```
        p := x;
        i := 1;
        while (i != y) do begin
                p := p * x;
                i := i + 1;
        end;
end;
```

Write formal specifications for a program to compute the product of two numbers. Then, using the axiomatic method, prove that this program is correct.

11. Consider the following two algorithms for searching an element E in a sorted array A, which contains n integers. The first procedure implements a simple linear search algorithm. The second performs a binary search. Binary search is generally much more efficient in terms of execution time compared to the linear search.

```
function lin_search (A, E): boolean
var
        i : integer;
        found: boolean;
begin
        found := false;
        i := 1;
        while (not found) and (i ≤ n) do begin
        if (A[i] = E) then found := true;
        i := i + 1;
        end;
        lin_search := found;
end;

function bin_search (A, E): boolean
var
        low, high, mid, i, j : integer;
        found : boolean;
begin
        low := 1;
        high := n;
        found := false;
        while (low ≤ high) and (not found) do begin
                mid := (low + high)/2;
                if E < A[mid] then high := mid - 1
                        else if E > A[mid] then low := mid + 1
```

```
                    else found := true;
            end;
            bin_search := found;
    end;
```

Determine the cyclomatic complexity and live variable complexity for these two functions. Is the ratio of the two complexity measures similar for the two functions?

12. What is Halstead's size measure for these two modules? Compare this size with the size measured in LOC.

13. Consider the size measure as the number of bytes needed to store the object code of a program. How useful is this size measure? Is it closer to LOC or Halstead's metric? Explain.

14. Not all control statements are equally complex. Assign complexity weights (0-10) to different control statements in Java, and then determine a formula to calculate the complexity of a program. How will you determine if this measure is better or worse than other complexity measures?

15. A combination of conditions in a decision makes a decision more complex. Such decisions should be treated as a combination of different decisions. Compared to the simple measure where each decision is treated as one, how much will the difference in the cyclomatic complexity of a program with 20% of its conditional statements having two conditions and 20% having three conditions be, when evaluated by this new approach?

16. Design an experiment to study the correlation between some of the complexity measures and between some of the size measures.

17. Design an experiment to study if the "error-proneness" of a module is related to a complexity measure for the module.

Case Studies

Implementation of Structured Design of Case Study 1

The programs were written in C on a Sun workstation, as required. The first version almost directly implemented the modules specified in the function-oriented design. The total size of the program was about 1320 lines. We determined various code based complexity and size metrics for this code using the tool `complexity` that we developed. This is shown below.

MODULE	SIZE	CYCLOMATIC COMPLEXITY
validate_file2	111	18
validate_dept_courses	88	17
sched_ug_pref	104	16
validate_class_rooms	92	15
validate_lec_times	84	15
print_conflicts	50	11
print_TimeTable	42	10
chk_fmt_time_slot	36	10
sched_pg_pref	82	9
separate_courses	46	9
Total Size: 1322	Total Cyclomatic Complexity:	243
Avg. size: 33	Avg. Cyclomatic Complexity:	6

From these metrics, it was clear that some of the modules were too large and had a high complexity value. Based on this information, we carefully reviewed some of these modules to see if their size or complexity could be reduced. During the reviews we found that in these modules some parts of the code were actually implementing some support functions that can be separated by forming clean, functionally cohesive modules.

As a result of this, a few new modules were formed. The complexity of many of the modules was reduced, and there was a general decline in the average complexity. It is worth noting that the total size and complexity is reduced by this exercise, besides the reduction in the complexity and size of the individual modules. That is, by this exercise we did not just redistribute the complexity, we actually reduced the overall complexity. The overall figures after the changes are:

```
------------------------------------------------------------
Total Size:   1264     Total Cyclomatic Complexity:  235
Avg.  Size:    30     Avg. Cyclomatic Complexity:    5
------------------------------------------------------------
```

OO Design Implementation of Case Study 1

The object-oriented design of the case study given earlier was implemented in C++. The implementation did extend the design a little, as is to be expected, but the extension was mostly in the addition of data members and some methods. No major design changes were required due to implementation issues. The code could have been analyzed by using some of the metrics and then modified, as was done in the code implementing the structured design. However, this was not done for this implementation for three reasons. First, we did not have tools to analyze the C++ programs. Secondly, some of the tools that were available for use through other sources worked on a different version of C++ (our implementation is in GNU C++). And finally, the OO metrics are still relatively new, and not much data about their use is available.

The C++ code for the case study is also available from the home page of the book.

Implementation of Case Study 2

This case study was implemented in Java on a PC. Some unit testing was done on some of the modules using Junit. The unit testing report is available from the Web site.

The entire code for this case study is also available from the Web site.

Chapter 10

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 will be 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. Testing is the activity where the errors remaining from all the previous phases must be detected. Hence, testing performs a very critical role for ensuring quality. The focus of this chapter is primarily on system testing in which the entire software system is tested, though testing is also performed on individual programs written by programmers and the concepts discussed are also applicable for individual program testing.

During testing, the software to be tested is executed with a set of test cases, and the behavior of the system for the test cases is evaluated to determine if the system is performing as expected. Clearly, the success of testing in revealing errors depends critically on the test cases. Much of this chapter is devoted to test case selection, criteria for selecting test cases, and their effect on testing.

We begin this chapter by discussing some definitions and concepts pertinent to testing. Then we discuss the two basic approaches to testing—black box or functional testing and white-box or structural testing. Aspects of testing process is discussed next, followed by a discussion on how testing data can be used for defect prevention. Then we discuss reliability estimation, as reliability is the main metric of interest during testing. This chapter ends with case studies.

10.1 Testing Fundamentals

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 can proceed, the need for oracles for testing, the importance of psychology of the tester, and

some desirable properties for the criteria used for testing. Once these are discussed, we will proceed with the issue of selection of test cases.

10.1.1 Error, Fault, and Failure

So far, we have used the intuitive meaning of the term *error* to refer to problems in requirements, design, or code. Sometimes error, fault, and failure are used interchangeably, and sometimes they refer to different concepts. Let us start by defining these concepts clearly. We follow the IEEE definitions [91] for these terms.

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 synonymous with the commonly used term *bug*. 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. It should be noted that the only faults that a software has are "design faults"; there is no wear and tear in software.

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 due to functional or performance reasons. A failure is produced only when there is a fault in the system. However, presence of a fault does not guarantee a failure. In other words, faults have the potential to cause failures and their presence is a necessary but not a sufficient condition for failure to occur. 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.

Note also that what is called a "failure" is dependent on the project, and its exact definition is often left to the tester or project manager. For example, is a misplaced line in the output a failure or not? Clearly, it depends on the project; some will consider it a failure and others will not. Take another example. If the output is not produced within a given time period, is it a failure or not? For a real-time system this may be viewed as a failure, but for an operating system it may not be viewed as a failure. This means that there can be no general definition of failure, and it is up to the project manager or end user to decide what will be considered a failure for reliability purposes. Note that in the example of a misplaced line, a defect might be recorded, and even corrected later, but its occurrence might not be considered a failure.

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 to occur. Whether a fault actually manifests itself in a certain time duration depends on many factors. This means that if we observe the behavior of a system for some time duration and 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 in this duration, we can say that there are some faults in the system.

There are direct consequences of this on testing. In testing, system behavior is observed, and by observing the behavior of a system or a component during testing, we determine whether or not there is a failure. Because of this fundamental reliance on behavior observation, testing can only reveal the presence of faults, not their absence. By observing failures of the system we can deduce the presence of faults; but by not observing a failure during our observation (or testing) interval we cannot claim that there are no faults in the system. An immediate consequence of this is that it becomes hard to decide for how long we should test a system without observing any failures before deciding to stop testing. This makes "when to stop testing" one of the hard issues in testing.

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, compared to methods that directly observe faults.

10.1.2 Test Oracles

To test any program, we need to have a description of its expected behavior and a method of determining whether the observed behavior conforms to the expected behavior. For this we need a *test oracle.*

A test oracle is a mechanism, different from the program itself, that can be used to check the correctness of the output of the program for the test cases. Conceptually, we can consider testing a process in which the test cases are given to the test oracle and the program under testing. The output of the two is then compared to determine if the program behaved correctly for the test cases, as shown in Figure 10.

Test oracles are necessary for testing. Ideally, we would like an automated oracle, which always gives a correct answer. However, often the oracles are human beings, who can make mistakes. As a result, when there is a discrepancy between the results of the program and the oracle, we have to verify the result produced by the oracle, before declaring that there is a fault in the program.

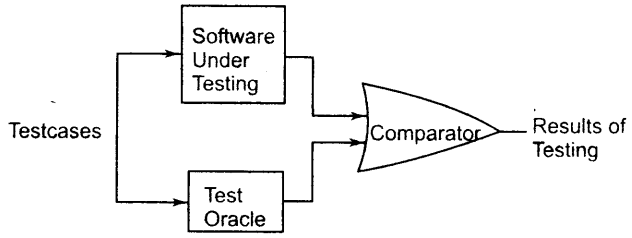

Figure 10.1: Testing and test oracles.

The human oracles generally use the specifications of the program to decide what the "correct" behavior of the program should be. However, the specifications themselves may contain errors, be imprecise, or contain ambiguities. Such shortcomings in the specifications are the major cause of situations where one party claims that a particular condition is not a failure while the other claims it is. There is no easy solution to this problem, as testing does require some specifications against which the given system is tested.

There are some systems where oracles are automatically generated from specifications of programs or modules. With such oracles, we are assured that the output of the oracle is consistent with the specifications. These oracles also eliminate the effort of determining the expected behavior for a test case. However, even this approach does not solve all our problems, because of the possibility of errors in the specifications. Consequently, an oracle generated from the specifications will only produce correct results if the specifications are correct, and it will not be dependable in the case of specification errors. Furthermore, such systems that generate oracles from specifications are likely to require formal specifications, which are frequently not generated during design.

10.1.3 Test Cases and Test Criteria

Having test cases that are good at revealing the presence of faults is central to successful testing. The reason for this is that if there is a fault in a program, the program can still provide the expected behavior for many inputs. Only for the set of inputs that exercise the fault in the program will the output of the program deviate from the expected behavior. Hence, it is fair to say that testing is as good as its test cases.

Ideally, we would like to determine a set of test cases such that successful execution of all of them implies that there are no errors in the program. This ideal goal cannot usually be achieved due to practical and theoretical constraints. Each test case costs money, as effort is needed to generate the test case, machine time is needed to execute the program for that test case, and more effort is needed to evaluate the results. Therefore,

we would also like to minimize the number of test cases needed to detect errors. These are the two fundamental goals of a practical testing activity—maximize the number of errors detected and minimize the number of test cases (i.e., minimize the cost). As these two are frequently contradictory, the problem of selecting the set of test cases with which a program should be tested becomes more complex.

While selecting test cases the primary objective is to ensure that if there is an error or fault in the program, it is exercised by one of the test cases. An ideal test case set is one that succeeds (meaning that its execution reveals no errors) only if there are no errors in the program. One possible ideal set of test cases is one that includes all the possible inputs to the program. This is often called *exhaustive* testing. However, exhaustive testing is impractical and infeasible, as even for small programs the number of elements in the input domain can be extremely large.

So, how should we select our test cases? On what basis should we include some element of the program domain in the set of test cases and not include others? For this *test selection criterion* (or simply *test criterion*) can be used. For a given program P and its specifications S, a test selection criterion specifies the conditions that must be satisfied by a set of test cases T. The criterion becomes a basis for test case selection. For example, if the criterion is that all statements in the program be executed at least once during testing, then a set of test cases T satisfies this criterion for a program P if the execution of P with T ensures that each statement in P is executed at least once.

There are two fundamental properties for a testing criterion: reliability and validity [73]. A criterion is reliable if all the sets (of test cases) that satisfy the criterion detect the same errors. That is, it is insignificant which of the sets satisfying the criterion is chosen; every set will detect exactly the same errors. A criterion is valid if for any error in the program there is some set satisfying the criterion that will reveal the error. A fundamental theorem of testing is that if a testing criterion is valid and reliable, if a set satisfying the criterion succeeds (revealing no faults), then the program contains no errors [73]. However, it has been shown that no algorithm exists that will determine a valid criterion for an arbitrary program.

Getting a criterion that is reliable and valid and that can be satisfied by a manageable number of test cases is usually not possible. So, often criteria are chosen that are not valid or reliable like "90% of the statements should be executed at least once." Often a criterion is not even clearly specified, as in "all special values in the domain must be included" (what is a "special value"?).

Even when the criterion is specified, generating test cases to satisfy a criterion is not simple. In general, generating test cases for most of the criteria cannot be automated. For example, even for a simple criterion like "each statement of the program should be executed," it is extremely hard to construct a set of test cases that will satisfy this criterion for a large program, even if we assume that all the statements can be executed (i.e., there is no part that is not reachable).

A criterion C_1 includes (or subsumes) the criterion C_2 if for every program P and its specification S, any set of test cases that satisfy C_1 also satisfy C_2 [145, 67]. This relation is represented as $C_1 \Rightarrow C_2$, and is a transitive relation. One may think that if $C_1 \Rightarrow C_2$, testing based on C_1 will always be better than testing based on C_2. Unfortunately, this is not the case. The reason is that the fault-detection capability of a set of test cases T that satisfy a criterion C depends on the actual test cases in T and not just C (i.e., the criterion is not valid). In other words, if T_1 and T_2 both satisfy C for a program P, it does not mean that T_1 and T_2 will execute the same paths of P and detect the same faults in P. Because the actual test cases also play a role in whether or not an error in a program is detected, in general, it is possible to have a situation where $C_1 \Rightarrow C_2$, T_1 satisfies C_1, T_2 satisfies C_2, but T_2 detects an error that T_1 does not. However, if similar methods are used for test case generation then, generally speaking, C_1 will be better for testing than C_2 if $C_1 \Rightarrow C_2$.

The intent of the preceding discussion is to illustrate that no single criterion will serve the purpose of detecting a reasonable number of errors in a program. Though frequently the focus is on the criterion, to use a criterion for testing, the strategy for generating test cases to satisfy a criterion is also important. As it is generally known that all the faults in a program cannot be practically revealed by testing, and due to the limitations of the test criterion, it is best that during testing more than one criterion be used.

10.1.4 Psychology of Testing

As we have seen, 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.

The 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; but the intent should be to show that a program does not work. With this in mind we can define testing as the process of executing a program with the intent of finding errors [121].

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 adver-

sary that must be beaten by the tester by showing the presence of errors. With this in mind, a test case is "good" if it detects an as-yet-undetected error in the program, and our goal during designing test cases should be to design such "good" test cases.

One of the reasons many organizations require a product to be tested by people not involved with developing the program before finally delivering it to the customer is this psychological factor. It is hard to be destructive to something we have created ourselves, and we all like to believe that the program we have written "works." So, it is not easy for someone to test his own program with the proper frame of mind for testing. Another reason for independent testing is that sometimes errors occur because the programmer did not understand the specifications clearly. Testing of a program by its programmer will not detect such errors, whereas independent testing may succeed in finding them.

This approach towards testing is suitable for earlier stages of testing, where indeed the objective is to reveal errors. However, often the last stages of testing are meant more for evaluating the product. In these types of testing, test cases are selected primarily to mimic the user behavior or user scenarios.

10.2 Black-Box Testing

There are two basic approaches to 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 in functional testing 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 as we have stated, 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. In fact, there are no precise criteria for selecting test cases. 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.

10.2.1 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 [121] 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 an alphabet, 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 rates—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 are 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 10.

Input	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
s	EQ1: Contains numbers EQ2: Contains lower case letters EQ3: Contains upper case 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

Table 10.1: Valid and invalid equivalence classes.

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 lower case, upper case, 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 4 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 5. This covers EQ1 and EQ6. Then we will need test cases for EQ2 through EQ5, and separate test cases for IEQ1 through IEQ3.

10.2.2 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 the boundary value analysis. In boundary value analysis [121], 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 $0.0 \leq x \leq 1.0$, 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 6 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 10.

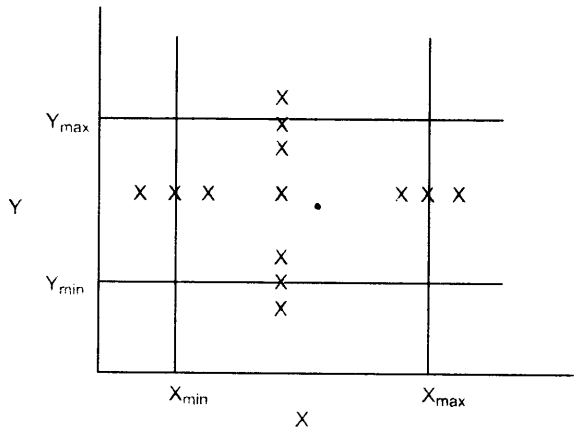

Figure 10.2: Test cases for BVA.

A second strategy is to try all possible combinations for the values for the different variables. As there are 7 values for each variable (6 boundary values and one nominal value), if there are n variables, there will be a total of 7^n test cases.

10.2.3 Cause-Effect Graphing

One weakness with the equivalence class partitioning and boundary value methods is that they consider each input separately. That is, both concentrate on the conditions and classes of one input. They do not consider combinations of input circumstances that may form interesting situations that should be tested. One way to exercise combinations of different input conditions is to consider all valid combinations of the equivalence classes of input conditions. This simple approach will result in an unusually large number of test cases, many of which will not be useful for revealing any new errors. For example, if there are n different input conditions, such that any combination of the input conditions is valid, we will have 2^n test cases.

Cause-effect graphing [121] is a technique that aids in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large. The technique starts with identifying causes and effects of the system under testing. A *cause* is a distinct input condition, and an *effect* is a distinct output condition. Each condition forms a node in the cause-effect graph. The conditions should be stated such that they can be set to either true or false. For example, an input condition can be "file is empty," which can be set to true by having

an empty input file, and false by a nonempty file. After identifying the causes and effects, for each effect we identify the causes that can produce that effect and how the conditions have to be combined to make the effect true. Conditions are combined using the Boolean operators "and," "or," and "not," which are represented in the graph by &, |, and $\sim$. Then for each effect, all combinations of the causes that the effect depends on which will make the effect true are generated (the causes that the effect does not depend on are essentially "don't care"). By doing this, we identify the combinations of conditions that make different effects true. A test case is then generated for each combination of conditions, which make some effect true.

Let us illustrate this technique with a small example. Suppose that for a bank database there are two commands allowed:

credit acct_number transaction_amount
debit acct_number transaction_amount

The requirements are that if the command is credit and the acct_number is valid, then the account is credited. If the command is debit, the acct_number is valid, and the transaction_amount is valid (less than the balance), then the account is debited. If the command is not valid, the account number is not valid, or the debit amount is not valid, a suitable message is generated. We can identify the following causes and effects from these requirements:

Causes:

c1. Command is credit
c2. Command is debit
c3. Account number is valid
c4. Transaction_amt is valid

Effects:

e1. Print "invalid command"
e2. Print "invalid account_number"
e3. Print "Debit amount not valid"
e4. Debit account
e5. Credit account

The cause-effect of this is shown in Figure 10. In the graph, the cause-effect relationship of this example is captured. For all effects, one can easily determine the causes each effect depends on and the exact nature of the dependency. For example, according to this graph the effect e5 depends on the causes c2, c3, and c4 in a manner such that the effect e5 is enabled when all c2, c3, and c4 are true. Similarly, the effect e2 is enabled if c3 is false.

From this graph, a list of test cases can be generated. The basic strategy is to set an effect to 1 and then set the causes that enable this condition. The condition of causes forms the test case. A cause may be set to false, true, or don't care (in the case when the effect does not depend at all on the cause). To do this for all the effects, it is convenient to use a decision table. The decision table for this example is shown in Figure 10.

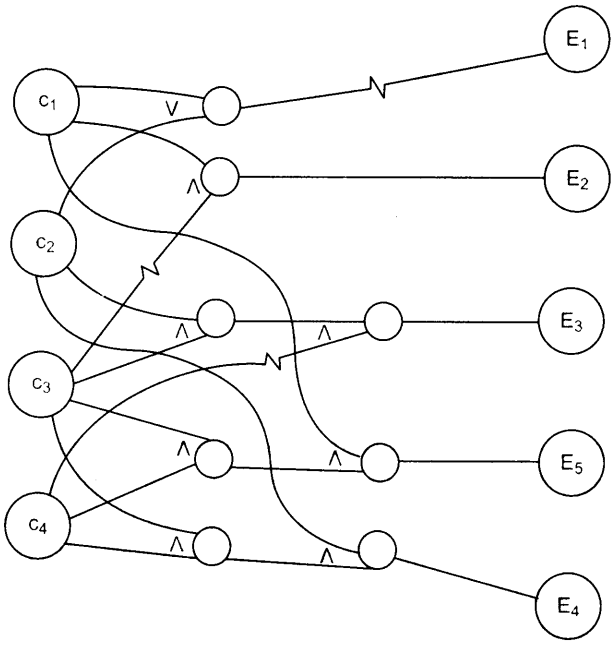

Figure 10.3: The cause-effect graph.

This table lists the combinations of conditions to set different effects. Each combination of conditions in the table for an effect is a test case. Together, these condition combinations check for various effects the software should display. For example, to test for the effect e3, both c2 and c4 have to be set. That is, to test the effect "Print debit amount not valid," the test case should be: Command is debit (setting c2 to True), the account number is valid (setting c3 to False), and the transaction money is not proper (setting c4 to False).

Cause-effect graphing, beyond generating high-yield test cases, also aids the understanding of the functionality of the system, because the tester must identify the distinct causes and effects. There are methods of reducing the number of test cases generated by proper traversing of the graph. Once the causes and effects are listed and their dependencies specified, much of the remaining work can also be automated.

SNo.	1	2	3	4	5
c1	0	1	x	x	1
C2	0	x	1	1	x
c3	x	0	1	1	1
c4	x	x	0	1	1
e1	1				
e2		1			
e3			1		
e4				1	
e5					1

Figure 10.4: Decision table for the cause-effect graph.

10.2.4 Pair-wise 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 single-mode fault [125]. Simple examples of single mode fault are a software not able to print for a particular type of printer, a software that cannot compute fare properly when the traveller is a minor, 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 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 night time calling (one parameter) to a particular country (another parameter). Or an airline ticketing system that has incorrect behavior when a minor (one parameter) is travelling 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 one 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 an interaction of pair of values [40]. 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 is exercised. This is called *pair-wise testing.*

In pair-wise 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 are $m * m * n * (n - 1)/2$.

The objective of pair-wise 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 pair-wise 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 10, along with the pairs they cover.

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)

Table 10.2: Test cases for pair-wise testing.

As should be clear, generating test cases to cover all the pairs is not a simple task. The minimum set of test cases are those 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 many situations where manual generation is feasible, the following approach can be followed. Start with one combination of parameter values. Keep adding new combinations, choosing values such that no two values exist together in any earlier test case, until all pairs are covered. When selecting such values is not possible, select the values that has the fewest values that have existed together in an earlier test case. Essentially we are generating a test case that can cover as many as 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 pair-wise testing exist. In [40] 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!

Pair-wise 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.

10.2.5 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.*

The 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.

10.2.6 State-Based Testing

There are some systems that are essentially state-less 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 the state of the system, that is, 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 [34].

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 [16]. 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 4. 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 last architecture given in Figure 4.11, which had a cache between the server and the database, and in which the survey and results are cached and updated only after 5 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 six 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 is goes to state 2. The

state model for this system is shown in Figure 10 (i represents an input from the user for taking the survey).

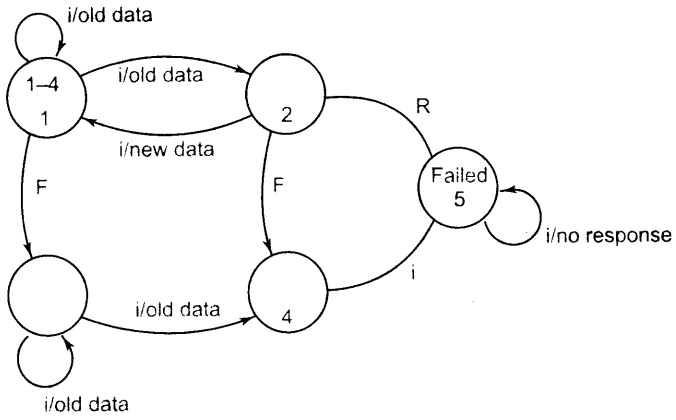

Figure 10.5: State model for the student survey system.

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 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 [123]. 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 of 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 10. 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.

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

Table 10.3: Test cases for a state based testing criteria.

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.

10.3 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 three different approaches to structural testing: control flow-based testing, data flow-based testing, and mutation testing.

10.3.1 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 [129]. 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 criteria 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 [129].

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 little 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 [129]. Branch coverage implies statement coverage, as each statement is a part of some branch. In other words, $C_{branch} \Rightarrow C_{stmt}$. 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. It should be clear that $C_{path} \Rightarrow C_{branch}$.

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.

Another such approach based on the cyclomatic complexity has been proposed in [116]. The test criterion is that if the cyclomatic complexity of a module is V, then at least V distinct paths must be executed during testing. We have seen that cyclomatic complexity V of a module is the number of independent paths in the flow graph of a module. As these are independent paths, all other paths can be represented as a combination of these basic paths. These basic paths are finite, whereas the total number of paths in a module having loops may be infinite.

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.

10.3.2 Data Flow-Based Testing

Now we discuss some criteria that select the paths to be executed during testing based on data flow analysis, rather than control flow analysis. In the previous chapter, we discussed use of data flow analysis for static testing of programs. In the data flow-based testing approaches, besides the control flow, information about where the variables are defined and where the definitions are used is also used to specify the test cases. The basic idea behind data flow-based testing is to make sure that during testing, the definitions of variables and their subsequent use is tested. Just like the all-nodes and all-edges criteria try to generate confidence in testing by making sure that at least all statements and all branches have been tested, the data flow testing tries to ensure some coverage of the definitions and uses of variables. Approaches for use of data flow information have been proposed in [109, 129]. Our discussion here is based on the family of data flow-based testing criteria that were proposed in [129]. We discuss some of these criteria here.

For data flow-based criteria, a *definition-use graph* (*def/use* graph, for short) for the program is first constructed from the control flow graph of the program. A statement in a node in the flow graph representing a block of code has variable occurrences in it. A variable occurrence can be one of the following three types [129]:

- *def* represents the definition of a variable. The variable on the left-hand side of an assignment statement is the one getting defined.
- *c-use* represents computational use of a variable. Any statement (e.g., read, write, an assignment) that uses the value of variables for computational purposes is said to be making c-use of the variables. In an assignment statement, all variables on the right-hand side have a c-use occurrence. In a read and a write statement, all variable occurrences are of this type.
- *p-use* represents predicate use. These are all the occurrences of the variables in a predicate (i.e., variables whose values are used for computing the value of the predicate), which is used for transfer of control.

Based on this classification, the following can be defined [129]. Note that c-use variables may also affect the flow of control, though they do it indirectly by affecting the value of the p-use variables. Because we are interested in the flow of data between nodes, a c-use of a variable x is considered *global c-use* if there is no def of x within the block preceding the c-use. With each node i, we associate all the global c-use variables in that node. The p-use is associated with edges. If $x_1, x_2, ..., x_n$ had p-use occurrences in the

statement of a block from where two edges go to two different blocks j and k (e.g., with an `if then else`), then $x_1, ..., x_n$ are associated with the two edges (i, j) and (i, k).

A path from node i to node j is called a *def-clear* path with respect to (w.r.t.) a variable x if there is no def of x in the nodes in the path from i to j (nodes i and j may have a def). Similarly, a def-clear path w.r.t. x from a node i to an edge (j, k) is one in which no node on the path contains a definition of x. A def of a variable x in a node i is a *global def*, if it is the last def of x in the block being represented by i, and there is a def-clear path from i to some node with a global c-use of x. Essentially, a def is a global def if it can be used outside the block in which it is defined.

The def/use graph for a program P is constructed by associating sets of variables with edges and nodes in the flow graph. For a node i, the set *def(i)* is the set of variables for which there is a global def in the node i, and the set *c-use(i)* is the set of variables for which there is a global c-use in the node i. For an edge (i, j), the set *p-use(i, j)* is the set of variables for which there is a p-use for the edge (i, j).

Suppose a variable x is in $def(i)$ of a node i. Then, $dcu(x, i)$ is the set of nodes, such that each node has x in its c-use, $x \in def(i)$, and there is a def-clear path from i to j. That is, dcu(x, i) represents all those nodes in which the (global) c-use of x uses the value assigned by the def of x in i. Similarly, $dpu(x, i)$ is the set of edges, such that each edge has x in its p-use, $x \in def(i)$, and there is a def-clear path from i to (j, k). That is, dpu(x, i) represents all those edges in which the p-use of x uses the value assigned by the def of x in i.

Based on these definitions proposed in [129], a family of test case selection criteria was proposed in [129], a few of which we discuss here. Let G be the def/use graph for a program, and let P be a set of complete paths of G (i.e., path representing a complete execution of the program). A test case selection criterion defines the contents of P.

P satisfies the *all-defs* criterion if for every node i in G and every x in $def(i)$, P includes a def-clear path w.r.t. x to some member of $dcu(x, i)$ or some member of $dpu(x, i)$. This criterion says that for the def of every variable, one of its uses (either p-use or c-use) must be included in a path. That is, we want to make sure that during testing the use of the definitions of all variables is tested.

The *all-p-uses* criterion requires that for every $x \in def(i)$, P include a def-clear path w.r.t. x from i to some member of $dpu(x, i)$. That is, according to this criterion all the p-uses of all the definitions should be tested. However, by this criterion a c-use of a variable may not be tested. The *all-p-uses, some-c-uses* criterion requires that all p-uses of a variable definition must be exercised, and some c-uses must also be exercised. Similarly, the *all-c-uses, some-p-uses* criterion requires that all c-uses of a variable definition be exercised, and some p-uses must also be exercised.

The *all-uses* criterion requires that all p-uses and all c-uses of a definition must be exercised. That is, the set P must include, for every node i and every $x \in def(i)$, a def-clear path w.r.t. x from i to all elements of $dcu(x, i)$ and to all elements of $dpu(x, i)$. A few other criteria have been proposed in [129].

In terms of the number of test cases that might be needed to satisfy the data flow-based criteria, it has been shown that though the theoretical limit on the size of the test case set is up to quadratic in the number of two-way decision statements in the program, the actual number of test cases that satisfy a criterion is quite small in practice [146]. Empirical observation in [146] seems to suggest that in most cases the number of test cases grows linearly with the number of two-way decisions in the program.

As mentioned earlier, a criterion C_1 includes another criterion C_2 (represented by $C_1 \Rightarrow C_2$) if any set of test cases that satisfy criterion C_1 also satisfy the criterion C_2. The inclusion relationship between the various data flow criteria and the control flow criteria is given in Figure 10 [129].

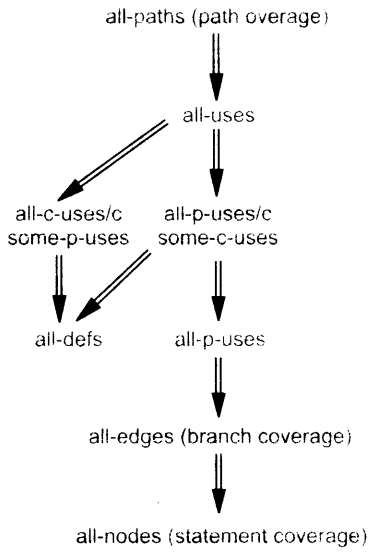

Figure 10.6: Relationship between different criteria.

It should be quite clear that all-paths will include all-uses and all other structure-based criteria. All-uses, in turn, includes all-p-uses, all-defs, and all-edges. However, all-defs does not include all-edges (and the reverse is not true). The reason is that all-defs is focusing on all definitions getting used, while all-edges is focusing on all decisions evaluating to both true and false. For example, a decision may evaluate to true and false in two different test cases, but the use of a definition of a variable x may not have been exercised. Hence, the all-defs and all-edges criteria are, in some sense, incomparable.

As mentioned earlier, inclusion does not imply that one criterion is always better than another. At best, it means that if the test case generation strategy for two criteria C_1 and C_2 is similar, and if $C_1 \Rightarrow C_2$, then statistically speaking, the set of test cases satisfying C_1 will be better than a set of test cases satisfying C_2. The experiments

reported in [67] show that no one criterion (out of a set of control flow-based and data flow-based criteria) does significantly better than another consistently. However, it does show that testing done by using all-branch or all-uses criterion generally does perform better than randomly selected test cases.

10.3.3 An Example

Let us illustrate the use of some of the control flow-based and data flow-based criteria through the use of an example. Consider the following example of a simple program for computing x^y for any integer x and y [129]:

```
1. scanf(x, y); if (y < 0)
2.      pow = 0 - y;
3. else pow = y;
4. z = 1.0;
5. while (pow != 0)
6.      { z = z * x; pow = pow - 1; }
7. if (y < 0)
8.      z = 1.0/z;
9. printf(z);
```

The def/use graph for this program is given in the Figure 10 [129]. In the graph, the line numbers given in the code segment are used to number the nodes (each line contains all the statements of that block). For each node, the def set (i.e., the set of variables defined in the block) and the c-use set (i.e., the set of variables that have a c-use in the block) are given along with the node. For each edge, if the p-use set is not empty, it is given in the graph.

The various sets are easily determined from the block of code representing a node. To determine the dcu and dpu the graph has to be traversed. The dcu for various node and variable combination is given next:

(node, var)	dcu	dpu
(1, x)	{6}	ϕ
(1, y)	{2, 3}	{(1,2), (1,3), (7, 8), (7, 9)}
(2, pow)	{6}	{(5, 6), (5, 7)}
(3, pow)	{6}	{(5, 6), (5, 7)}
(4, z)	{6, 8, 9}	ϕ
(6, z)	{6, 8, 9}	ϕ
(6, pow)	{6}	{(5, 6), (5, 7)}
(8, z)	{9}	ϕ

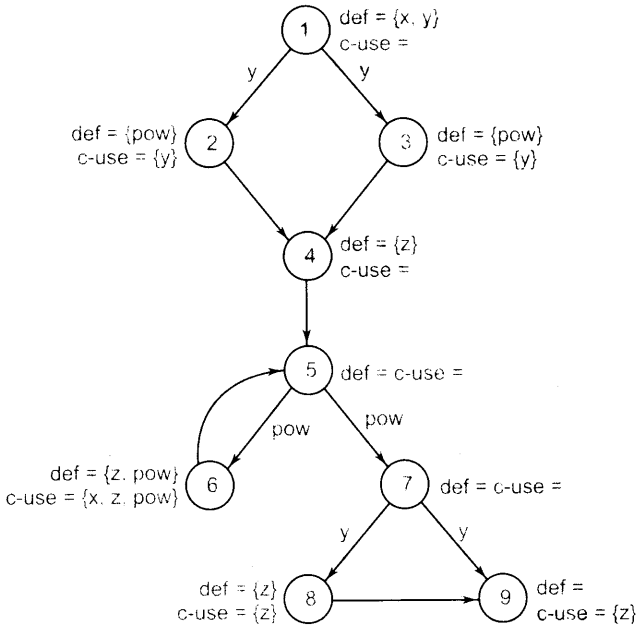

Figure 10.7: def/use graph for the example.

Now let us discuss the issue of generating test cases for this program using various criteria. We can divide the problem of test case selection into two parts. First we identify some paths that together satisfy the chosen criterion. Then we identify the test cases that will execute those paths. As the first issue is more relevant when discussing coverage criteria, frequently in testing literature only the paths that satisfy the criterion are discussed. While selecting paths that satisfy a given coverage brings us to the question of whether the path is *feasible*, that is, if it is possible to have some test data that will execute that path. It is known that a program may contain paths that are not feasible. A simple example is in a program with a `for` loop. In such a program, no path that executes the loop fewer than the number of times specified by the `for` loop is feasible. In general, the issue of feasibility of paths cannot be solved algorithmically, as the problem is undecidable. However, the programmer can use his judgment and knowledge about the program to decide whether or not a particular path is infeasible. With the presence of infeasible paths, it is not possible to fully satisfy the criterion like all-uses, and the programmer will have to use his judgment to avoid considering the infeasible paths.

Let us first consider the all-edges criterion, which is the same as 100% branch coverage. In this we want to make sure that each edge in the graph is traversed during

testing. For this, if the paths executed by the test cases include the following paths, we can see that all edges are indeed covered:

$$(1; 2; 4; 5; 6; 7; 8; 9), (1; 3; 4; 5; 7; 9)$$

Here we could have chosen a set of paths with (1; 2; 4; 5; 6; 7; 9) as one of them. But a closer examination of the program will tell us that this path is not feasible, as going from 1 to 2 implies that y is negative, which in turn implies that from 7 we must go to 8 and cannot go directly to 9. As can be seen even from this simple example, it is very easy to have paths that are infeasible. To execute the selected paths (or paths that include these paths), the following two test cases will suffice: $(x = 3, y = 1)$ and $(x = 3, y = -1)$. That is, a set consisting of these two test cases will satisfy the all-edges criterion.

Now let us consider the all-defs criterion, which requires that for all definitions of all variables, at least one use (c-use or p-use) must be exercised during testing. First let us observe that the set of paths given earlier for the all-edges criterion does not satisfy the all-uses criterion. The reason is that to satisfy all-uses, we must have some path in which the defs in node 6 (i.e., for z and pow) also get used. As the only way to get the def of pow in node 6 to be used is to visit 6 again, these paths fail to satisfy the criterion. The following set of paths will satisfy the all-defs criterion:

$$(1; 2; 4; 5; 6; 5; 6; 7; 8; 9), (1; 3; 4; 5; 6; 7; 9)$$

Let's consider the first path in this. The prefix 1; 2; 4; 5; 6; ensures that all the defs of nodes 1, 2, and 4 have been used. Having another 5; 6 after this ensures that the defs in node 6 are used. This is not needed by the branch coverage, but it comes because of the def-use constraints. It can also be easily seen that the set of test cases selected for the branch coverage will not suffice here. The following two test cases will satisfy the criteria: $(x = 3, y = 4)$ and $(x = 3, y = -2)$.

Let us finally consider the all-uses criterion, which requires that all p-uses and all c-uses of all variable definitions be tried during testing. In other words, we have to construct a set of paths that include a path from any node having a def to all nodes in its dcu and its dpu. The dcu and dpu sets for all nodes were given earlier. In this example, as it turns out, the paths given earlier for all-defs also satisfy the all-uses criterion. Hence, the corresponding test cases will also suffice. We leave the details of this as an exercise for the reader.

10.3.4 Mutation Testing

Mutation testing is another structural testing technique that differs fundamentally from the approaches discussed earlier. In control flow-based and data flow-based testing, the focus was on which paths to execute during testing. Mutation testing does not take a

path-based approach. Instead, it takes the program and creates many mutants of it by making simple changes to the program. The goal of testing is to make sure that during the course of testing, each mutant produces an output different from the output of the original program. In other words, the mutation testing criterion does not say that the set of test cases must be such that certain paths are executed; instead it requires the set of test cases to be such that they can distinguish between the original program and its mutants. The description of mutation testing given here is based on [50, 115].

In hardware, testing is based on some fault models that have been developed and that model the actual faults closely. The fault models provide a set of simple faults, combination of which can model any fault in the hardware. In software, however, no such fault model exists. That is why most of the testing techniques try to guess where the faults might lie and then select the test cases that will reveal those faults. In mutation testing, faults of some pre-decided types are introduced in the program being tested. Testing then tries to identify those faults in the mutants. The idea is that if all these "faults" can be identified, then the original program should not have these faults; otherwise they would have been identified in that program by the set of test cases.

Clearly this technique will be successful only if the changes introduced in the main program capture the most likely faults in some form. This is assumed to hold due to the *competent programmer hypothesis* and the *coupling effect.* The competent programmer hypothesis says that programmers are generally very competent and do not create programs at random, and for a given problem, a programmer will produce a program that is very "close" to a correct program. In other words, a correct program can be constructed from an incorrect program with some minor changes in the program. The coupling effect says that the test cases that distinguish programs with minor differences with each other are so sensitive that they will also distinguish programs with more complex differences. In [115], some experiments are cited in which it has been shown that the test data that can distinguish mutants created by simple changes can also distinguish up to 99% of the mutants that have been created by applying a series of simple changes.

Now let us discuss the mutation testing approach in a bit more detail. For a program under test P, mutation testing prepares a set of *mutants* by applying *mutation operators* on the text of P. The set of mutation operators depends on the language in which P is written. In general, a mutation operator makes a small unit change in the program to produce a mutant. Examples of mutation operators are: replace an arithmetic operator with some other arithmetic operator, change an array reference (say, from A to B), replace a constant with another constant of the same type (e.g., change a constant to 1), change the label for a goto statement, and replace a variable by some special value (e.g., an integer or a real variable with 0). Each application of a mutation operator results in one mutant. As an example, consider a mutation operator that replaces an arithmetic operator with another one from the set $\{+, -, *, **, /\}$. If a program P

contains an expression

$$a = b * (c - d),$$

then this particular mutation operator will produce a total of eight mutants (four by replacing '*' and four by replacing '-'). The mutation operators that make exactly one syntactic change in the program to produce a mutant are said to be of *first order*. If the coupling effect holds, then the first-order mutation operators should be sufficient, and there is no need for higher-order mutation operators.

Mutation testing of a program P proceeds as follows. First a set of test cases T is prepared by the tester, and P is tested by the set of test cases in T. If P fails, then T reveals some errors, and they are corrected. If P does not fail during testing by T, then it could mean that either the program P is correct or that P is not correct but T is not sensitive enough to detect the faults in P. To rule out the latter possibility (and therefore to claim that the confidence in P is high), the sensitivity of T is evaluated through mutation testing and more test cases are added to T until the set is considered sensitive enough for "most" faults. So, if P does not fail on T, the following steps are performed [115]:

1. Generate mutants for P. Suppose there are N mutants.
2. By executing each mutant and P on each test case in T, find how many mutants can be distinguished by T. Let D be the number of mutants that are distinguished: such mutants are called *dead*.
3. For each mutant that cannot be distinguished by T (called a *live* mutant), find out which of them are equivalent to P. That is, determine the mutants that will always produce the same output as P. Let E be the number of equivalent mutants.
4. The *mutation score* is computed as $D/(N - E)$.
5. Add more test cases to T and continue testing until the mutation score is 1.

In this approach, for the mutants that have not been distinguished by T, their equivalence with P has to be determined. As determining the equivalence of two programs is undecidable, this cannot be done algorithmically and will have to be done manually (tools can be used to aid the process). There are many situations where this can be determined easily. For example, if a condition $x <= 0$ (in a program to compute the absolute value, say) is changed to $x < 0$, we can see immediately that the mutant produced through this change will be equivalent to the original program P, as it does not matter which path the program takes when the value of x is 0. In other situations, it may be very hard to determine equivalence. One thing is clear: the tester will have to compare P with all the live mutants to determine which are equivalent to P. This analysis can then be used to add further test cases to T in an attempt to kill those live mutants that are not equivalent.

Determining test cases to distinguish mutants from the original program is also not easy. In an attempt to form a test case to kill a mutant, a tester will have to examine the mutant (and the original program) and then reason which test case is likely to distinguish the mutant. This can be a complex exercise, depending on the complexity of the program being tested and the exact nature of the difference between the mutant and the original program. Suppose that a statement at line l of the program P has been mutated to produce the mutant M. The first property that a test case t needs to have to distinguish M and P is that the test case should force the execution to reach the statement at l. The test case t should also be such that after execution of the statement at l, different states are reached by P and M. Before reaching l, the state while executing the programs P and M will be the same as the programs are same until l. If the test case is such that after executing the statement at l, the execution of the programs P and M either takes a different path or the values in the state are different, then there is a possibility that this difference will be manifested in output being different. If the state after executing the statement at l continues to be the same in P and M, we will not be able to distinguish P and M. Finally, t should be such that when P and M terminate, their states are different (assuming that P and M output their complete state at the end only). As one can imagine, constructing a test case that will satisfy these three properties is not going to be, in general, an easy task.

Finally, let us discuss the issue of detecting errors in the original program P, which is one of the basic goals of testing. In mutation testing, errors in the original program are frequently revealed when test cases are being designed to distinguish mutants from the original program. If no errors are detected and the mutation score reaches 1, then the testing is considered *adequate* by the mutation testing criterion. It should be noted that even if no errors have been found in the program under test during mutation testing, the confidence in the testing increases considerably if the mutation score of 1 is achieved, as we know that the set of test case with which P has been tested has been able to kill all (nonequivalent) mutants of P. This suggests that if P had an error, one of its mutants would have been closer to the correct program, and then the test case that distinguished the mutant from P would have also revealed that P is incorrect (it is assumed that the output of all test cases are evaluated to see if P is behaving correctly).

One of the main problems of mutation testing relates to its performance. The number of mutants that can be generated by applying first-order mutation operators is quite large and depends on the language and the size of the mutation operator set. For a FORTRAN program containing L lines of code to which the mutation operator can be applied, the total number of mutants is of the order of L^2 [115]. These many programs have to be compiled and executed on the selected test case set. This requires an enormous amount of computer time. For example, for a 950-line program, it was estimated that a total of about 900,000 mutants will be produced, the testing of which would take more than 70,000 hours of time on a Sun SPARC station [115]. Further, the tester might have to spend considerable time, as he will have to examine many mutants,

besides the original program, to determine whether or not they are equivalent. These performance issues make mutation testing impractical for large programs.

10.3.5 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 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.

Tools for data flow-based testing and mutation testing are even more complex. Some tools have been built for aiding data flow-based testing [66, 81]. A data flow testing tool has to keep track of definitions of variables and their uses, besides keeping track of the control flow graph. For example, the ASSET tool for data flow testing [66] first analyzes a Pascal program unit to determine all the definition-use associations. It then instruments the program so that the paths executed during testing are recorded. After the program has been executed with the test cases, the recorded paths are evaluated for satisfaction of the chosen criterion using the definition-use associations generated earlier. The list of definition-use associations that have not yet been executed is also output, which can then be used by the tester to select further test cases.

It should be pointed out that when testing a complete program that consists of many modules invoked by each other, the presence of procedures considerably complicates data flow testing. The main reason is that the presence of global variable creates def-use pairs in which the statements may exist in different procedures, e.g., a (global) variable may be defined in one procedure and then used in another. To use data flow-based testing on complete programs (rather than just modules), inter-procedural data flow analysis will be needed. Though some methods have been developed for performing data flow-based testing on programs with procedures [83], the presence of multiple procedures complicates data flow-based testing. It should be noted that this problem does not arise with statement coverage and branch coverage, where there are no special linkages between modules. The statement or branch coverage of a program can be computed simply from the statement or branch coverage of its modules. This is one of the reasons for the popularity of these coverage measures and tools.

In mutation testing, the tool is generally given a program P and a set of test cases T. The tool has to first use the mutation operations for the language in which P is written to produce the mutants. Then P and all the mutants and P are executed with T. Based on the output of different programs, the mutation score, and the number and identity of dead and live mutants are determined and reported to the tester. The score tells the tester the quality of T according to the mutation criterion, and the set of live mutants give the feedback to the tester for selecting further test cases to increase the mutation score. Some mutation testing tools have also been built [27, 49].

10.4 Testing Process

The basic goal of the software development process is to produce software that has no errors or very few errors. In an effort to detect errors soon after they are introduced, each phase ends with a verification activity such as a review. However, most of these verification activities in the early phases of software development are based on human evaluation and cannot detect all the errors. This unreliability of the quality assurance activities in the early part of the development cycle places a very high responsibility on testing. In other words, as testing is the last activity before the final software is delivered, it has the enormous responsibility of detecting any type of error that may be in the software.

Furthermore, we know that software typically undergoes changes even after it has been delivered. And to validate that a change has not affected some old functionality of the system, regression testing is done. In regression testing, old test cases are executed with the expectation that the same old results will be produced. Need for regression testing places additional requirements on the testing phase; it must provide the "old" test cases and their outputs.

In addition, as we have seen in the discussions in this chapter, testing has its own limitations. These limitations require that additional care be taken while performing testing. As testing is the costliest activity in software development, it is important that it be done efficiently.

All these factors mean that testing should not be done on-the-fly, as is sometimes done. It has to be carefully planned and the plan has to be properly executed. The testing process focuses on how testing should proceed for a particular project. Having discussed various methods of selecting test cases, we turn our attention to the testing process.

10.4.1 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, and system 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 10.

The first level of testing is called *unit testing*. In this, different modules are tested against the specifications produced during design for the modules. Unit testing is essentially for verification of the code produced during the coding phase, and hence the goal is to test the internal logic of the modules. It is typically done by the programmer

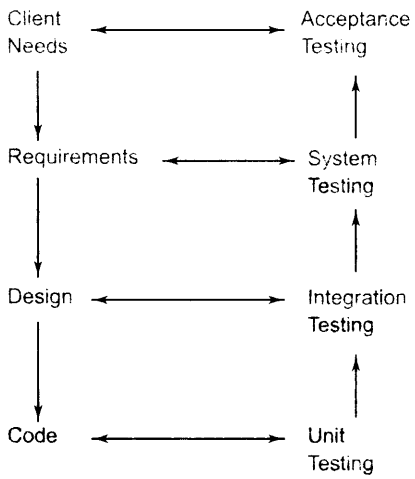

Figure 10.8: Levels of testing.

of the module. A module is considered for integration and use by others only after it has been unit tested satisfactorily. We have discussed it in more detail the previous chapter.

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 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 essentially a validation exercise, and in many situations it is the only validation activity. Acceptance testing is sometimes performed with realistic data of the client to demonstrate that the software is working satisfactorily. Testing here focuses on the external behavior of the system; the internal logic of the program is not emphasized. Consequently, mostly functional testing is performed at these levels.

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. Frequently, a change is made to "upgrade" the software by adding new features and functionality. Clearly, the modified software needs to be tested to make sure that the new features to be added do indeed work. However, as modifications have been 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.

A consequence of this is that the test cases for systems should be properly documented for future use in regression testing. In fact, for many systems that are frequently changed, regression testing scripts are used, which automate performing regression testing after some changes. A regression testing script executes a suite of test cases. For each test case, it sets the system state for testing, executes the test case, determines the output or some aspect of system state after executing the test case, and checks the system state or output against expected values. These scripts are typically produced during system testing, as regression testing is generally done only for complete systems. When the system is modified, the scripts are executed again, giving the inputs specified in the scripts and comparing the outputs with the outputs given in the scripts. Given the scripts, through the use of tools, regression testing can be largely automated.

Even with testing scripts, regression testing of large systems can take a considerable amount of time, particularly because execution and checking of all the test cases cannot be automated. If a small change is made to the system, often executing the entire suite of test cases is not justified, and the system is tested only with a subset of test cases. This requires *prioritization of test cases.* For prioritization, generally more data about each test case is recorded, which is then used during a regression testing to prioritize. For example, one approach is to record the set of blocks that each test case executes. If some part of the code has changed, then the test cases that execute the changed portion get the highest priority for regression testing. Test case prioritization is an active research area and many different approaches have been proposed in literature for this. We will not discuss it any further.

10.4.2 Test Plan

In general, testing commences with a *test plan* and terminates with 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 the entire testing process 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) system 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

One of the most important activities of the test plan is to identify the test units. A *test unit* is a set of one or more modules, together with associated data, that are from a single computer program and that are the object of testing. A test unit can occur at any level and can contain from a single module to the entire system. Thus, a test unit may be a module, a few modules, or a complete system.

As seen earlier, different levels of testing have to be used during the testing activity. The levels are specified in the test plan by identifying the test units for the project. Different units are usually specified for unit, integration, and system testing. 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. In general, a *test case specification* report, *test summary report*, and a *list of defects* should always be specified as deliverables. Test case specification is discussed later. The test summary report summarizes the results of the testing activities and evaluates the results. It defines the items tested, the environment in which testing was done, and a summary of defects found during testing.

The test plan, if it is a document separate from the project management plan, typically also specifies the schedule and effort to be spent on different activities of testing. This schedule should be consistent with the overall project schedule. For detailed planning and execution, the different tasks in the test plan should be enumerated and allocated 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.

10.4.3 Test Case Specifications

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 specification has to be done separately for *each unit*. Based on the approach specified in the test plan, first the features to be tested for this unit must be determined. The overall approach stated in the plan is refined into specific test techniques that should be followed and into the criteria to be used for evaluation. Based on these, the test cases are 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. Test case specifications look like a table of the form shown in Figure 10.

Sometimes, a few columns are also provided for recording the outcome of different rounds of testing. That is, sometimes 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 the entries will pass.

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

Figure 10.9: Test case specifications.

Test case specification 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 have considered many methods of generating test cases and criteria for evaluating test cases. A combination of these can be used to select the test cases. It should be pointed out that test case specifications contain not only the test cases, but also the rationale of selecting each test case (such as what condition it is testing) and the expected output for the test case.

There are two basic reasons 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. Even for a given criterion, the exact nature of the test cases affects the effectiveness of testing. Constructing "good" test cases that will reveal errors in programs is still a very creative activity that depends a great deal on the ingenuity of the tester. Clearly, it is important to ensure that the set of test cases used is of "high quality."

As with many other verification methods, evaluation of quality of test cases is done through "test case review." For any review, a formal document or work product is needed. This is the primary reason for having the test case specification in the form of a document. 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 in general cover the various aspects of the unit to be tested. For this purpose, the reason for selecting the test case and the expected output are also given in the test case specification document. By looking at the conditions being tested by the test cases, the reviewers can check if all the important conditions are being tested. As conditions can also be based on the output, by considering the expected outputs of the test cases, it can also be determined if the production of all the different types of outputs the unit is supposed to produce are being tested. Another reason for specifying the expected outputs is to use it as the "oracle" when the test case is executed.

Besides reviewing, another reason for specifying the test cases in a document is that the process of sitting down and specifying all the test cases that will be used for testing

helps the tester in selecting a good set of test cases. 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.

Another reason for formal test case specifications is that the specifications can be used as "scripts" during regression testing, particularly if regression testing is to be performed manually. Generally, the test case specification document itself is used to record the results of testing. That is, a column is created when test cases are specified that is left blank. When the test cases are executed, the results of the test cases are recorded in this column. Hence, the specification document eventually also becomes a record of the testing results.

10.4.4 Test Case Execution and Analysis

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. Sometimes, the steps to be performed to execute the test cases are specified in a separate document called the *test procedure specification*. This document specifies any special requirements that exist for setting the test environment and describes the methods and formats for reporting the results of testing. Measurements, if needed, are also specified, along with methods to obtain them.

Various outputs are produced as a result of test case execution for the unit under test. These outputs are needed to evaluate if the testing has been satisfactory. The most common outputs are the *test summary report*, and the *error report*. The test summary report is meant for project management, where the summary of the entire test case execution is provided. The summary gives the total number of test cases executed, the number and nature of errors found, and a summary of the metrics data collected. The error report is the details of the errors found during testing.

Testing requires careful monitoring, as it consumes the maximum effort, and has a great impact on final quality. A few metrics are very useful for monitoring testing. *Testing effort* is the total effort actually spent by the team in testing activities, and is an indicator of whether or not sufficient testing is being performed. If inadequate testing is done, it will be reflected in a reduced testing effort or reduced testing schedule. From the plan and past experience we should know the expected effort and duration of testing. The estimated effort is used for monitoring. Such monitoring can catch the "miracle finish" cases, where the project "finishes" suddenly, soon after the coding is done. Such "finishes" occur for reasons such as unreasonable schedules, personnel shortages, and slippage of schedule. Such a finish usually implies that to finish the project the testing

phase has been compressed too much, which is likely to mean that the software has not been evaluated properly.

Computer time consumed during testing is another measure that can give valuable information to project management. In general, in a software development project, the computer time consumption is low at the start, increases as time progresses, and reaches a peak. Thereafter it is reduced as the project reaches its completion. Maximum computer time is consumed during the latter part of coding and testing. By monitoring the computer time consumed, one can get an idea about how thorough the testing has been. Again, by comparing the previous buildups in computer time consumption, computer time consumption of the current project can provide valuable information about whether or not the testing is adequate.

The error report gives the list of all the defects found. The defects are generally also categorized into different categories. To facilitate reporting and tracking of defects found during testing (and other quality control activities), defects found must be properly recorded. This recording is generally done using tools. Let us now look at the defect logging and tracking activity, and how some simple analysis can be done on the defect data to aid project monitoring. With defect logging using tools, the error report is really a view of the logged defect data.

10.4.5 Defect Logging and Tracking

A large software project may include thousands of defects that are found by different people at different stages of the project. Often the person who fixes a defect is different than the person who finds or reports the defect. In such a scenario, defect reporting and closing cannot be done informally. The use of informal mechanisms may 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 [26], 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 10. A defect that is not closed is also called open.

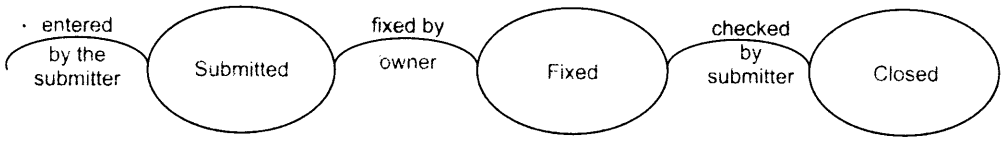

Figure 10.10: Life cycle of a defect.

This is a typical life cycle of a defect which is used in many organizations (e.g. [97]). However, the life cycle can be expanded or contracted to suit the purposes of the project or the organization. For example, some organizations developing critical systems may have more stages in the life cycle to track the defect more closely. Similarly, in a small non-critical project, the life cycle may have only two states—open and closed.

When logging a defect, sufficient information has to be recorded so that the effects can be recreated and debugging and fixing can be done. However, just tracking each defect is not sufficient for most projects, as analysis of defect data can also be very useful for improving the quality. To permit such analysis, suitable information has to be recorded. What data is recorded depends on the organization, and an example from an organization can be found in [97].

To understand the nature of defects being found, frequently defects are categorized into a few types, and the type of each defect is recorded. Such a classification is essential if causes of defects are to be identified later and then removed in an attempt to prevent defects from occurring. The defects can be classified in many different ways, and many schemes have been proposed. The orthogonal defect classification scheme [33], for example, classifies defects in categories that include functional, interface, assignment, timing, documentation, and algorithm. Some of the defect types used in a commercial organization are: Logic, Standards, User Interface, Component Interface, Performance, and Documentation [97].

The severity of the defect with respect to its impact on the working of the system is also often divided into few categories. This information is important for project management. For example, if a defect impacts a lot of users or has a catastrophic effect, then a project leader will want to fix it urgently. Similarly, if a defect is of a minor nature, it may be scheduled at ease. Hence classification of defects with respect to severity is very important for managing a project. Recording severity of defects found is also a standard practice in most software organizations. Most often a four-level classification is used. One such classification is:

- *Critical.* Show stopper: affects a lot of users: can delay project.

- *Major.* Has a large impact but workaround exists: considerable amount of work needed to fix it, though schedule impact is less.
- *Minor.* An isolated defect that manifests rarely and with little impact.
- *Cosmetic.* Small mistakes that don't impact the correct working.

At the end of the project, ideally no open defects should remain. However, this ideal situation is often not practical for most large systems. Using severity classification, a project may have release criteria like "software can be released only if there are no critical and major bugs, and minor bugs are less than x per feature."

The defect data can be analyzed in other ways to improve project monitoring and control. A standard analysis done on almost all long lasting projects is to plot and observe the defect arrival and closure trend. Plotting both the arrival and removal can at a glance provide a view of the state of the quality control tasks in the project. An example of such a curve is shown in Figure 10 [97]. According to this curve, the gap between the total defects and the total closed defects is gradually increasing, although the increase is not too alarming. (In the project, this visibility prompted a change in the project schedule—development activity was slowed and resources were assigned to defect fixing such that the number of open defects was brought down.)

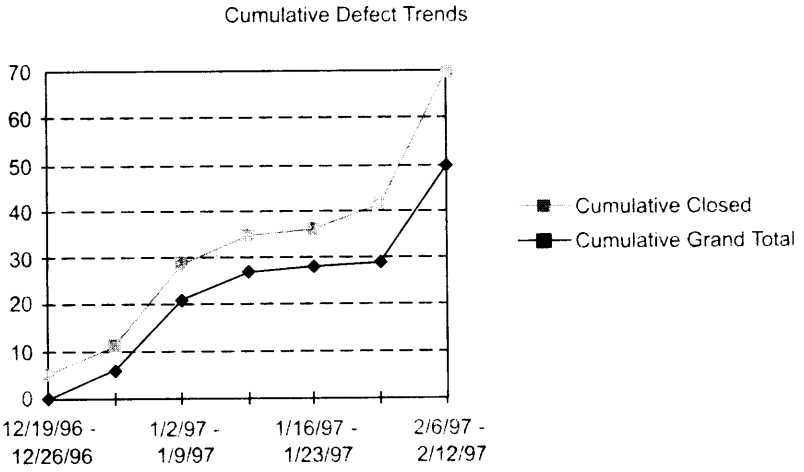

Figure 10.11: Defect arrival and closure trend.

In addition to plotting the arrival and fixing, the volume of open defects can also be plotted. This gives a direct plot of how many defects are still not closed. This plot, generally increases with time first, and then starts decreasing. Towards project completion this plot should reach towards zero. For some intervals, the number of open

defects might touch zero. That is, at some point during the project, all defects have been closed. Of course, this does not mean that there are no defects in the software—after reaching the zero open defect, further testing (and adding of code) may reveal defects. In other words, this plot is not monotonically decreasing, though it is expected that for most controlled projects its general trend will be downwards.

The defect data can also be analyzed for improving the process. One specific technique for doing this is defect prevention. We will discuss this further in the following section.

10.5 Defect Analysis and Prevention

We have seen that defects are introduced during development and are removed by the various quality control tasks in the process. Whereas the focus of the quality control tasks it to identify and remove the defects, the aim of defect prevention is to learn from defects found so far on the project and prevent defects from getting injected in the rest of the project. Some forms of defect prevention are naturally practiced and in a sense the goal of all standards, methodologies, and rules. is basically to prevent defects. However, when actual defect data is available, more effective defect prevention is possible through defect data analysis [76, 75]. Here we discuss an approach for doing focused defect prevention, based on practices of a commercial organization [97].

Defects analysis and prevention can be done at the organization level as well as at the project level. At the organization level, analysis of defects can lead to enhancements of organization-wide checklists, processes, or training. Defects analysis at the project level, aims to learn from defects found so far on the project and prevent defects in the rest of the project. Here we discuss only project-level analysis.

The main reason behind any defect prevention activity is to improve quality and improve productivity. Quality improves as with fewer defects injected, with the same effectiveness of quality control processes, the final system will have fewer defects. Productivity improves as lesser effort is spent on removing defects.

For a project, defect analysis for prevention can be done after some amount of coding has been done and a representative set of defects is known. If an iterative process is used, then the natural place for doing defect analysis will be after an iteration. The main tasks to be performed for doing defect prevention are: Do Pareto analysis to identify the main defect types, perform causal analysis to identify the causes of defects, and identify solutions to attack the causes.

10.5.1 Pareto Analysis

Pareto analysis is a common statistical technique used for analyzing causes, and is one of the primary tools for quality management [119, 139]. It is also sometimes called the

80-20 rule: 80% of the problems come from 20% of the possible sources. In software it can mean that 80% of the defects are caused by 20% of the root causes or that 80% of the defects are found in 20% of the code.

The first step for defect prevention is to draw a Pareto chart from the defect data. The number of defects found of different types is determined from the defect data and is plotted as a bar chart in the decreasing order. Along with the bar chart, a chart is also plotted on the same graph showing the cumulative number of defects as we move from types of defects given on the left of the x-axis to the right of the x-axis. The Pareto chart makes it immediately clear in visual as well as quantitative terms which are the main types of defects, and also which types of defects together form 80-85% of the total defects. If defects are being logged with information about their type, it is relatively easy to draw the Pareto chart.

As an example, consider the Pareto chart of the defect data for a project shown in Figure 10 [97]. This is a project in which features are being added to an existing system. The defects data for all enhancements done so far was used for this analysis. As can be seen, the logic defects are the most, followed by user interface defects, followed by standards defects. Defects in these three categories together account for more than 88% of the total defects, while the defects in the top two categories account for over 75% of the defects. Clearly, the target for defect prevention should be the top two or the top three categories such that defects in these categories can be reduced.

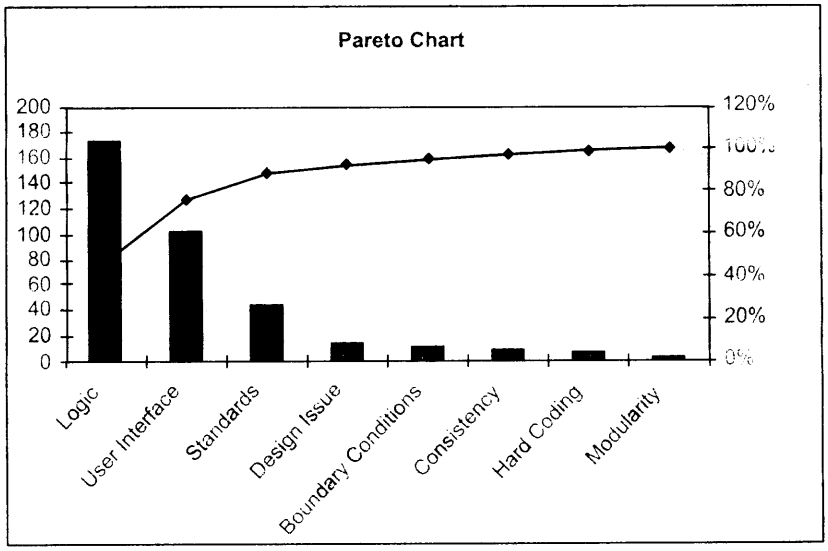

Figure 10.12: Pareto chart for defects found in ACE project.