– Features to be tested

– Approach for testing

– Test deliverables

– Schedule and task allocation

As seen earlier, different levels of testing have to be performed in a project. The levels are specified in the test plan by identifying the test units for the project. A test unit is a set of one or more modules that form a software under test (SUT). The identification of test units establishes the different levels of testing that will be performed in the project. Generally, a number of test units are formed during the testing, starting from the lower-level modules, which have to be unit-tested. That is, first the modules that have to be tested individually are specified as test units. Then the higher-level units are specified, which may be a combination of already tested units or may combine some already tested units with some untested modules. The basic idea behind forming test units is to make sure that testing is being performed incrementally, with each increment including only a few aspects that need to be tested.

An important factor while forming a unit is the "testability" of a unit. A unit should be such that it can be easily tested. In other words, it should be possible to form meaningful test cases and execute the unit without much effort with these test cases. For example, a module that manipulates the complex data structure formed from a file input by an input module might not be a suitable unit from the point of view of testability, as forming meaningful test cases for the unit will be hard, and driver routines will have to be written to convert inputs from files or terminals that are given by the tester into data structures suitable for the module. In this case, it might be better to form the unit by including the input module as well. Then the file input expected by the input module can contain the test cases.

Features to be tested include all software features and combinations of features that should be tested. A software feature is a software characteristic specified or implied by the requirements or design documents. These may include functionality, performance, design constraints, and attributes.

The *approach* for testing specifies the overall approach to be followed in the current project. The techniques that will be used to judge the testing effort should also be specified. This is sometimes called the *testing criterion* or the criterion for evaluating the set of test cases used in testing. In the previous sections we discussed many criteria for evaluating and selecting test cases.

Testing deliverables should be specified in the test plan before the actual testing begins. Deliverables could be a list of test cases that were used, detailed results of testing including the list of defects found, test summary report, and data about the code coverage.

The test plan typically also specifies the schedule and effort to be spent on different activities of testing, and the tools to be used. This schedule should be consistent with the overall project schedule. The detailed plan may list all the testing tasks and allocate them to *test resources* who are responsible for performing them. Many large products have separate testing teams and therefore a separate test plan. A smaller project may include the test plan as part of its quality plan in the project management plan.

8.2.2 Test Case Design

The test plan focuses on how the testing for the project will proceed, which units will be tested, and what approaches (and tools) are to be used during the various stages of testing. However, it does not deal with the details of testing a unit, nor does it specify which test cases are to be used.

Test case design has to be done separately for each unit. Based on the approach specified in the test plan, and the features to be tested, the test cases are designed and specified for testing the unit. Test case specification gives, for each unit to be tested, all test cases, inputs to be used in the test cases, conditions being tested by the test case, and outputs expected for those test cases. If test cases are specified in a document, the specifications look like a table of the form shown in Figure 8.2.

Requirement Number	Condition to be tested	Test data and settings	Expected output

Figure 8.2: Test case specifications.

Sometimes, a few columns are also provided for recording the outcome of different rounds of testing. That is, sometimes the test case specifications document is also used to record the result of testing. In a round of testing, the outcome of all the test cases is recorded (i.e., pass or fail). Hopefully, in a few rounds all test cases will pass.

With testing frameworks and automated testing, the testing scripts can be considered as test case specifications, as they clearly show what inputs are being given and what output to expect. With suitable comments, the intent of the test case can also be easily specified.

Test case design is a major activity in the testing process. Careful selection of test cases that satisfy the criterion and approach specified is essential for proper testing. We will later consider different techniques for designing good test cases.

There are some good reasons why test cases are specified before they are used for testing. It is known that testing has severe limitations and the effectiveness of testing depends very heavily on the exact nature of the test cases. It is therefore important to ensure that the set of test cases used is of high quality. Evaluation of test cases is often done through test case review. As for any review, a formal document or work product is needed, for review of test cases, the test case specification document is required. This is the primary reason for documenting the test cases. The test case specification document is reviewed, using a formal review process, to make sure that the test cases are consistent with the policy specified in the plan, satisfy the chosen criterion, and cover the various aspects of the unit to be tested. By reviewing the conditions being tested by the test cases, the reviewers can also check if all the important conditions are being tested.

Another reason for specifying the test cases in a document or a script is that by doing this, the tester can see the testing of the unit in totality and the effect of the total set of test cases. This type of evaluation is hard to do in on-the-fly testing where test cases are determined as testing proceeds. It also allows optimizing the number of test cases as evaluation of the test suite may show that some test cases are redundant.

8.2.3 Test Case Execution

With the specification of test cases, the next step in the testing process is to execute them. This step is also not straightforward. The test case specifications only specify the set of test cases for the unit to be tested. However, executing the test cases may require construction of driver modules or stubs. It may also require modules to set up the environment as stated in the test plan and test case specifications. Only after all these are ready can the test cases be executed.

If test frameworks are being used, then the setting of the environment as well as inputs for a test case is already done in the test scripts, and execution is straightforward.

During test case execution, defects are found. These defects are then fixed and tesing is done again to verify the fix. To facilitate reporting and tracking of defects found during testing (and other quality control activities), defects found are often logged. Defect logging is particularly important in a large software project which may have hundreds or thousands of defects that are found by different people at different stages of the project. Often the person who fixes a defect is not the person who finds or reports the defect. For example, a tester may find the defect while the developer of the code may actually fix it. In such a scenario, defect reporting and closing cannot be done informally. The use of informal mechanisms may easily lead to defects being found but later forgotten, resulting in defects not getting removed or in extra effort in finding the defect again. Hence, defects found must be properly logged in a system and their closure tracked. Defect logging and tracking is considered one of the best practices for managing a project [17], and is followed by most software organizations.

Let us understand the life cycle of a defect. A defect can be found by anyone at anytime. When a defect is found, it is logged in a defect control system, along with sufficient information about the defect. The defect is then in the state "submitted," essentially implying that it has been logged along with information about it. The job of fixing the defect is then assigned to some person, who is generally the author of the document or code in which the defect is found. The assigned person does the debugging and fixes the reported defect, and the defect then enters the "fixed" state. However, a defect that is fixed is still not considered as fully done. The successful fixing of the defect is verified. This verification may be done by another person (often the submitter), or by a test team, and typically involves running some tests. Once the defect fixing is verified, then the defect can be marked as "closed." In other words, the general life cycle of a defect has three states—submitted, fixed, and closed, as shown in Figure 8.3. A defect that is not closed is also called open.

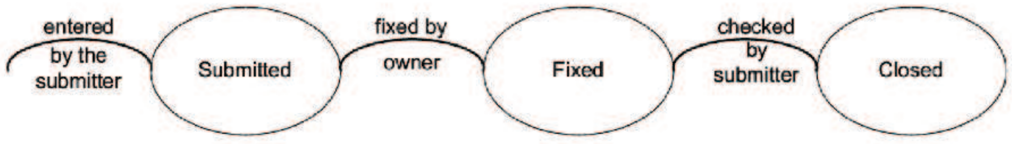

Figure 8.3: Life cycle of a defect.

This is a typical life cycle of a defect which is used in many organizations (e.g., [58]). Ideally, at the end of the project, no open defects should remain. However, this ideal situation is often not practical for most large systems. Besides using the log for tracking defects, the data in the log can also be used for analysis purposes. We will discuss some possible analysis later in the chapter.

8.3 Black-Box Testing

As we have seen, good test case design is the key to suitable testing of the SUT. The goal while testing a SUT is to detect most (hopefully all) of the defects, through as small a set of test cases as possible. Due to this basic goal, it is important to select test cases carefully—best are those test cases that have a high probability of detecting a defect, if it exists, and also whose execution will give a confidence that no failures during testing implies that there are few (hopefully none) defects in the software.

There are two basic approaches to designing the test cases to be used in testing: black-box and white-box. In black-box testing the structure of the program is not considered. Test cases are decided solely on the basis of the requirements or specifications of the program or module, and the internals of the module or the program are not considered for selection of test cases. In this section, we will present some techniques for generating test cases for black-box testing. White-box testing is discussed in the next section.

In black-box testing, the tester only knows the inputs that can be given to the system and what output the system should give. In other words, the basis for deciding test cases is the requirements or specifications of the system or module. This form of testing is also called functional or behavioral testing.

The most obvious functional testing procedure is exhaustive testing, which is impractical. One criterion for generating test cases is to generate them randomly. This strategy has little chance of resulting in a set of test cases that is close to optimal (i.e., that detects the maximum errors with minimum test cases). Hence, we need some other criterion or rule for selecting test cases. There are no formal rules for designing test cases for functional testing. However, there are a number of techniques or heuristics that can be used to select test cases that have been found to be very successful in detecting errors. Here we mention some of these techniques.

8.3.1 Equivalence Class Partitioning

Because we cannot do exhaustive testing, the next natural approach is to divide the input domain into a set of equivalence classes, so that if the program works correctly for a value, then it will work correctly for all the other values in that class. If we can indeed identify such classes, then testing the program with one value from each equivalence class is equivalent to doing an exhaustive test of the program.

However, without looking at the internal structure of the program, it is impossible to determine such ideal equivalence classes (even with the internal structure, it usually cannot be done). The equivalence class partitioning method [68] tries to approximate this ideal. An equivalence class is formed of the inputs for which the behavior of the system is specified or expected to be similar. Each group of inputs for which the behavior is expected to be different from others is considered a separate equivalence class. The rationale of forming equivalence classes like this is the assumption that if the specifications require the same behavior for each element in a class of values, then the program is likely to be constructed so that it either succeeds or fails for each of the values in that class. For example, the specifications of a module that determines the absolute value for integers specify one behavior for positive integers and another for negative integers. In this case, we will form two equivalence classes—one consisting of positive integers and the other consisting of negative integers.

For robust software, we must also consider invalid inputs. That is, we should define equivalence classes for invalid inputs also.

Equivalence classes are usually formed by considering each condition specified on an input as specifying a valid equivalence class and one or more invalid equivalence classes. For example, if an input condition specifies a range of values (say, $0 < \text{count} < \text{Max}$), then form a valid equivalence class with that range and two invalid equivalence classes, one with values less than the lower bound of the range (i.e., $\text{count} < 0$) and the other with values higher than the higher bound ($\text{count} > \text{Max}$). If the input specifies a set of values and the requirements specify different behavior for different elements in the set, then a valid equivalence class is formed for each of the elements in the set and an invalid class for an entity not belonging to the set.

One common approach for determining equivalence classes is as follows. If there is reason to believe that the entire range of an input will not be treated in the same manner, then the range should be split into two or more equivalence classes, each consisting of values for which the behavior is expected to be similar. For example, for a character input, if we have reasons to believe that the program will perform different actions if the character is a letter, a number, or a special character, then we should split the input into three valid equivalence classes.

Another approach for forming equivalence classes is to consider any special value for which the behavior could be different as an equivalence class. For example, the value 0 could be a special value for an integer input.

Also, for each valid equivalence class, one or more invalid equivalence classes should be identified.

It is often useful to consider equivalence classes in the output. For an output equivalence class, the goal is to have inputs such that the output for that test case lies in the output equivalence class. As an example, consider a program for determining rate of return for some investment. There are three clear output equivalence classes—positive rate of return, negative rate of return, and zero rate of return. During testing, it is important to test for each of these, that is, give inputs such that each of these three outputs is generated. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.

Once equivalence classes are selected for each of the inputs, then the issue is to select test cases suitably. There are different ways to select the test cases. One strategy is to select each test case covering as many valid equivalence classes as it can, and one separate test case for each invalid equivalence class. A somewhat better strategy which requires more test cases is to have a test case cover at most one valid equivalence class for each input, and have one separate test case for each invalid equivalence class. In the latter case, the number of test cases for valid equivalence classes is equal to the largest number of equivalence classes for any input, plus the total number of invalid equivalence classes.

As an example, consider a program that takes two inputs—a string s of length up to N and an integer n. The program is to determine the top n highest occurring characters in s. The tester believes that the programmer may deal with different types of characters separately. One set of valid and invalid equivalence classes for this is shown in Table 8.1.

Table 8.1: Valid and invalid equivalence classes.

Input	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
s	EQ1: Contains numbers EQ2: Contains lowercase letters EQ3: Contains uppercase letters EQ4: Contains special characters EQ5: String length between 0-N	IEQ1: non-ASCII characters IEQ2: String length > N
n	EQ6: Integer in valid range	IEQ3: Integer out of range

With these as the equivalence classes, we have to select the test cases. A test case for this is a pair of values for s and n. With the first strategy for deciding test cases, one test case could be: s as a string of length less than N containing lowercase, uppercase, numbers, and special characters; and n as the number 5. This one test case covers all the valid equivalence classes (EQ1 through EQ6). Then we will have one test case each for covering IEQ1, IEQ2, and IEQ3. That is, a total of four test cases is needed.

With the second approach, in one test case we can cover one equivalence class for one input only. So, one test case could be: a string of numbers, and the number 5. This covers EQ1 and EQ6. Then we will need test cases for EQ2 through EQ5, and separate test cases for IEQ1 through IEQ3.

8.3.2 Boundary Value Analysis

It has been observed that programs that work correctly for a set of values in an equivalence class fail on some special values. These values often lie on the boundary of the equivalence class. Test cases that have values on the boundaries of equivalence classes are therefore likely to be "high-yield" test cases, and selecting such test cases is the aim of boundary value analysis. In boundary value analysis [68], we choose an input for a test case from an equivalence class, such that the input lies at the edge of the equivalence classes. Boundary values for each equivalence class, including the equivalence classes of the output, should be covered. Boundary value test cases are also called "extreme cases." Hence, we can say that a boundary value test case is a set of input data that lies on the edge or boundary of a class of input data or that generates output that lies at the boundary of a class of output data.

In case of ranges, for boundary value analysis it is useful to select the boundary elements of the range and an invalid value just beyond the two ends (for the two invalid equivalence classes). So, if the range is $0.0 \leq \mathrm{x} \leq 1.0$, then the test cases are 0.0, 1.0 (valid inputs), and -0.1, and 1.1 (for invalid inputs). Similarly, if the input is a list, attention should be focused on the first and last elements of the list.

We should also consider the outputs for boundary value analysis. If an equivalence class can be identified in the output, we should try to generate test cases that will produce the output that lies at the boundaries of the equivalence classes. Furthermore, we should try to form test cases that will produce an output that does not lie in the equivalence class. (If we can produce an input case that produces the output outside the equivalence class, we have detected an error.)

Like in equivalence class partitioning, in boundary value analysis we first determine values for each of the variables that should be exercised during testing. If there are multiple inputs, then how should the set of test cases be formed covering the boundary values? Suppose each input variable has a defined range. Then there are six boundary values—the extreme ends of the range, just beyond the ends, and just before the ends. If an integer range is min to max, then the six values are $min - 1, min, min + 1, max - 1, max, max + 1$. Suppose there are n such input variables. There are two strategies for combining the boundary values for the different variables in test cases.

In the first strategy, we select the different boundary values for one variable, and keep the other variables at some nominal value. And we select one test case consisting of nominal values of all the variables. In this case, we will have $6n+1$ test cases. For two variables X and Y, the 13 test cases will be as shown in Figure 8.4.

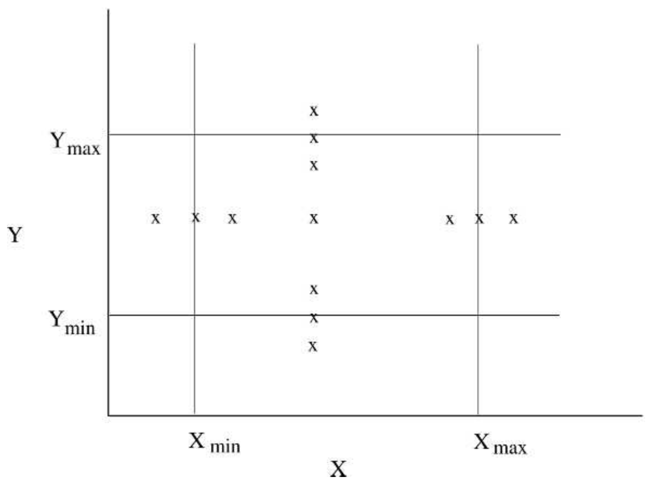

Figure 8.4: Test cases for boundary value analysis.

Another strategy would be to try all possible combinations for the values for the different variables. As there are seven values for each variable (six boundary values and one nominal value), if there are n variables, there will be a total of 7^n test cases—too large for practical testing.

8.3.3 Pairwise Testing

There are generally many parameters that determine the behavior of a software system. These parameters could be direct input to the software or implicit settings like those for devices. These parameters can take different values, and for some of them the software may not work correctly. Many of the defects in software generally involve one condition, that is, some special value of one of the parameters. Such a defect is called a single-mode fault [70]. Simple examples of single-mode faults are a software not able to print for a particular type of printer, a software that cannot compute fare properly when the traveler is a minor, and a telephone billing software that does not compute the bill properly for a particular country.

Single-mode faults can be detected by testing for different values of different parameters. So, if there are n parameters for a system, and each one of them can take m different values (or m different classes of values, each class being considered as the same for purposes of testing as in equivalence class partitioning), then with each test case we can test one different value of each parameter. In other words, we can test for all the different values in m test cases.

However, all faults are not single-mode and there are combinations of inputs that reveal the presence of faults: for example, a telephone billing software that does not compute correctly for nighttime calling (one parameter) to a particular country (another parameter), or an airline ticketing system that has incorrect behavior when a minor (one parameter) is traveling business class (another parameter) and not staying over the weekend (third parameter). These multi-mode faults can be revealed during testing by trying different combinations of the parameter values—an approach called *combinatorial testing*.

Unfortunately, full combinatorial testing is often not feasible. For a system with n parameters, each having m values, the number of different combinations is n^m. For a simple system with 5 parameters, each having 5 different values, the total number of combinations is 3,125. And if testing each combination takes 5 minutes, it will take over 1 month to test all combinations. Clearly, for complex systems that have many parameters and each parameter may have many values, a full combinatorial testing is not feasible and practical techniques are needed to reduce the number of tests.

Some research has suggested that most software faults are revealed on some special single values or by interaction of a pair of values [25]. That it, most faults tend to be either single-mode or double-mode. For testing for double-mode faults, we need not test the system with all the combinations of parameter values, but need to test such that all combinations of values for each pair of parameters are exercised. This is called *pairwise testing*.

In pairwise testing, all pairs of values have to be exercised during testing. If there are n parameters, each with m values, then between each two parameter we have $m * m$ pairs. The first parameter will have these many pairs with each of the remaining $n - 1$ parameters, the second one will have new pairs with $n - 2$ parameters (as its pairs with the first are already included in the first parameter pairs), the third will have pairs with $n - 3$ parameters, and so on. That is, the total number of pairs is $m * m * n * (n - 1)/2$.

The objective of pairwise testing is to have a set of test cases that cover all the pairs. As there are n parameters, a test case is a combination of values of these parameters and will cover $(n-1)+(n-2)+... = n(n-1)/2$ pairs. In the best case when each pair is covered exactly once by one test case, m^2 different test cases will be needed to cover all the pairs.

As an example, consider a software product being developed for multiple platforms that uses the browser as its interface. Suppose the software is being designed to work for three different operating systems and three different browsers. In addition, as the product is memory intensive there is a desire to test its performance under different levels of memory. So, we have the following three parameters with their different values:

```
Operating System: Windows, Solaris, Linux
Memory Size: 128M, 256M, 512M
Browser: IE, Netscape, Mozilla
```

For discussion, we can say that the system has three parameters: A (operating system), B (memory size), and C (browser). Each of them can have three values which we will refer to as a_1, a_2, a_3, b_1, b_2, b_3, and c_1, c_2, c_3. The total number of pairwise combinations is 9 * 3 = 27. The number of test cases, however, to cover all the pairs is much less. A test case consisting of values of the three parameters covers three combinations (of A-B, B-C, and A-C). Hence, in the best case, we can cover all 27 combinations by 27/3=9 test cases. These test cases are shown in Table 8.2, along with the pairs they cover.

Table 8.2: Test cases for pairwise testing.

A	**B**	**C**	**Pairs**
a1	b1	c1	(a1,b1) (a1,c1) (b1,c1)
a1	b2	c2	(a1,b2) (a1,c2) (b2,c2)
a1	b3	c3	(a1,b3) (a1,c3) (b3,c3)
a2	b1	c2	(a2,b1) (a2,c2) (b1,c2)
a2	b2	c3	(a2,b2) (a2,c3) (b2,c3)
a2	b3	c1	(a2,b3) (a2,c1) (b3,c1)
a3	b1	c3	(a3,b1) (a3,c3) (b1,c3)
a3	b2	c1	(a3,b2) (a3,c1) (b2,c1)
a3	b3	c2	(a3,b3) (a3,c2) (b3,c2)

As should be clear, generating test cases to cover all the pairs is not a simple task. The minimum set of test cases is that in which each pair is covered by exactly one test case. Often, it will not be possible to generate the minimum set of test cases, particularly when the number of values for different parameters is different. Various algorithms have been proposed, and some programs are available online to generate the test cases to cover all the pairs.

For situations where manual generation is feasible, the following approach can be followed. Start with initial test cases formed by all combinations of values for the two parameters which have the largest number of values (as we must have at least this many test cases to test all the pairs for these two parameters). Then complete each of these test cases by adding value for other parameters such that they add pairs that have not yet been covered by any test case. When all are completed, form additional test cases by combining as many uncovered pairs as possible. Essentially we are generating test cases such that a test case covers as many new pairs as possible. By avoiding covering pairs multiple times, we can produce a small set of test cases that cover all pairs. Efficient algorithms of generating the smallest number of test cases for pairwise testing exist. In [25] an example is given in which for 13 parameters, each having three distinct values, all pairs are covered in merely 15 test cases, while the total number of combinations is over 1 million!

Pairwise testing is a practical way of testing large software systems that have many different parameters with distinct functioning expected for different values. An example would be a billing system (for telephone, hotel, airline, etc.) which has different rates for different parameter values. It is also a practical approach for testing general-purpose software products that are expected to run on different platforms and configurations, or a system that is expected to work with different types of systems.

8.3.4 Special Cases

It has been seen that programs often produce incorrect behavior when inputs form some special cases. The reason is that in programs, some combinations of inputs need special treatment, and providing proper handling for these special cases is easily overlooked. For example, in an arithmetic routine, if there is a division and the divisor is zero, some special action has to be taken, which could easily be forgotten by the programmer. These special cases form particularly good test cases, which can reveal errors that will usually not be detected by other test cases.

Special cases will often depend on the data structures and the function of the module. There are no rules to determine special cases, and the tester has to use his intuition and experience to identify such test cases. Consequently, determining special cases is also called *error guessing.*

Psychology is particularly important for error guessing. The tester should play the “devil’s advocate” and try to guess the incorrect assumptions the programmer could have made and the situations the programmer could have overlooked or handled incorrectly. Essentially, the tester is trying to identify error-prone situations. Then test cases are written for these situations. For example, in the problem of finding the number of different words in a file (discussed in earlier chapters) some of the special cases can be: file is empty, only one word in the file, only one word in a line, some empty lines in the input file, presence of more than one blank between words, all words are the same, the words are already sorted, and blanks at the start and end of the file.

Incorrect assumptions are usually made because the specifications are not complete or the writer of specifications may not have stated some properties, assuming them to be obvious. Whenever there is reliance on tacit understanding rather than explicit statement of specifications, there is scope for making wrong assumptions. Frequently, wrong assumptions are made about the environments. However, it should be pointed out that special cases depend heavily on the problem, and the tester should really try to "get into the shoes" of the designer and coder to determine these cases.

8.3.5 State-Based Testing

There are some systems that are essentially stateless in that for the same inputs they always give the same outputs or exhibit the same behavior. Many batch processing systems, computational systems, and servers fall in this category. In hardware, combinatorial circuits fall in this category. At a smaller level, most functions are supposed to behave in this manner. There are, however, many systems whose behavior is state-based in that for identical inputs they behave differently at different times and may produce different outputs. The reason for different behavior is that the state of the system may be different. In other words, the behavior and outputs of the system depend not only on the inputs provided, but also on the state of the system. The state of the system depends on the past inputs the system has received. In other words, the state represents the cumulative impact of all the past inputs on the system. In hardware, the sequential systems fall in this category. In software, many large systems fall in this category as past state is captured in databases or files and used to control the behavior of the system. For such systems, another approach for selecting test cases is the state-based testing approach [22].

Theoretically, any software that saves state can be modeled as a state machine. However, the state space of any reasonable program is almost infinite, as it is a cross product of the domains of all the variables that form the state. For many systems the state space can be partitioned into a few states, each representing a logical combination of values of different state variables which share some property of interest [9]. If the set of states of a system is manageable, a state model of the system can be built. A state model for a system has four components:

– *States.* Represent the impact of the past inputs to the system.

– *Transitions.* Represent how the state of the system changes from one state to another in response to some events.

– *Events.* Inputs to the system.

– *Actions.* The outputs for the events.

The state model shows what state transitions occur and what actions are performed in a system in response to events. When a state model is built from the requirements of a system, we can only include the states, transitions, and actions that are stated in the requirements or can be inferred from them. If more information is available from the design specifications, then a richer state model can be built.

For example, consider the student survey example discussed in Chapter 5. According to the requirements, a system is to be created for taking a student survey. The student takes a survey and is returned the current result of the survey. The survey result can be up to five surveys old. We consider the architecture which had a cache between the server and the database, and in which the survey and results are cached and updated only after five surveys, on arrival of a request. The proposed architecture has a database at the back, which may go down.

To create a state machine model of this system, we notice that of a series of 6 requests, the first 5 may be treated differently. Hence, we divide into two states: one representing the the receiving of $1 - 4$ requests (state 1), and the other representing the receiving of request 5 (state 2). Next we see that the database can be up or down, and it can go down in any of these two states. However, the behavior of requests, if the database is down, may be different. Hence, we create another pair of states (states 3 and 4). Once the database has failed, then the first 5 requests are serviced using old data. When a request is received after receiving 5 requests, the system enters a failed state (state 5), in which it does not give any response. When the system recovers from the failed state, it must update its cache immediately, hence goes to state 2. The state model for this system is shown in Figure 8.5 (i represents an input from the user for taking the survey).

Note that we are assuming that the state model of the system can be created from its specifications or design. This is how most state modeling is done, and that is how the model was built in the example. Once the state model is built, we can use it to select test cases. When the design is implemented, these test

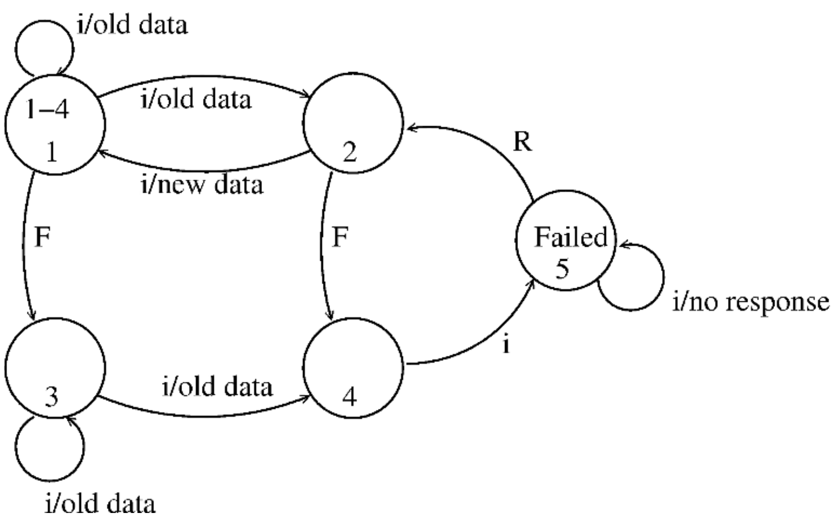

Figure 8.5: State model for the student survey system.

cases can be used for testing the code. It is because of this we treat state-based testing as a black box testing strategy.

However, the state model often requires information about the design of the system. In the example above, some knowledge of the architecture is utilized. Sometimes making the state model may require detailed information about the design of the system. For example, for a class, we have seen that the state modeling is done during design, and when a lot is already known about the class, its attributes, and its methods. Due to this, the state-based testing may be considered as somewhat between black-box and white-box testing. Such strategies are sometimes called *gray-box testing.*

Given a state model of a system, how should test cases be generated? Many coverage criteria have been proposed [69]. We discuss only a few here. Suppose the set of test cases is T. Some of the criteria are:

– **All transition coverage (AT).** T must ensure that every transition in the state graph is exercised.

– **All transitions pair coverage (ATP).** T must execute all pairs of adjacent transitions. (An adjacent transition pair comprises two transitions: an incoming transition to a state and an outgoing transition from that state.)

– **Transition tree coverage (TT).** T must execute all simple paths, where a simple path is one which starts from the start state and reaches a state that it has already visited in this path or a final state.

The first criterion states that during testing all transitions get fired. This will also ensure that all states are visited. The transition pair coverage is a stronger criterion requiring that all combinations of incoming and outgoing transitions for each state must be exercised by T. If a state has two incoming transitions t1 and t2, and two outgoing transitions t3 and t4, then a set of test cases T that executes t1;t3 and t2;t4 will satisfy AT. However, to satisfy ATP, T must also ensure execution of t1;t4 and t2;t3. The transition tree coverage is named in this manner as a transition tree can be constructed from the graph and then used to identify the paths. In ATP, we are going beyond transitions, and stating that different paths in the state diagram should be exercised during testing. ATP will generally include AT.

For the example above, the set of test cases for AT are given below in Table 8.3. Here req() means that a request for taking the survey should be given, fail() means that the database should be failed, and recover() means that the failed database should be recovered.

Table 8.3: Test cases for state-based testing criteria.

S.No.	**Transition**	**Test case**
1	$1 \rightarrow 2$	req()
2	$1 \rightarrow 2$	req();req();req();req();req();req()
3	$2 \rightarrow 1$	seq for 2; req()
4	$1 \rightarrow 3$	req();fail()
5	$3 \rightarrow 3$	req();fail();req()
6	$3 \rightarrow 4$	req();fail();req();req();req();req();req()
7	$4 \rightarrow 5$	seq for 6; req()
8	$5 \rightarrow 2$	seq for 6; req();recover()

As we can see, state-based testing draws attention to the states and transitions. Even in the above simple case, we can see different scenarios get tested (e.g., system behavior when the database fails, and system behavior when it fails and recovers thereafter). Many of these scenarios are easy to overlook if test cases are designed only by looking at the input domains. The set of test cases is richer if the other criteria are used. For this example, we leave it as an exercise to determine the test cases for other criteria.

8.4 White-Box Testing

In the previous section we discussed black-box testing, which is concerned with the function that the tested program is supposed to perform and does not deal with the internal structure of the program responsible for actually implementing that function. Thus, black-box testing is concerned with functionality rather than implementation of the program. White-box testing, on the other hand, is concerned with testing the implementation of the program. The intent of this testing is not to exercise all the different input or output conditions (although that may be a by-product) but to exercise the different programming structures and data structures used in the program. White-box testing is also called *structural testing*, and we will use the two terms interchangeably.

To test the structure of a program, structural testing aims to achieve test cases that will force the desired coverage of different structures. Various criteria have been proposed for this. Unlike the criteria for functional testing, which are frequently imprecise, the criteria for structural testing are generally quite precise as they are based on program structures, which are formal and precise. Here we will discuss one approach to structural testing: control flow-based testing, which is most commonly used in practice.

8.4.1 Control Flow-Based Criteria

Most common structure-based criteria are based on the control flow of the program. In these criteria, the control flow graph of a program is considered and coverage of various aspects of the graph are specified as criteria. Hence, before we consider the criteria, let us precisely define a control flow graph for a program.

Let the *control flow graph* (or simply *flow graph*) of a program P be G. A node in this graph represents a block of statements that is always executed together, i.e., whenever the first statement is executed, all other statements are also executed. An edge (i, j) (from node i to node j) represents a possible transfer of control after executing the last statement of the block represented by node i to the first statement of the block represented by node j. A node corresponding to a block whose first statement is the start statement of P is called the *start* node of G, and a node corresponding to a block whose last statement is an exit statement is called an *exit* node [73]. A *path* is a finite sequence of nodes $(n_1, n_2, ..., n_k), k > 1$, such that there is an edge (n_i, n_{i+1}) for all nodes n_i in the sequence (except the last node n_k). A *complete path* is a path whose first node is the start node and the last node is an exit node.

Now let us consider control flow-based criteria. Perhaps the simplest coverage criterion is *statement coverage*, which requires that each statement of the program be executed at least once during testing. In other words, it requires that the paths executed during testing include all the nodes in the graph. This is also called the *all-nodes* criterion [73].

This coverage criterion is not very strong, and can leave errors undetected. For example, if there is an `if` statement in the program without having an `else` clause, the statement coverage criterion for this statement will be satisfied by a test case that evaluates the condition to true. No test case is needed that ensures that the condition in the `if` statement evaluates to false. This is a serious shortcoming because decisions in programs are potential sources of errors. As an example, consider the following function to compute the absolute value of a number:

```
int abs (x)
int x;
{
      if (x >= 0) x = 0 - x;
      return (x)
}
```

This program is clearly wrong. Suppose we execute the function with the set of test cases { x=0 } (i.e., the set has only one test case). The statement coverage criterion will be satisfied by testing with this set, but the error will not be revealed.

A more general coverage criterion is *branch coverage*, which requires that each edge in the control flow graph be traversed at least once during testing. In other words, branch coverage requires that each decision in the program be evaluated to true and false values at least once during testing. Testing based on branch coverage is often called *branch testing.* The 100% branch coverage criterion is also called the *all-edges* criterion [73]. Branch coverage implies statement coverage, as each statement is a part of some branch. In the preceding example, a set of test cases satisfying this criterion will detect the error.

The trouble with branch coverage comes if a decision has many conditions in it (consisting of a Boolean expression with Boolean operators *and* and *or*). In such situations, a decision can evaluate to true and false without actually exercising all the conditions. For example, consider the following function that checks the validity of a data item. The data item is valid if it lies between 0 and 100.

```
int check(x)
int x;
{
     if ((x >= ) && (x <= 200))
          check = True;
     else check = False;
}
```

The module is incorrect, as it is checking for x $\leq$ 200 instead of 100 (perhaps a typing error made by the programmer). Suppose the module is tested with the following set of test cases: { x = 5, x = -5 }. The branch coverage criterion will be satisfied for this module by this set. However, the error will not be revealed, and the behavior of the module is consistent with its specifications for all test cases in this set. Thus, the coverage criterion is satisfied, but the error is not detected. This occurs because the decision is evaluating to true and false because of the condition (x $\geq$ 0). The condition (x $\leq$ 200) never evaluates to false during this test, hence the error in this condition is not revealed.

This problem can be resolved by requiring that all conditions evaluate to true and false. However, situations can occur where a decision may not get both true and false values even if each individual condition evaluates to true and false. An obvious solution to this problem is to require decision/condition coverage, where all the decisions and all the conditions in the decisions take both true and false values during the course of testing.

Studies have indicated that there are many errors whose presence is not detected by branch testing because some errors are related to some combinations of branches and their presence is revealed by an execution that follows the path that includes those branches. Hence, a more general coverage criterion is one that requires all possible paths in the control flow graph be executed during testing. This is called the *path coverage* criterion or the *all-paths* criterion, and the testing based on this criterion is often called *path testing.* The difficulty with this criterion is that programs that contain loops can have an infinite number of possible paths. Furthermore, not all paths in a graph may be "feasible" in the sense that there may not be any inputs for which the path can be executed.

As the path coverage criterion leads to a potentially infinite number of paths, some efforts have been made to suggest criteria between the branch coverage and path coverage. The basic aim of these approaches is to select a set of paths that ensure branch coverage criterion and try some other paths that may help reveal errors. One method to limit the number of paths is to consider two paths the same if they differ only in their subpaths that are caused due to the loops. Even with this restriction, the number of paths can be extremely large.

It should be pointed out that none of these criteria is sufficient to detect all kind of errors in programs. For example, if a program is missing some control flow paths that are needed to check for a special value (like pointer equals nil and divisor equals zero), then even executing all the paths will not necessarily detect the error. Similarly, if the set of paths is such that they satisfy the all-path criterion but exercise only one part of a compound condition, then the set will not reveal any error in the part of the condition that is not exercised.

Hence, even the path coverage criterion, which is the strongest of the criteria we have discussed, is not strong enough to guarantee detection of all the errors.

8.4.2 Test Case Generation and Tool Support

Once a coverage criterion is decided, two problems have to be solved to use the chosen criterion for testing. The first is to decide if a set of test cases satisfy the criterion, and the second is to generate a set of test cases for a given criterion. Deciding whether a set of test cases satisfy a criterion without the aid of any tools is a cumbersome task, though it is theoretically possible to do manually. For almost all the structural testing techniques, tools are used to determine whether the criterion has been satisfied. Generally, these tools will provide feedback regarding what needs to be tested to fully satisfy the criterion.

To generate the test cases, tools are not that easily available, and due to the nature of the problem (i.e., undecidability of "feasibility" of a path), a fully automated tool for selecting test cases to satisfy a criterion is generally not possible. Hence, tools can, at best, aid the tester. One method for generating test cases is to randomly select test data until the desired criterion is satisfied (which is determined by a tool). This can result in a lot of redundant test cases, as many test cases will exercise the same paths.

As test case generation cannot be fully automated, frequently the test case selection is done manually by the tester by performing structural testing in an iterative manner, starting with an initial test case set and selecting more test cases based on the feedback provided by the tool for test case evaluation. The test case evaluation tool can tell which paths need to be executed or which mutants need to be killed. This information can be used to select further test cases.

Even with the aid of tools, selecting test cases is not a simple process. Selecting test cases to execute some parts of as yet unexecuted code is often very difficult. Because of this, and for other reasons, the criteria are often weakened. For example, instead of requiring 100% coverage of statements and branches, the goal might be to achieve some acceptably high percentage (but less than 100%).

There are many tools available for statement and branch coverage, the criteria that are used most often. Both commercial and freeware tools are available for different source languages. These tools often also give higher-level coverage data like function coverage, method coverage, and class coverage. To get the coverage data, the execution of the program during testing has to be closely monitored. This requires that the program be instrumented so that required data can be collected. A common method of instrumenting is to insert some statements called *probes* in the program. The sole purpose of the probes is to generate data about program execution during testing that can be used to compute the coverage. With this, we can identify three phases in generating coverage data:

1. Instrument the program with probes
2. Execute the program with test cases
3. Analyze the results of the probe data

Probe insertion can be done automatically by a *preprocessor*. The execution of the program is done by the tester. After testing, the coverage data is displayed by the tool—sometimes graphical representations are also shown.

8.5 Metrics

We have seen that during testing the software under test is executed with a set of test cases. As the quality of delivered software depends substantially on the quality of testing, a few natural questions arise while testing:

– How good is the testing that has been done?

– What is the quality or reliability of software after testing is completed?

During testing, the primary purpose of metrics is to try to answer these and other related questions. We will discuss some metrics that may be used for this purpose.

8.5.1 Coverage Analysis

One of the most commonly used approaches for evaluating the thoroughness of testing is to use some coverage measures. We have discussed above some of the common coverage measures that are used in practice—statement coverage and branch coverage. To use these coverage measures for evaluating the quality of testing, proper coverage analysis tools will have to be employed which can inform not only the coverage achieved during testing but also which portions are not yet covered.

Often, organizations build guidelines for the level of coverage that must be achieved during testing. Generally, the coverage requirement will be higher for unit testing, but lower for system testing as it is much more difficult to ensure execution of identified blocks when the entire system is being executed. Often the coverage requirement at unit level can be 90% to 100% (keep in mind that 100% may not be always possible as there may be unreachable code).

Besides the coverage of program constructs, coverage of requirements is also often examined. It is for facilitating this evaluation that in test case specification the requirement or condition being tested is mentioned. This coverage is generally established by evaluating the set of test cases to ensure that sufficient number of test cases with suitable data are included for all the requirements. The coverage measure here is the percentage of requirements or their clauses/-conditions for which at least one test case exists. Often a full coverage may be required at requirement level before testing is considered as acceptable.

8.5.2 Reliability

After testing is done and the software is delivered, the development is considered over. It will clearly be desirable to know, in quantifiable terms, the reliability of the software being delivered. As reliability of software depends considerably on the quality of testing, by assessing reliability we can also judge the quality of testing. Alternatively, reliability estimation can be used to decide whether enough testing has been done. In other words, besides characterizing an important quality property of the product being delivered, reliability estimation has a direct role in project management—it can be used by the project manager to decide whether enough testing has been done and when to stop testing.

Reliability of a product specifies the probability of failure-free operation of that product for a given time duration. Most reliability models require that the occurrence of failure be a random phenomenon. In software even though failures occur due to preexisting bugs, this assumption will generally hold for larger systems, but may not hold for small programs that have bugs (in which case one might be able to predict the failures). Hence, reliability modeling is more meaningful for larger systems.

Let X be the random variable that represents the life of a system. Reliability of a system is the probability that the system has not failed by time t. In other words,

$$R(t) = P(X > t).$$

The reliability of a system can also be specified as the *mean time to failure* (MTTF). MTTF represents the expected lifetime of the system. From the reliability function, it can be obtained as [80]

$$MTTF = \int_0^\infty R(x)dx.$$

Reliability can also be defined in terms of failure intensity which is the failure rate (i.e., number of failures per unit time) of the software at time t.

From the measurement perspective, during testing, measuring failure rate is the easiest, if defects are being logged. A simple way to do this is to compute the number of failures every week or every day during the last stages of testing. And number of failures can be approximated by the number of defects logged. (Though failures and defects are different, in the last stages of testing it is assumed that defects that cause failures are fixed soon enough and therefore do not cause multiple failures.) Generally, this failure rate increases in the start of testing as more and more defects are found, peaks somewhere in the middle of testing, and then continues to drop as fewer defects are reported. For a given test suite, if all defects are fixed, then there should be almost no failures toward the end. And that could be considered as proper time for release of this software. That is, a release criterion could be that the failure rate at release time is zero failures in some time duration, or zero failures while executing a test suite.

Though failure rate tracking gives a rough sense of reliability in terms of failures per day or per week, for more accurate reliability estimation, better models have to be used. Software reliability modeling is a complex task, requiring rigorous models and sophisticated statistical analysis. Many models have been proposed for software reliability assessment, and a survey of many of the models is given in [33, 67].

It should be mentioned that as failure of software also depends critically on the environment in which it is executing, failure rates experienced in testing will reflect the ultimate reliability experienced by the user after software release only if testing closely mimics the user behavior. This may not be the case, particularly with lower levels of testing. However, often at higher levels, active effort is made to have the final test suite mimic the actual usage. If this is the case, then reliability estimation can be applied with a higher confidence.

8.5.3 Defect Removal Efficiency

Another analysis of interest is *defect removal efficiency*, though this can only be determined sometime after the software has been released. The purpose of this analysis is to evaluate the effectiveness of the testing process being employed, not the quality of testing for a project. This analysis is useful for improving the testing process in the future.

Usually, after the software has been released to the client, the client will find defects, which have to be fixed (generally by the original developer, as this is often part of the contract). This defect data is also generally logged. Within a few months, most of the defects would be uncovered by the client (often the "warranty" period is 3 to 6 months).

Once the total number of defects (or a close approximation to the total) is known, the *defect removal efficiency* (DRE) of testing can be computed. The defect removal efficiency of a quality control activity is defined as the percentage reduction in the number of defects by executing that activity [61]. As an example, suppose the total number of defects logged is 500, out of which 20 were found after delivery, and 200 were found during the system testing. The defect removal efficiency of system testing is 200/220 (just about 90%), as the total number of defects present in the system when testing started was 220. The defect removal efficiency of the overall quality process is 480/500, which is 96%. Incidentally, this level of DRE is decent and is what many commercial organizations achieve.

It should be clear that DRE is a general concept which can be applied to any defect removal activity. For example, we can compute the DRE of design review, or unit testing. This can be done if for each defect, besides logging when and where the defect is found, the phase in which the defect was introduced is also analyzed and logged. With this information, when all the defects are logged, the DRE of the main quality control tasks can be determined. This information is extremely useful in improving the overall quality process.

8.6 Summary

– Testing is a dynamic method for verification and validation, where the software to be tested is executed with carefully designed test cases and the behavior of the software system is observed. A test case is a set of inputs and test conditions along with the expected outcome of testing. A test suite is a set of test cases that are generally executed together to test some specific behavior. During testing only the failures of the system are observed, from which the presence of faults is deduced; separate activities have to be performed to identify the faults and remove them.

– The intent of testing is to increase confidence in the correctness of the software. For this, the set of test cases used for testing should be such that for any defect in the system, there is likely to be a test case that will reveal it. To ensure this, it is important that the test cases are carefully designed with the intent of revealing defects.

– Due to the limitations of the verification methods for early phases, design and requirement faults also appear in the code. Testing is used to detect these errors also, in addition to the errors introduced during the coding phase. Hence, different levels of testing are often used for detecting defects injected during different stages. The commonly employed testing levels are unit testing, integration testing, system testing, and acceptance testing.

– For testing a software product, overall testing should be planned, and for testing each unit identified in the plan, test cases should be carefully designed to reveal errors and specified in a document or a test script.

– There are two approaches for designing test cases: black-box and white-box. In black-box testing, the internal logic of the system under testing is not considered and the test cases are decided from the specifications or the requirements. Equivalence class partitioning, boundary value analysis, and cause-effect graphing are examples of methods for selecting test cases for black-box testing. State-based testing is another approach in which the system is modeled as a state machine and then this model is used to select test cases using some transition or path-based coverage criteria. State-based testing can also be viewed as gray-box testing in that it often requires more information than just the requirements.

– In white-box testing, the test cases are decided based on the internal logic of the program being tested. Often a criterion is specified, but the procedure for selecting test cases to satisfy the criteria is left to the tester. The most common criteria are statement coverage and branch coverage.

– The main metric of interest during testing is the reliability of the software under testing. If defects are being logged, reliability can be assessed in terms of failure rate per week or day, though better models for estimation exist. Coverage achieved during testing, and defect removal efficiency, are other metrics of interest.

Self-Assessment Exercises

1. Define fault, error, and failure.
2. Suppose you have to test a procedure that takes two input parameters, does some computation with them, and then manipulates a global table, the manipulation itself depending on the state of the table. What will the complete specification of a test case for this procedure contain?
3. What are the different levels of testing and the goals of the different levels?
4. Suppose for logging defects, each defect will be treated as an object of a class Defect. Give the definition of this class.
5. Suppose a software has three inputs, each having a defined valid range. How many test cases will you need to test all the boundary values?

6. For boundary value analysis, if the strategy for generating test cases is to consider all possible combinations for the different values, what will be the set of test cases for a software that has three inputs X, Y, and Z?
7. Suppose a software has five different configuration variables that are set independently. If three of them are binary (have two possible values), and the rest have three values, how many test cases will be needed if pairwise testing method is used?
8. Consider a vending machine that takes quarters and when it has received two quarters, gives a can of soda. Develop a state model of this system, and then generate sets of test cases for the various criteria.
9. Consider a simple text formatter problem. Given a text consisting of words separated by blanks (BL) or newline (NL) characters, the text formatter has to covert it into lines, so that no line has more than MAXPOS characters, breaks between lines occur at BL or NL, and the maximum possible number of words are in each line. The following program has been written for this text formatter [41]:

```
alarm := false;
bufpos := 0;
fill := 0;
repeat
    inchar(c);
    if (c = BL) or (c = NL) or (c = EOF)
    then
        if bufpos != 0
        then begin
            if (fill + bufpos < MAXPOS) and (fill != 0)
            then begin
                outchar(BL);
                fill := fill + 1; end
            else begin
                outchar(NL);
                fill := 0; end;
            for k:=1 to bufpos do
                outchar(buffer[k]);
            fill := fill + bufpos;
            bufpos := 0; end
        else
            if bufpos = MAXPOS
            then alarm := true
            else begin
                bufpos := bufpos + 1;
                buffer[bufpos] := c; end
until alarm or (c = EOF);
```

For this program, do the following:

a) Select a set of test cases using the black-box testing approach. Use as many techniques as possible and select test cases for special cases using the "error guessing" method.
b) Select a set of test cases that will provide 100% branch coverage.

10. Suppose that the last round of testing, in which all the test suites were executed but no faults were fixed, took 7 full days (24 hours each). And in this testing, the number of failures that were logged every day were: 2, 0, 1, 2, 1, 1, 0. If it is expected that an average user will use the software for two hours each day in a manner that is similar to what was done in testing, what is the expected reliability of this software for the user?