Software Testing

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.

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.

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. 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. 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.

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.

Testing Process

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

Furthermore, we know that software typically undergoes changes evenafter it has been delivered. And to validate that a change has not affectedsome old functionality of the system, regression testing is done. In regressiontesting, old test cases are executed with the expectation that the sameold results will be produced. Need for regression testing places additionalrequirements on the testing phase; it must provide the "old" test cases andtheir outputs.

In addition, testing hasits own limitations. These limitations require that additional care be takenwhile 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 issometimes done. It has to be carefully planned and the plan has to be properlyexecuted. The testing process focuses on how testing should proceedfor a particular project.

Levels of Testing

Testing is usually relied upon to detect the faults remaining from earlierstages, 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 testingaims to test different aspects of the system.

The basic levels are unit testing, integration testing, and system and acceptancetesting. These different levels of testing attempt to detect differenttypes of faults. The relation of the faults introduced in different phases, andthe different levels of testing are shown in Figure below.

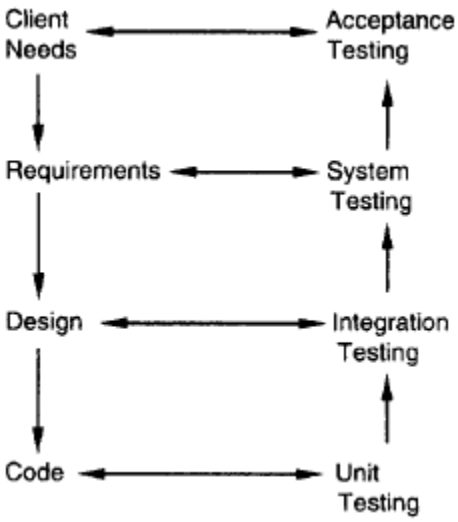

Fig. Levels of Testing

The first level of testing is called **Unit Testing**. In this, different modulesare tested against the specifications produced during design for the modules.Unit testing is essentially for verification of the code produced during thecoding phase, and hence the goal is to test the internal logic of the modules.It is typically done by the programmer of the module. A module is consideredfor integration and use by others only after it has been unit tested satisfactorily.

Unit testing is like regular testing where programs are executed with some test cases except that the focus is on testing smaller programs or modules called units. A unit may be a function, a small collection or functions, a class, or a small collection of classes. Most often, it is the unit a programmer is writing code for, and hence unit testing is most often done by a programmer to test the code that he or she has written.

Testing of modules or software systems is a difficult and challenging task. Selection of test cases is a key issue in any form of testing. During unit testing the tester, who isgenerally the programmer, will execute the unit for a variety of test casesand study the actual behavior of the units being tested for these test cases. Based on the behavior, the tester decides whether the unit is working correctlyor not. If the behavior is not as expected for some test case, then theprogrammer finds the defect in the program (an activity called debugging),and fixes it. After removing the defect, the programmer will generally executethe test case that caused the unit to fail again to ensure that the fixinghas indeed made the unit behave correctly.

For a functional unit, unit testing will involve testing the function withdifferent test data as input. In this, the tester will select different types of test data to exercise the function. Typically, the test data will include somedata representing the normal case, that is, the one that is most likely tooccur. In addition, test data will be selected for special cases which must bedealt with by the program and which might result in special or exceptionalresult.

An issue with unit testing is that as the unit being tested is not a completesystem but just a part, it is not executable by itself. Furthermore, in itsexecution it may use other modules that have not been developed yet. Dueto this, unit testing often requires drivers or stubs to be written. Driversplay the role of the "calling" module and are often responsible for gettingthe test data, executing the unit

with the test data, and then reporting theresult. Stubs are essentially "dummy" modules that are used in place of theactual module to facilitate unit testing. So, if a module M uses services fromanother module M' that has not yet been developed, then for unit testingM, some stub for M' will have to be written so M can invoke the servicesin some manner on M' so that unit testing can proceed. The need for stubscan be avoided, if coding and testing proceeds in a bottom-up manner—themodules at lower levels are coded and tested first such that when modulesat higher levels of hierarchy are tested, the code for lower level modules isalready available.

If incremental coding is practiced, as discussed above, then unit testingneeds to be performed every time the programmer adds some code. Clearly,for this, automated scripts for unit testing are essential. With automatedscripts, whether the programs pass the unit tests or not can be determinedsimply by executing a script. For incremental testing it is desirable that theprogrammer develops this unit testing script and keeps enhancing it withadditional test cases as the code evolves. That is, instead of executing theunit by executing it and manually inputting the test data, it is better ifexecution of the unit with the chosen test data is all programmed. Thenthis program can be executed every time testing needs to be done. Sometools are available to facilitate this.

In object-oriented programs, the unit to be tested is usually an objectof a class. Testing of objects can be defined as the process of exercisingthe routines provided by an object with the goal of uncovering errors in theimplementation of the routines or state of the object or both. For anobject, we can test a method using approaches for testing functions, but wecannot test the object using these approaches, as the issue of state comes in.To test an object, we also have to test the interaction between the methodsprovided on the object.

State-based testing is a technique that can be used for unit testing anobject. In the simplest form, a method is tested in ah possible states that theobject can assume, and after each invocation the resulting state is checkedto see whether or not the method takes the object under test to the expectedstate. For state-based testing to be practical, the set of states in which amethod is tested has to be limited. State modeling of classes can help here, or the tester can determine the important states of the object. Oncethe different object states are decided, then a method is tested in all thosestates that form valid input for it.

To test a class, the programmer needs to create an object of that class,take the object to a particular state, invoke a method on it, and then checkwhether the state of the object is as expected. This sequence has to be executedmany times for a method, and has to be performed for all the methods.

The next level of testing is often called **Integration Testing**. In this, manyunit 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 activitycan be considered testing the design.

The next levels are **System Testing** and **Acceptance Testing**. Here the entiresoftware system is tested. The reference document for this process isthe requirements document, and the goal is to see if the software meets itsrequirements. This is essentially a validation exercise, and in many situationsit is the only validation activity. Acceptance testing is sometimes performedwith realistic data of the client to demonstrate that the software is workingsatisfactorily. Testing here focuses on

the external behavior of the system; the internal logic of the program is not emphasized. Consequently, mostlyfunctional testing is performed at these levels.

These levels of testing are performed when a system is being built fromthe components that have been coded. There is another level of testing,called **Regression Testing** that is performed when some changes are made toan existing system. We know that changes are fundamental to software; anysoftware must undergo changes. Frequently, a change is made to "upgrade"the software by adding new features and functionality. Clearly, the modifiedsoftware needs to be tested to make sure that the new features to be addeddo indeed work. However, as modifications have been made to an existingsystem, testing also has to be done to make sure that the modification hasnot had any undesired side effect of making some of the earlier services faulty.

That is, besides ensuring the desired behavior of the new services, testinghas 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 oldsystem are maintained, along with the output produced by the old system.These test cases are executed again on the modified system and its outputcompared with the earlier output to make sure that the system is working asbefore on these test cases. This frequently is a major task when modificationsare to be made to existing systems.

A consequence of this is that the test cases for systems should be properlydocumented 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 testingscript executes a suite of test cases. For each test case, it sets the systemstate for testing, executes the test case, determines the output or some aspect of system state after executing the test case, and checks the systemstate 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 oftools, regression testing can be largely automated.

Even with testing scripts, regression testing of large systems can take aconsiderable amount of time, particularly because execution and checkingof all the test cases cannot be automated. If a small change is made tothe 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.

Alpha Beta and Gamma Testing

Usage of the "alpha/beta" test terminology originated at IBM. The software release life cycle is the different stages that describe the stability of the software product/project. Each major version of the

product usually goes through a stage which known as a software release life cycle. It has different phases through which a project goes to the production and every phase have their own objective.

Pre-alpha: Pre alpha refers to all the activities performed during the software product prior to testing. These activities can include requirements analysis, software design, software developments and unit testing.

A QA can do following things in Pre-alpha stage:

- Requirement Analysis.
- Requirement Testing.
- Test Plan creation.
- Create the test scenarios and test cases.
- Test bed creation.
- Create the test execution plan.

Alpha Testing Phase: This is the first phase of actual software testing (alpha is the first letter of the Greek alpha beta), in this phase, we use the white box/Black box/Gray box software testing techniques to test the software product. This is the in-house testing of the product in presence of developers in laboratory setting. Alpha testing is done by developer himself, or separate testing team, or by client. Generally we perform all testing types in alpha testing phase. Alpha testing phase ends with a feature freeze, indicating that no more features will be added to the software.

Types of testing done by tester in Alpha phase:

- Smoke testing.
- Integration Testing.
- System testing.
- UI and Usability testing.
- Functional Testing.
- Security Testing.
- Performance Testing.
- Regression testing.
- Sanity Testing.
- Acceptance Testing.

Purpose of Alpha testing: The purpose of the alpha testing is to validate the product in all perspective, which can be functional label, UI & usability label, security or performance label.

Suppose if we are going to release the build for 10 features, and 3 of them have certain Block and Major issues so either we should resolve them or to release the product with 7 feature in Beta. Before going to alpha generally in-house testers insure that testing of all application area has been done, no Block/crash/Major issues remain.

Beta Testing Phase: In software development, a beta test is the second phase of software testing in which a sampling of the intended audience tries the product out. (Beta is the second letter of the Greek alphabet.)

Originally, the term alpha test meant the first phase of testing in a software development process. The first phase includes unit testing, component testing, and system testing.

Beta testing can be considered "pre-release testing". Beta test versions of software are now distributed to a wide audience (Selected group of real users, outside the development environment) on the Web partly to give the program a "real-world" test and partly to provide a preview of the next release.

Purpose of Beta testing: The main objective behind the Beta testing is to get the feedback from the different groups of customers and check the compatibility of the product in different kind of networks, hardware's, impact of the different installed software on product, check the usability of the product. This is typically the first time that the software is available outside of the organization that developed it. The users of a beta version are called beta testers.

Type of Beta: Developers release either a closed beta or an open beta; closed beta versions are released to a select group of individuals for a user test and are invitation only, while open betas are from a larger group to the general public and anyone interested. The testers report any bugs that they find, and sometimes suggest additional features they think should be available in the final version.

Open betas serve the dual purpose of demonstrating a product to potential consumers, and testing among an extremely wide user base likely to bring to light obscure errors that a much smaller testing team may not find.

Gamma Testing Phase: This is the third phase of software testing. Gamma testing is done once the software is ready for release with specified requirements. This testing is done directly by skipping all the in-house test activities (no need to do all in-house quality check).

The software is almost ready for final release. No feature development or enhancement of the software is undertaken; tightly scoped bug fixes are the only code you're allowed to write in this phase, and even then only for the most heinous and debilitating of bugs.

Gamma Check: Gamma check is performed when the application is ready for release to the specified requirements; this check is performed directly without going through all the testing activities at home.

Purpose of Beta testing: Objective of the gamma testing is to validate the all functional are of the product is working fine or not and is product ready for release. Here QA do the testing like acceptance testing.

Test Case Methods

Black Box Testing

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 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. Some of these techniques are:

Equivalence Class Partitioning: As 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). 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 behaviour 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, $0 < count < Max$), 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., $count < 0$) and the other with values higher than the higher bound ($count > Max$). 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 A'' 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 below:

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: 5 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, EQ2, 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 lEQ1 through IEQ3.

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, 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 < x < 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 below.

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.

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 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 ~. 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:

creditacct_numbertransaction_amount
debitacct_numbertransaction_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 below (see the graph). 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.

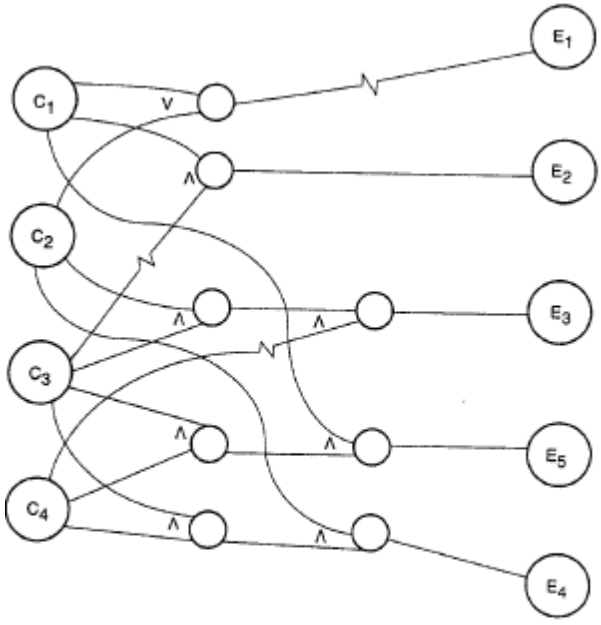

Figure 10.3: The cause-effect graph.

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

Decision table for the cause-effect graph.

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 above.

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.

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. 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 within 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 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 — l)/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 a1, a2, a3, b1, b2, b3, and c1, c2, c3.
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 10.2, 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. 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.

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.

Theoretically, any software that saves state can be modelled 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. 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.

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.

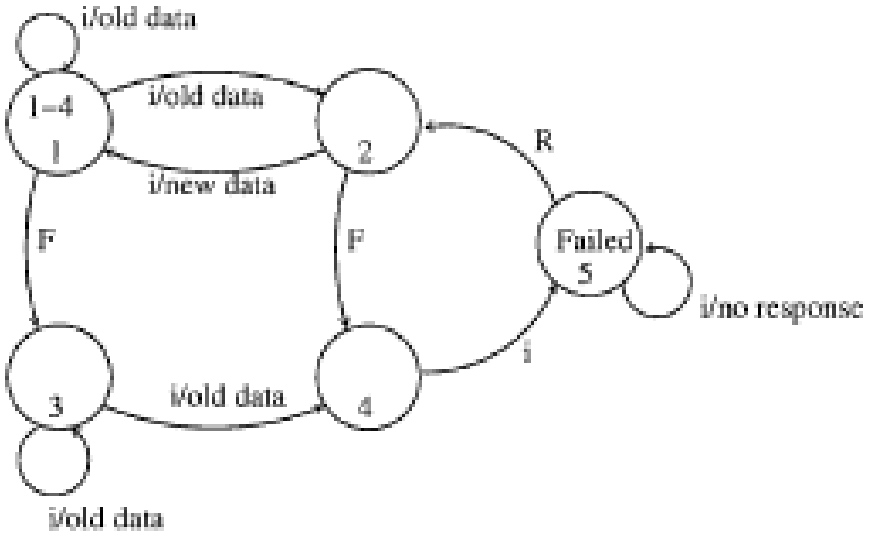

Figure 10.5: State model for the student survey system.

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.

[Extra about state based testing: 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 modelling 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.

One important question is: given a state model of a system how should test cases be generated? Many coverage criteria have been proposed. 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 10.3. Here req() means that a request for taking the survey should be given, fail() means that the database should be failed, and recover() means that the failed database should be recovered.
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.]

White Box/Glassbox Testing

White-box testing 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. Whitebox 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.

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. 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+i}) for all nodes U_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*.

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 case 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. Branch coverage implies statement coverage, as each statement is a part of some branch. In other words, C_{branch} => 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/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 >= 0) && (x <= 200))
check = True;
else check = False;
}
```

The module is incorrect, as it is checking for x < 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 >= 0). The condition (x <= 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} => 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. 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.

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 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.

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:

- *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.

Data flow testing is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects. Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used.

Data-flow testing looks at the life-cycle of a particular piece of data (i.e. a variable) in an application. By looking for patterns of data usage, risky areas of code can be found and more test cases can be applied.

There are four ways data can be used: defined, used in a predicate, used in a calculation, and killed. Certain patterns, using a piece of data in a calculation after it has been killed, show an anomaly in the code, and therefore the possibility of a bug. Data Flow Testing helps us to pinpoint any of the following issues:

- A variable that is declared but never used within the program.
- A variable that is used but never declared.
- A variable that is defined multiple times before it is used.
- Deallocating a variable before it is used

Mutation Testing

Mutation testing 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. 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.

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 '-').

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:

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 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.

Gray Box Testing

Gray-box testing is a combination of white-box testing and black-box testing. The aim of this testing is to search for the defects if any due to improper structure or improper usage of applications. A black-box tester is unaware of the internal structure of the application to be tested, while a white-box tester has access to the internal structure of the application. A gray-box tester partially knows the internal structure, which includes access to the documentation of internal data structures as well as the algorithms used. Gray-box testing is beneficial because it takes the straightforward technique of black-box testing and combines it with the code targeted systems in white-box testing.

Test Metrics

Once the project is finished, one can look at the overall productivity achieved by the programmers during the project. As discussed earlier, productivity can be measured as lines of code (or function points) per person-month. However, at the end of testing, as most defects have been uncovered, removal efficiencies can be estimated.

Reliability of software often depends considerably on the quality of testing. Hence, by assessing reliability we can also judge the quality of testing. Alternatively, reliability estimation can be used to decide whether enough testing has been done. Hence, besides characterizing an important quality property of the product being delivered, reliability estimation has a direct role in project management—the reliability models being used by the project manager to decide when to stop testing.

As software does not "wear out" or "age" as a mechanical or an electronic system does, the unreliability of software is primarily due to bugs or design faults in the software. It is widely believed that with the current level of technology it is impossible to detect and remove all the faults in a large software system (particularly before delivery). Consequently, a software system is expected to have some faults in it.

Reliability is a probabilistic measure that assumes that the occurrence of failure of software is a random phenomenon. Let X be the random variable that represents the life of a system. Reliability of a system is the probability that the system has not failed by time t. In other words,

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

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

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

Note that one can obtain the MTTF from the reliability function but the reverse is not always true. The reliability function can, however, be obtained from the MTTF if the failure process is assumed to be Poisson, that is, the life time has an exponential distribution. With exponential distribution, if the failure rate of the system is known as A, the MTTF is equal to $1/\lambda$

Reliability can also be defined in terms of the number of failures experienced by the system by time t. Clearly, this number will also be random as failures are random. With this random variable, we define the *failure intensity* A(t) of the system as the number of expected failures per unit time at time t.

Musa's reliability model focuses on failure intensity while modelling reliability. It assumes that the failure intensity decreases with time, that is, as (execution) time increases, the failure intensity decreases. This assumption is generally true as the following is assumed about the software testing activity, during which data is being collected: during testing, if a failure is observed, the fault that caused that failure is detected and the fault is removed. Consequently, the failure intensity decreases. Most other models make similar assumption which is consistent with actual observations.

Testing Documents

a) 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.

b) 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 10.9.

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.

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.

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

Figure 10.9: Test case specifications.

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.

More on Testing types (short description only)

Ad-hoc testing
This type of software testing is very informal and unstructured and can be performed by any stakeholder with no reference to any test case or test design documents.
The person performing Ad-hoc testing has a good understanding of the domain and workflows of the application to try to find defects and break the software. Ad-hoc testing is intended to find defects that were not found by existing test cases.

Acceptance Testing
Acceptance testing is a formal type of software testing that is performed by end user when the features have been delivered by developers. The aim of this testing is to check if the software confirms to their business needs and to the requirements provided earlier. Acceptance tests are

normally documented at the beginning of the sprint (in agile) and is a means for testers and developers to work towards a common understanding and shared business domain knowledge.

Accessibility Testing
In accessibility testing, the aim of the testing is to determine if the contents of the website can be easily accessed by disable people. Various checks such as color and contrast (for color blind people), font size for visually impaired, clear and concise text that is easy to read and understand.

Agile Testing
Agile Testing is a type of software testing that accommodates agile software development approach and practices. In an Agile development environment, testing is an integral part of software development and is done along with coding. Agile testing allows incremental and iterative coding and testing.

API Testing
API testing is a type of testing that is similar to unit testing. Each of the Software APIs are tested as per API specification. API testing is mostly done by testing team unless APIs to be tested or complex and needs extensive coding. API testing requires understanding both API functionality and possessing good coding skills.

Automated testing
This is a testing approach that makes use of testing tools and/or programming to run the test cases using software or custom developed test utilities. Most of the automated tools provided capture and playback facility, however there are tools that require writing extensive scripting or programming to automate test cases.

All Pairs testing
Also known as Pair wise testing, is a black box testing approach and a testing method where in for each input is tested in pairs of inputs, which helps to test software works as expected with all possible input combinations.

Beta Testing
This is a formal type of software testing that is carried out by end customers before releasing or handing over software to end users. Successful completion of Beta testing means customer acceptance of the software.

Black Box testing
Black box testing is a software testing method where in testers are not required to know coding or internal structure of the software. Black box testing method relies on testing software with various inputs and validating results against expected output.

Backward Compatibility Testing
Type of software testing performed to check newer version of the software can work successfully installed over previous version of the software and newer version of the software works as fine with table structure, data structures, files that were created by previous version of the software.

Boundary Value Testing (BVT)
Boundary Value Testing is a testing technique that is based on concept "error aggregates at boundaries". In this testing technique, testing is done extensively to check for defects at boundary conditions. If a field accepts value 1 to 100 then testing is done for values 0, 1, 2, 99, 100 and 101.

Big Bang Integration testing
This is one of the integration testing approaches, in Big Bang integration testing all or all most all of the modules are developed and then coupled together.

Bottom up Integration testing
Bottom up integration testing is an integration testing approach where in testing starts with smaller pieces or sub systems of the software till all the way up covering entire software system. Bottom up integration testing begins with smaller portion of the software and eventually scale up in terms of size, complexity and completeness.

Branch Testing
Is a white box testing method for designing test cases to test code for every branching condition. Branch testing method is applied during unit testing.

Browser compatibility Testing
It is one of the sub types of testing of compatibility testing performed by testing team. Browser compatibility testing is performed for web applications with combination of different browsers and operating systems.

Compatibility testing
Compatibility testing is one of the test types performed by testing team. Compatibility testing checks if the software can be run on different hardware, operating system, bandwidth, databases, web servers, application servers, hardware peripherals, emulators, different configuration, processor, different browsers and different versions of the browsers etc.,

Component Testing
This type of software testing is performed by developers. Component testing is carried out after completing unit testing. Component testing involves testing a group of units as code together as a whole rather than testing individual functions, methods.

Condition Coverage Testing
Condition coverage testing is a testing technique used during unit testing, where in developer tests for all the condition statements like if, if else, case etc., in the code being unit tested.

Dynamic Testing
Testing can be performed as Static Testing and Dynamic testing, Dynamic testing is a testing approach where-in testing can be done only by executing code or software are classified as Dynamic Testing. Unit testing, Functional testing, regression testing, performance testing etc.,

Decision Coverage Testing
Is a testing technique that is used in Unit testing, objective of decision coverage testing is to expertise and validate each and every decisions made in the code e.g. if, if else, case statements.

End-to-end Testing
End to end testing is performed by testing team, focus of end to end testing is to test end to end flows e.g. right from order creation till reporting or order creation till item return etc and checking. End to end testing is usually focused mimicking real life scenarios and usage. End to end testing involves testing information flow across applications.

Exploratory Testing
Exploratory testing is an informal type of testing conducted to learn the software at the same time looking for errors or application behavior that seems non-obvious. Exploratory testing is usually done by testers but can be done by other stake holders as well like Business Analysts, developers, end users etc. who are interested in learning functions of the software and at the same time looking for errors or behavior is seems non-obvious.

Equivalence Partitioning
Equivalence partitioning is also known as Equivalence Class Partitioning is a software testing technique and not a type of testing by itself. Equivalence partitioning technique is used in black box and grey box testing types. Equivalence partitioning classifies test data into Equivalence classes as positive Equivalence classes and negative Equivalence classes, such classification ensures both positive and negative conditions are tested.

Functional Testing
Functional testing is a formal type of testing performed by testers. Functional testing focuses on testing software against design document, Use cases and requirements document. Functional testing is a black box type of testing and does not require internal working of the software unlike white box testing.

Fuzz Testing
Fuzz testing or fuzzing is a software testing technique that involves testing with unexpected or random inputs. Software is monitored for failures or error messages that are presented due to the input errors.

GUI (Graphical User Interface) testing
This type of software testing is aimed at testing the software GUI (Graphical User Interface) of the software meets the requirements as mentioned in the GUI mockups and Detailed designed documents. For e.g. checking the length and capacity of the input fields provided on the form, type of input field provided, e.g. some of the form fields can be displayed as dropdown box or a set of

radio buttons. So GUI testing ensures GUI elements of the software are as per approved GUI mockups, detailed design documents and functional requirements. Most of the functional test automation tools work on GUI capture and playback capabilities. This makes script recording faster at the same time increases the effort on script maintenance.

Glass box Testing
Glass box testing is another name for White box testing. Glass box testing is a testing method that involves testing individual statements, functions etc., Unit testing is one of the Glass box testing methods.

Gorilla Testing
This type of software testing is done by software testing team, has a scary name though. Objective of Gorilla Testing is to exercise one or few functionality thoroughly or exhaustively by having multiple people test the same functionality.

Happy path testing
Also known as Golden path testing, this type of testing focuses on selective execution of tests that do not exercise the software for negative or error conditions.

Integration Testing
Integration testing also known as I&T in short, in one of the important types of software testing. Once the individual units or components are tested by developers as working then testing team will run tests that will test the connectivity among these units/component or multiple units/components. There are different approaches for Integration testing namely, Top-down integration testing, Bottom-up integration testing and a combination of these two known as Sand witch testing.

Interface Testing
Software provides support for one or more interfaces like “Graphical user interface”, “Command Line Interface” or “Application programming interface” to interact with its users or other software. Interfaces serves as medium for software to accept input from user and provide result. Approach for interface testing depends on the type of the interface being testing like GUI or API or CLI.

Internationalization Testing
Internationalization testing is a type of testing that is performed by software testing team to check the extent to which software can support Internationalization i.e., usage of different languages, different character sets, double byte characters etc., For e.g.: Gmail, is a web application that is used by people all over work with different languages, single by or multi byte character sets.

Keyword-driven Testing
Keyword driver testing is more of a automated software testing approach than a type of testing itself. Keyword driven testing is known as action driven testing or table driven testing.

Load Testing
Load testing is a type of non-functional testing; load testing is done to check the behavior of the software under normal and over peak load conditions. Load testing is usually performed using automated testing tools. Load testing intends to find bottlenecks or issues that prevent software from performing as intended at its peak workloads.

Localization Testing
Localization testing a type of software testing performed by software testers, in this type of testing, software is expected to adapt to a particular locale, it should support a particular locale/language in terms of display, accepting input in that particular locale, display, font, date time, currency etc., related to a particular locale. For e.g. many web applications allow choice of locale like English, French, German or Japanese. So once locale is defined or set in the configuration of software, software is expected to work as expected with a set language/locale.

Negative Testing
This type of software testing approach, which calls out the "attitude to break", these are functional and non-functional tests that are intended to break the software by entering incorrect data like incorrect date, time or string or upload binary file when text files supposed to be upload or enter huge text string for input fields etc. It is also a positive test for an error condition.

Non-functional testing
Software are built to fulfil functional and non-functional requirements, non-functional requirements like performance, usability, localization etc., There are many types of testing like compatibility testing, compliance testing, localization testing, usability testing, volume testing etc., that are carried out for checking non-functional requirements.

Pair Testing
is a software testing technique that can be done by software testers, developers or Business analysts (BA). As the name suggests, two people are paired together, one to test and other to monitor and record test results. Pair testing can also be performed in combination of tester-developer, tester-business analyst or developer-business analyst combination. Combining testers and developers in pair testing helps to detect defects faster, identify root cause, fix and test the fix.

Performance Testing
is a type of software testing and part of performance engineering that is performed to check some of the quality attributes of software like Stability, reliability, availability. Performance testing is carried out by performance engineering team. Unlike Functional testing, Performance testing is done to check non-functional requirements. Performance testing checks how well software works in anticipated and peak workloads. There are different variations or sub types of performance like load testing, stress testing, volume testing, soak testing and configuration testing.

Penetration Testing
is a type of security testing, also known as pentest in short. Penetration testing is done to tests how secure software and its environments (Hardware, Operating system and network) are when subject to attack by an external or internal intruder. Intruder can be a human/hacker or malicious programs. Pentest uses methods to forcibly intrude (by brute force attack) or by using a weakness

(vulnerability) to gain access to a software or data or hardware with an intent to expose ways to steal, manipulate or corrupt data, software files or configuration. Penetration Testing is a way of ethical hacking, an experienced Penetration tester will use the same methods and tools that a hacker would use but the intention of Penetration tester is to identify vulnerability and get them fixed before a real hacker or malicious program exploits it.

Regression Testing
is a type of software testing that is carried out by software testers as functional regression tests and developers as Unit regression tests. Objective of regression tests are to find defects that got introduced to defect fix(es) or introduction of new feature(s). Regression tests are ideal candidate for automation.

Retesting
is a type of retesting that is carried out by software testers as a part of defect fix verification. For e.g. a tester is verifying a defect fix and let us say that there are 3 test cases failed due to this defect. Once tester verifies defect fix as resolved, tester will retest or test the same functionality again by executing the test cases that were failed earlier.

Risk based Testing
is a type of software testing and an different approach towards testing a software. In Risk based testing, requirements and functionality of software to be tested are prioritized as Critical, High, Medium and low. In this approach, all critical and High priority tests are tested and them followed by Medium. Low priority or low risk functionality are tested at the end or may not based on the time available for testing.

Smoke testing
is a type of testing that is carried out by software testers to check if the new build provided by development team is stable enough i.e., major functionality is working as expected in order to carry out further or detailed testing. Smoke testing is intended to find “show stopper” defects that can prevent testers from testing the application in detail. Smoke testing carried out for a build is also known as build verification test.

Security Testing
is a type of software testing carried out by specialized team of software testers. Objective of security testing is to secure the software is to external or internal threats from humans and malicious programs. Security testing basically checks, how good is software’s authorization mechanism, how strong is authentication, how software maintains confidentiality of the data, how does the software maintain integrity of the data, what is the availability of the software in an event of an attack on the software by hackers and malicious programs is for Security testing requires good knowledge of application, technology, networking, security testing tools. With increasing number of web applications necessarily of security testing has increased to a greater extent.

Sanity Testing
is a type of testing that is carried out mostly by testers and in some projects by developers as well. Sanity testing is a quick evaluation of the software, environment, network, external systems are up

& running, software environment as a whole is stable enough to proceed with extensive testing. Sanity tests are narrow and most of the time sanity tests are not documented.

Scalability Testing
is a non functional test intended to test one of the software quality attributes i.e. “Scalability”. Scalability test is not focused on just one or few functionality of the software instead performance of software as a whole. Scalability testing is usually done by performance engineering team. Objective of scalability testing is to test the ability of the software to scale up with increased users, increased transactions, increase in database size etc., It is not necessary that software’s performance increases with increase in hardware configuration, scalability tests helps to find out how much more workload the software can support with expanding user base, transactions, data storage etc.

Stability Testing
is a non functional test intended to test one of the software quality attributes i.e. “Stability”. Stability testing focuses on testing how stable software is when it is subject to loads at acceptable levels, peak loads, loads generated in spikes, with more volumes of data to be processed. Scalability testing will involve performing different types of performance tests like load testing, stress testing, spike testing, soak testing, spike testing etc.,

Static Testing
is a form of testing where in approaches like reviews, walkthroughs are employed to evaluate the correctness of the deliverable. In static testing software code is not executed instead it is reviewed for syntax, commenting, naming convention, size of the functions and methods etc. Static testing usually has check lists against which deliverables are evaluated. Static testing can be applied for requirements, designs, test cases by using approaches like reviews or walkthroughs.

Stress Testing
is a type of performance testing, in which software is subjected to peak loads and even to a break point to observe how the software would behave at breakpoint. Stress testing also tests the behavior of the software with insufficient resources like CPU, Memory, Network bandwidth, Disk space etc. Stress testing enables to check some of the quality attributes like robustness and reliability.

System Testing
this includes multiple software testing types that will enable to validate the software as a whole (software, hardware and network) against the requirements for which it was built. Different types of tests (GUI testing, Functional testing, Regression testing, Smoke testing, load testing, stress testing, security testing, stress testing, ad-hoc testing etc.,) are carried out to complete system testing.

Soak Testing
is a type of performance testing, where in software is subjected to load over a significant duration of time, soak testing may go on for few days or even for few weeks. Soak testing is a type of testing that is conducted to find errors that result in degeneration of software performance with continued usage. Soak testing is extensively done for electronic devices, which are expected to run continuously for days or months or years without restarting or rebooting. With growing web

applications soak testing has gained significant importance as web application availability is critical for sustaining and success of business.

System Integration Testing
known as SIT (in short) is a type of testing conducted by software testing team. As the name suggests, focus of System integration testing is to test for errors related to integration among different applications, services, third party vendor applications etc., As part of SIT, end-to-end scenarios are tested that would require software to interact (send or receive data) with other upstream or downstream applications, services, third party application calls etc.,

Unit testing
is a type of testing that is performed by software developers. Unit testing follows white box testing approach where developer will test units of source code like statements, branches, functions, methods OR class, interface in OOP (object oriented programming). Unit testing usually involves in developing stubs and drivers. Unit tests are ideal candidates for automation. Automated tests can run as Unit regression tests on new builds or new versions of the software. There are many useful unit testing frames works like Junit, Nunit etc., available that can make unit testing more effective.

Usability testing
is a type of software testing that is performed to understand how user friendly the software is. Objective of usability testing is to allow end users to use the software, observe their behavior, their emotional response (whether users liked using software or were they stressed using it? etc.,) and collect their feedback on how the software can be made more useable or user friendly and incorporate the changes that make the software easier to use.

User Acceptance testing (UAT)
User Acceptance testing is a must for any project; it is performed by clients/end users of the software. User Acceptance testing allows SMEs (Subject matter experts) from client to test the software with their actual business or real-world scenarios and to check if the software meets their business requirements.

Volume testing
is a non-functional type of testing carried out by performance engineering team. Volume testing is one of the types of performance testing. Volume testing is carried out to find the response of the software with different sizes of the data being received or to be processed by the software. For e.g. If you were to be testing Microsoft word, volume testing would be to see if word can open, save and work on files of different sizes (10 to 100 MB).

Vulnerability Testing
involves identifying, exposing the software, hardware or network Vulnerabilities that can be exploited by hackers and other malicious programs likes viruses or worms. Vulnerability Testing is key to software security and availability. With increase number of hackers and malicious programs, Vulnerability Testing is critical for success of a Business.

White box Testing

White box testing is also known as clear box testing, transparent box testing and glass box testing. White box testing is a software testing approach, which intends to test software with knowledge of internal working of the software. White box testing approach is used in Unit testing which is usually performed by software developers. White box testing intends to execute code and test statements, branches, path, decisions and data flow within the program being tested. White box testing and Black box testing complement each other as each of the testing approaches have potential to un-cover specific category of errors.
