SOFTWARE TESTING

UNIT-I

1. SOFTWARE TESTING TERMINOLOGY:

Testing: Testing is the process of executing a program with the intent of finding errors.
A successful test is one that uncovers an as-yet-undiscovered error.
Testing can show the presence of bugs but never their absence.

Failure: It means the inability of a system or component to perform a required function according to its specification.

Fault/Defect/Bug Failure is the term which is used to describe the problems in a system on the output side,
Fault: Is a condition that in actual causes a system to produce failure. Fault is synonymous with the words defect or bug.

Test case: Test case is a well-documented procedure designed to test the functionality of a feature in the system. A test case has an identity and is associated with program behavior. The primary purpose of designing a test case is to find errors in the system.
Test Case contains: Test Case ID, Purpose, Preconditions, Inputs and Expected Outputs

Testware: The documents created during testing activities are known as testware.

Test oracle: An oracle is the means to judge the success or failure of a test, i.e. to judge the correctness of the system for some test. The simplest oracle is comparing actual results with expected results by hand. This can be very time- consuming, so automated oracles are sought.

2. Evolution of Testing:

Based on the growth of software testing with time, we can divide the evolution of software testing into the following phases:

1. **Debugging-oriented Phase (Before 1957)**

This phase is the early period of testing. At that time, testing basics were unknown. Programs were written and then tested by the programmers until they were sure that all the bugs were removed. The term used for testing was checkout, focused on getting the system to run. Till 1956, there was no clear distinction between software development, testing, and debugging.

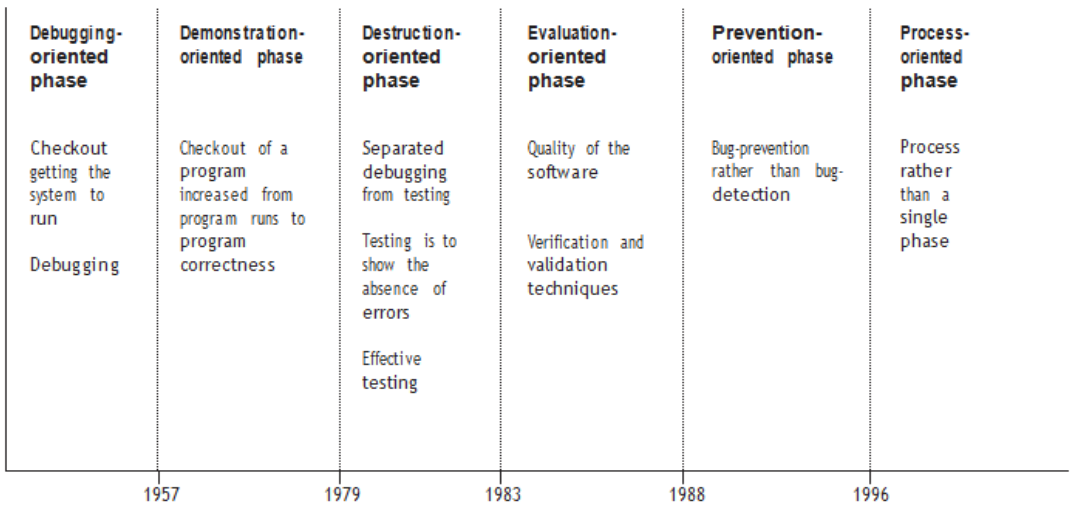
2. **Demonstration-oriented Phase (1957–78):** The term 'debugging' continued in this phase. Purpose of checkout is not only to run the software but also to demonstrate the correctness according to the mentioned requirements. Thus, the scope of checkout of a program increased from program runs to program correctness. Moreover, the purpose of checkout was to show the absence of errors. There was no stress on the test case design. In this phase, there was a misconception that the software could be tested exhaustively.

3. **Destruction-oriented Phase (1979–82):** This phase can be described as the revolutionary turning point in the history of software testing. View of testing from 'testing is to show the absence of errors' to 'testing is to find more and more errors.' Separated debugging from testing and stressed on the valuable test cases if they explore more bugs. This phase has given importance to effective testing in comparison to exhaustive testing. The importance of early testing was also realized in this phase.

4. **Evaluation-oriented Phase (1983–87):** With the concept of early testing, it was realized that if the bugs were identified at an early stage of development, it was cheaper to debug them as compared to the bugs found in implementation or post-implementation phases. This phase stresses on the quality of software products such that it can be evaluated at every stage of development. In fact, the early testing concept was established in the form of verification and validation activities which help in producing better quality software.

5. **Prevention Oriented Phase (1988-95):** The evaluation model stressed on the concept of bug-prevention as compared to the earlier concept of bug-detection. With the idea of early detection of bugs in earlier phases, we can prevent the bugs in implementation or further phases. Beyond this, bugs can also be prevented in other projects with the experience gained in similar software projects. The prevention model includes test planning, test analysis, and test design activities playing a major role, while the evaluation model mainly relies on analysis and reviewing techniques other than testing.

6. **Process-oriented Phase (1996 onwards):** In this phase, testing was established as a complete process rather than a single phase (performed after coding) in the software development life cycle (SDLC). The testing process starts as soon as the requirements for a project are specified and it runs parallel to SDLC. Moreover, the model for measuring the performance of a testing process has also been developed like CMM. The model for measuring the testing process is known as Testing Maturity Model (TMM). Thus, the emphasis in this phase is also on quantification of various parameters which decide the performance of a testing process.



Evolution phases of software testing

3. SOFTWARE TESTING—MYTHS AND FACTS

Myth1: Testing is a single phase in SDLC.
Truth: It is a myth, at least in the academia, that software testing is just a phase in SDLC and we perform testing only when the running code of the module is ready. But in reality, testing starts as soon as we get the requirement specifications for the software. And the testing work continues throughout the SDLC, even post-implementation of the software.

Myth2: Testing is easy.
Truth: This myth is more in the minds of students who have just passed out or are going to pass out of college and want to start a career in testing. So the general perception is that,

software testing is an easy job, wherein test cases are executed with testing tools only. But in reality, tools are there to automate the tasks and not to carry out all testing activities. Testers' job is not easy, as they have to plan and develop the test cases manually and it requires a thorough understanding of the project being developed with its overall design.

Myth3: Software development is worth more than testing.
Truth: This myth prevails in the minds of every team member and even in fresher's who are seeking jobs. As a fresher, we dream of a job as a developer. We get into the organization as a developer and feel superior to other team members. At the managerial level also, we feel happy about the achievements of the developers but not of the testers who work towards the quality of the product being developed.

Myth4: Complete testing is possible.
Truth: This myth also exists at various levels of the development team. Almost every person who has not experienced the process of designing and executing the test cases manually feels that complete testing is possible. Complete testing at the surface level assumes that if we are giving all the inputs to the software, then it must be tested for all of them. But in reality, it is not possible to provide all the possible inputs to test the software, as the input domain of even a small program is too large to test.

Myth5: Testing starts after program development.
Truth: Most of the team members, who are not aware of testing as a process, still feel that testing cannot commence before coding. But this is not true. As mentioned earlier, the work of a tester begins as soon as we get the and plans for the validation testing. He writes detailed test cases, executes the test cases, reports the test results, etc. Testing after coding is just a part of all the testing activities.

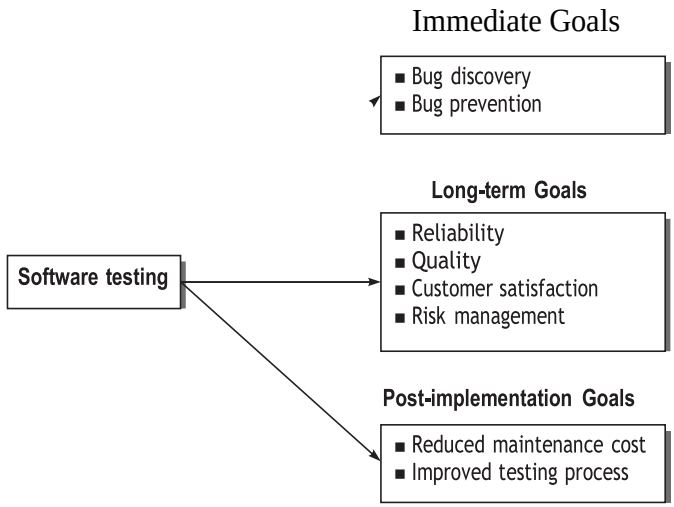
Myth6: The purpose of testing is to check the functionality of the software.
Truth: Today, all the testing activities are driven by quality goals. Ultimately, the goal of testing is also to ensure quality of the software. But quality does not imply checking only the functionalities of all the modules. There are various things related to quality of the software, for which test cases must be executed.

Myth7: Anyone can be a tester.
Truth: This is the extension of the myth that 'testing is easy.' Most of us think that testing is

an intuitive process and it can be performed easily without any training. And therefore, anyone can be a tester. As an established process, software testing as a career also needs training for various purposes, such as to understand (i) various phases of software testing life cycle, (ii) recent techniques to design test cases, (iii) various tools and how to work on them, etc. This is the reason that various testing courses for certified testers are being run.

4. GOALS OF SOFTWARE TESTING:

The goals of software testing may be classified into three major categories, as shown in the following figure:



Short-term or immediate goals: These goals are the immediate results after performing testing. These goals may be set in the individual phases of SDLC. Some of them are discussed below.

- **Bug discovery:** The immediate goal of testing is to find errors at any stage of software development. More the bugs discovered at an early stage, better will be the success rate of software testing.
- **Bug prevention:** It is the consequent action of bug discovery. From the behavior and interpretation of bugs discovered, everyone in the software development team gets to learn how to code safely such that the bugs discovered should not be repeated in later stages or future projects.

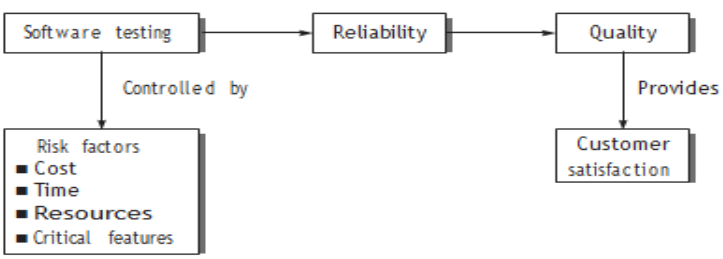
Long-term goals: These goals affect the product quality in the long run, when one cycle of the SDLC is over. Some of them are discussed here.

- **Quality** Since software is also a product; its quality is primary from the users' point of view. Thorough testing ensures superior quality. Therefore, the first goal of understanding and performing the testing process is to enhance the quality of the software product. Though quality depends on various factors, such as correctness, integrity, efficiency, etc., reliability is the major factor to achieve quality.

- **Reliability** is a matter of confidence that the software will not fail, and this level of confidence increases with rigorous testing. The confidence in reliability, in turn, increases the quality.

- **Customer satisfaction**: From the users perspective, the prime concern of testing is customer satisfaction only. If we want the customer to be satisfied with the software product, then testing should be complete and thorough. Testing should be complete in the sense that it must satisfy the user for all the specified requirements mentioned in the user manual, as well as for the unspecified requirements which are otherwise understood. A complete testing process achieves reliability, reliability enhances the quality, and quality in turn, increases the customer satisfaction.

- **Risk management**: Risk is the probability that undesirable events will occur in a system. Hence, it is the testers' responsibility to evaluate business risks (such as cost, time, resources, and critical features of the system being developed) and make the same a basis for testing choices. Testers should also categorize the levels of risks after their assessment (like high-risk, moderate-risk, low-risk) and this analysis becomes the basis for testing activities. Thus, risk management becomes the long-term goal for software testing.



Testing controlled by risk factors

Post-implementation goals: These goals are important after the product is released. Some of them are discussed here.

- **Reduced maintenance cost**: The maintenance cost of any software product is not its physical cost, as the software does not wear out. The only maintenance cost in a software product is its failure due to errors. Post-release errors are costlier to fix, as they are difficult to detect. Thus, if testing has been done rigorously and effectively, then the chances of failure are minimized and in turn, the maintenance cost is reduced.

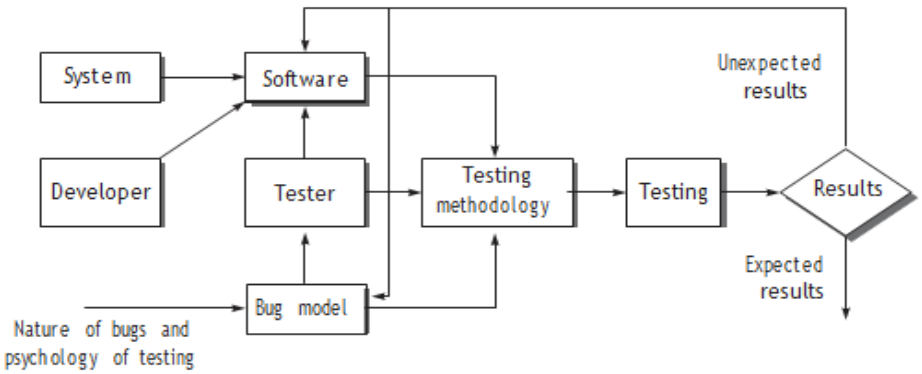
- **Improved software testing process**: A testing process for one project may not be successful and there may be scope for improvement. Therefore, the bug history and post-implementation results can be analyzed to find out snags in the present testing process, which can be rectified in future projects. Thus, the long-term post-implementation goal is to improve the testing process for future projects.

4. PSYCHOLOGY FOR SOFTWARE TESTING: Software testing is directly related to human psychology. Though software testing has not been defined till now, but most frequently, it is defined as, Testing is the process of demonstrating that there are no errors.

The purpose of testing is to show that the software performs its intended functions correctly. This definition is correct, but partially. If testing is per- formed keeping this goal in mind, then we cannot achieve the desired goals, as we will not be able to test the software as a whole. Myers first identified this approach of testing the software. This approach is based on the human psychology that human beings tend to work according to the goals fixed in their minds. On the other hand, if our goal is to demonstrate that a program has errors, then we will design test cases having a higher probability to uncover bugs.

Thus, if the process of testing is reversed, such that we always presume the presence of bugs in the software, then this psychology of being always suspicious of bugs widens the domain of testing. It means, now we don't think of testing only those features or specifications which have been mentioned in documents like SRS (software requirement specification), but we also think in terms of finding bugs in the domain or features which are understood but not specified. Thus, software testing may be defined as, Testing is the process of executing a program with the intent of finding errors. According to this psychology of testing, a successful test is that which finds errors.

5. MODEL FOR SOFTWARE TESTING: Testing should be performed in a planned way. For the planned execution of a testing process, we need to consider every element and every aspect related to software testing. Thus, in the testing model, we consider the related elements and team members involved



Testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed. Based on the software type and the bug model, testers decide a testing methodology which guides how the testing will be performed. With suitable testing techniques decided in the testing methodology, testing is performed on the software with a particular goal.

Software and Software Model: Software is built after analyzing the system in the environment. It is a complex entity which deals with environment, logic, programmer psychology, etc. But complex software makes it very difficult to test. Since in this model of testing, our aim is to concentrate on the testing process, therefore the software under consideration should not be so complex such that it would not be tested. In fact, this is the point of consideration for developers who design the software. They should design and code the software such that it is testable at every point. Thus, the software to be tested may be modeled such that it is testable, avoiding unnecessary complexities.

Bug Model: Bug model provides a perception of the kind of bugs expected. Considering the nature of all types of bugs, a bug model can be prepared that may help in deciding a testing strategy. However, every type of bug cannot be predicted. Therefore, if we get incorrect results, the bug model needs to be modified.

Testing methodology and Testing: Based on the inputs from the software model and the bug model, testers can develop a testing methodology that incorporates both testing strategy and testing tactics. Testing strategy is the roadmap that gives us well-defined steps for the overall testing process. It prepares the planned steps based on the risk factors and the testing phase. Once the planned steps of the testing process are prepared, software testing techniques and testing tools can be applied within these steps. Thus, testing is performed on this methodology. However, if we don't get the required results, the testing plans must be checked and modified accordingly.

6. EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING:

Exhaustive or complete software testing means that every statement in the program and every possible path combination with every possible combination of data must be executed. But soon, we will realize that exhaustive testing is out of scope. That is why the questions arise: (i) When are we done with testing? or (ii) How do we know that we have tested enough? There may be many answers for these questions with respect to time, cost, customer, quality, etc. exhaustive or complete testing is not possible. Therefore, we should concentrate on effective testing which emphasizes efficient techniques to test the software so that important features will be tested within the constrained resources.

Why complete testing is not possible.

The Domain of Possible Inputs to the Software is too Large to Test

If we consider the input data as the only part of the domain of testing, even then, we are not able to test the complete input data combination.

The do- main of input data has four sub-parts:

(a) valid inputs,

(b) invalid inputs,

(c) edited inputs, and

(d) race condition inputs

Valid inputs: It seems that we can test every valid input on the software. But look at a very simple example of adding two-digit two numbers. Their range is from –99 to 99 (total 199). So the total number of test case combinations will be 199 × 199 = 39601. Further, if we increase the range from two digits to four-digits, then the number of test cases will be

399,960,001. Most addition programs accept 8 or 10 digit numbers or more. How can we test all these combinations of valid inputs?

Invalid inputs: Testing the software with valid inputs is only one part of the input sub-domain. There is another part, invalid inputs, which must be tested for testing the software effectively. The important thing in this case is the behavior of the program as to how it responds when a user feeds invalid inputs. The set of invalid inputs is also too large to test. If we consider again the example of adding two numbers, then the following possibilities may occur from invalid inputs:

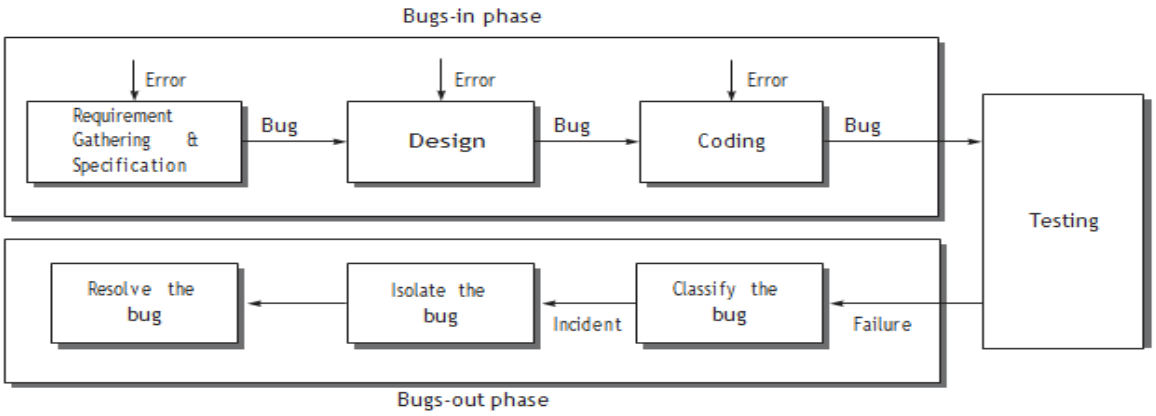
- Numbers out of range
- Combination of alphabets and digits
- Combination of all alphabets
- Combination of control characters
- Combination of any other key on the keyboard

Edited inputs: If we can edit inputs at the time of providing inputs to the program, then many unexpected input events may occur. For example, you can add many spaces in the input, which are not visible to the user. It can be a reason for non-functioning of the program.

Race condition inputs: The timing variation between two or more inputs is also one of the issues that limit the testing. For example, there are two input events,
A and B. According to the design, A precedes B in most of the cases. But, B can also come first in rare and restricted conditions. This is the race condition, whenever B precedes A. Usually the program fails due to race conditions, as the possibility of preceding B in restricted condition has not been taken care, resulting in a race condition bug. In this way, there may be many race conditions in the system, especially in multiprocessing systems and interactive systems. Race conditions are among the least tested.

7. LIFE CYCLE OF A BUG:

The whole life cycle of a bug can be classified into two phases: (i) bugs-in phase and (ii) bugs-out phase.



Life cycle of a bug

Bugs-In Phase: This phase is where the errors and bugs are introduced in the software. Whenever we commit a mistake, it creates errors on a specific location of the software and consequently, when this error goes unnoticed, it causes some conditions to fail, leading to a bug in the software. This bug is carried out to the subsequent phases of SDLC, if not detected. Thus, a phase may have its own errors as well as bugs received from the previous phase. If you are not performing verification on earlier phases, then there is no chance of detecting these bugs.

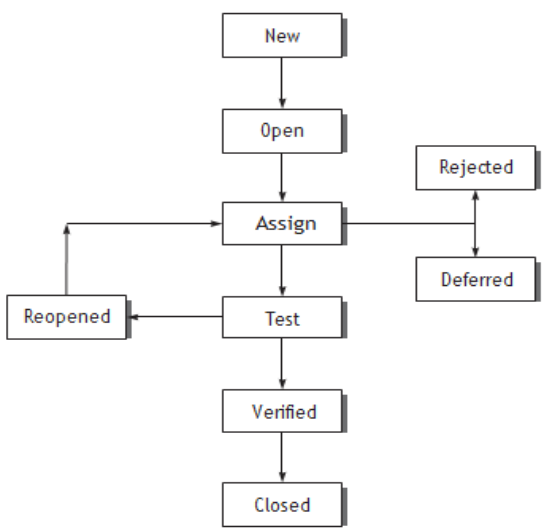
Bugs-Out Phase: If failures occur while testing a software product, we come to the conclusion that it is affected by bugs. However, there are situations when bugs are present, even though we don't observe any failures. That is another issue of discussion. In this phase, when we observe failures, the following activities are performed to get rid of the bugs.

7.1 STATES OF A BUG

Bug attains the following different states in its life cycle.

- **New**: The state is new when the bug is reported first time by a tester.
- **Open:** The new state does not verify that the bug is genuine. When the test leader approves that the bug is genuine, its state becomes open.
- **Assign**: An open bug comes to the development team where the development team verifies its validity. If the bug is valid, a developer is assigned the job to fix it and the state of the bug now is 'ASSIGN'.

- **Deferred**: The developer who has been assigned to fix the bug will check its validity and priority. If the priority of the reported bug is not high or there is not sufficient time to test it or the bug does not have any adverse effect on the software, then the bug is changed to deferred state which implies the bug is expected to be fixed in next releases.
- **Rejected:** It may be possible that the developer rejects the bug after checking its validity, as it is not a genuine one.
- **Test:** After fixing the valid bug, the developer sends it back to the testing team for next round of checking. Before releasing to the testing team, the developer changes the bug's state to 'TEST'. It specifies that the bug has been fixed by the development team but not tested and is released to the testing team.
- **Verified/fixed:** The tester tests the software and verifies whether the reported bug is fixed or not. After verifying, the developer approves that the bug is fixed and changes the status to 'VERIFIED'.
- **Reopened**: If the bug is still there even after fixing it, the tester changes its status to 'REOPENED'. The bug traverses the life cycle once again. In another case, a bug which has been closed earlier may be reopened if it appears again. In this case, the status will be REOPENED instead of OPEN.
- **Closed**: Once the tester and other team members are confirmed that the bug is completely eliminated, they change its status to 'CLOSED'.



States of a bug

7.2 BUG CLASSIFICATION BASED ON CRITICALITY:

Bugs can be classified based on the impact they have on the software under test. This classification can be used for the prioritization of bugs, as all bugs cannot be resolved in one release.

- **Critical Bugs:** This type of bugs has the worst effect such that it stops or hangs the normal functioning of the software. The person using the software becomes helpless when this type of bug appears. For example, in a sorting program, after providing the input numbers, the system hangs and needs to be reset.

- **Major Bugs:** This type of bug does not stop the functioning of the software but it causes a functionality to fail to meet its requirements as expected. For example, in a sorting program, the output is being displayed but not the correct one.

- **Medium Bugs:** Medium bugs are less critical in nature as compared to critical and major bugs. If the outputs are not according to the standards or conventions, e.g. redundant or truncated output, then the bug is a medium bug.

- **Minor Bugs:** These types of bugs do not affect the functionality of the software. These are just mild bugs which occur without any effect on the expected behavior or continuity of the software. For example, typographical error or misaligned printout.

BUG CLASSIFICATION BASED ON SDLC:

Since bugs can appear in any phase of SDLC, they can be classified based on SDLC phases which are described below:

- **Requirements and Specifications Bugs:** The first type of bug in SDLC is in the requirement gathering and specification phase. It has been observed that most of the bugs appear in this phase only. There may be a possibility that requirements specified are not exactly what the customers want. Moreover, specified requirements may be incomplete, ambiguous, or inconsistent. Specification problems lead to wrong missing, or superfluous features.

- **Design Bugs:** Design bugs may be the bugs from the previous phase and in addition those errors which are introduced in the present phase. The following design errors may be there:
- Control flow bugs, Logic bugs, Processing bugs, Data flow bugs, Error handling bugs, Race condition bugs Race conditions, Boundary-related bugs, User interface

bugs

- **Coding Bugs:** There may be a long list of coding bugs. If you are a programmer, then you are aware of some common mistakes made. For example, undeclared data, undeclared routines, dangling code, typographical errors, documentation bugs, i.e. erroneous comments lead to bugs in maintenance.

- **Interface and Integration Bugs:** External interface bugs include invalid timing or sequence assumptions related to external signals, misunderstanding external input and output formats, and user interface bugs. Internal interface bugs include input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call parameter bugs, and misunderstood entry or exit parameter values.

- **Integration bugs**: Result from inconsistencies or incompatibilities between modules discussed in the form of interface bugs. There may be bugs in data transfer and data sharing between the modules.

- **System Bugs:** There may be bugs while testing the system as a whole based on various parameters like performance, stress, compatibility, usability, etc. For example, in a real-time system, stress testing is very important, as the system must work under maximum load. If the system is put under maximum load at every factor like maximum number of users, maximum memory limit, etc. and if it fails, then there are system bugs.
- **Testing Bugs:** After all, testing is also performed by testers – humans. Some testing mistakes are: failure to notice/report a problem, failure to use the most promising test case, failure to make it clear how to reproduce the problem, failure to check for unresolved problems just before the release, failure to verify fixes, failure to provide summary report.

8 TESTING PRINCIPLES

These principles can be seen as guidelines for a tester.

1. **Effective testing, not exhaustive testing:** All possible combinations of tests become so large that it is impractical to test them all. So considering the domain of testing as infinite, exhaustive testing is not possible. Therefore, the tester's approach should be based on effective testing to adequately cover program logic and all conditions in the component level design.

2. **Testing is not a single phase performed in SDLC**: Testing is not just an activity performed after the coding in SDLC. As discussed, the testing phase after coding is just a part of the whole testing process. Testing process starts as soon.
3. **Destructive approach for constructive testing**: Testers must have the psychology that bugs are always present in the program and they must think about the technique of how to uncover them. This psychology of being always suspicious about bugs is a negative/destructive approach.
4. **Early testing is the best policy:** Testing starts as soon as requirement specifications are prepared. Moreover, the cost of bugs can be reduced tenfold, as bugs are harder to detect in later stages if they go undetected. Thus, the policy in testing is to start as early as possible.
5. **Probability of existence of an error in a section of a program is proportional to the number of errors already found in that section**: Suppose the history of a software is that you found 50 errors in Module X, 12 in Module Y, and 3 in Module Z. The software was debugged but after a span of time, we find some errors again and the software is given to a tester for testing. Where should the tester concentrate to find the bugs? This principle says that the tester should start with Module X which has the history of maximum errors.
6. **Testing strategy should start at the smallest module level and expand towards the whole program**: This principle supports the idea of incremental testing. Testing must begin at the unit or module level, gradually progressing towards integrated modules and finally the whole system.
7. **Testing should also be performed by an independent team**: When programmers develop the software, they test it at their individual modules. However, these programmers are not good testers of their own software.
8. **Everything must be recorded in software**: Testing demands that every detail be recorded and documented. We must have the record of every test case run and the bugs reported. Even the inputs provided during testing and the corresponding outputs are to be recorded. Executing the test cases in a recorded and documented way can greatly help while observing the bugs.
9. **Invalid inputs and unexpected behavior have a high probability of finding an error** whenever the software is tested, we test for valid inputs and for the functionality that the software is supposed to do. But thinking in a negative way, we must test the software with invalid inputs and the behavior which is not expected in general. This is also a part of effective testing.
10. **Testers must participate in specification and design reviews**: Testers' role is not only to get the software and documents and test them. If they are not participating in other

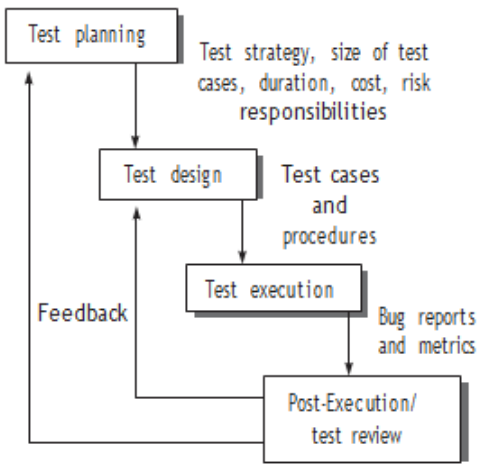
reviews like specification and design, it may be possible that either some specifications are not tested or some test cases are built for no specifications.

9. SOFTWARE TESTING LIFE CYCLE (STLC): The testing process divided into a well-defined sequence of steps is termed as software testing life cycle (STLC). This has a significant benefit in the project schedule and cost. The STLC also helps the management in measuring specific milestones.

STLC consists of the following phases:

Test Planning: The goal of test planning is to take into account the important issues of testing strategy, viz. resources, schedules, responsibilities, risks, and priorities, as a roadmap. Test planning issues are in tune with the overall project planning. Broadly, following are the activities during test planning:

Defining the test strategy: Estimate the number of test cases, their duration, and cost. Plan the resources like the manpower to test, tools required, documents required. Identifying areas of risks, defining the test completion criteria, Identification of methodologies, techniques, and tools for various test cases. The major output of test planning is the test plan document. Test plans are developed for each level of testing. After analyzing the issues, the following activities are performed:



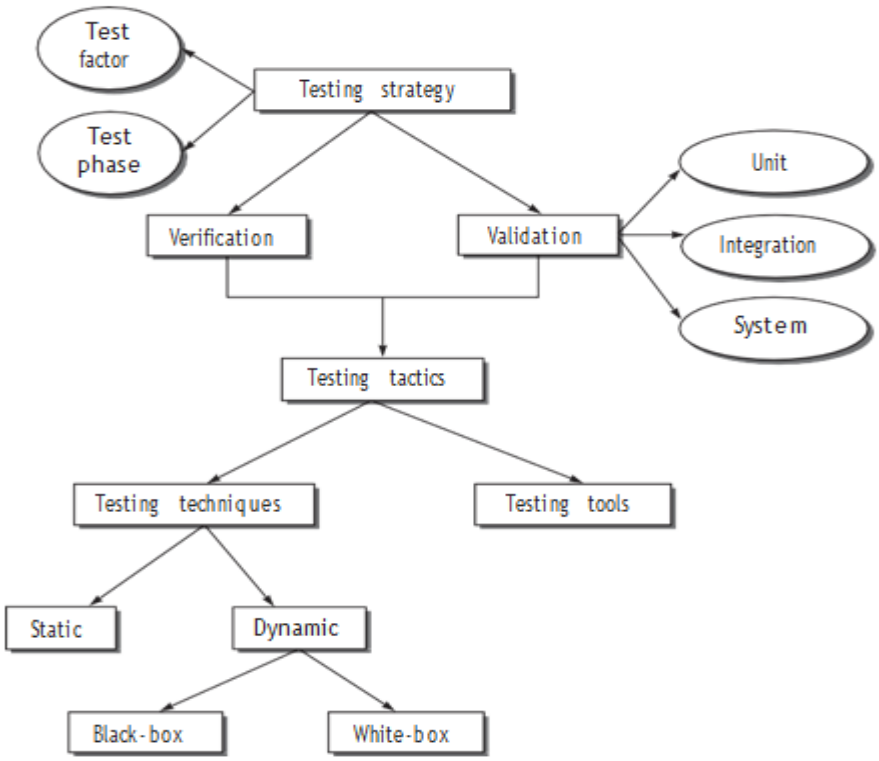
Software testing life cycle

Test Design: One of the major activities in testing is the design of test cases. However, this activity is not an intuitional process; rather it is a well-planned process.

Test Execution: In this phase, all test cases are executed including verification and validation. Verification test cases are started at the end of each phase of SDLC. Validation test cases are started after the completion of a module.

Post-Execution/Test Review: As we know, after successful test execution, bugs will be reported to the concerned developers. This phase is to analyse bug-related issues and get feed-back so that maximum number of bugs can be removed. This is the primary goal of all test activities done earlier.

11. **SOFTWARE TESTING METHODOLOGY:** Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved, as shown in Figure.



Testing methodology

SOFTWARE TESTING STRATEGY: Testing strategy is the planning of the whole testing

process into a well-planned series of steps. In other words, strategy provides a roadmap that includes very specific activities that must be performed by the test team in order to achieve a specific goal.

Test Factors: Test factors are risk factors or issues related to the system under development. Risk factors need to be selected and ranked according to a specific system under development. The testing process should reduce these test factors to a prescribed level.

Test Phase: This is another component on which the testing strategy is based. It refers to the phases of SDLC where testing will be performed. Testing strategy may be different for different models of SDLC, e.g. strategies will be different for waterfall and spiral models.

Validation Activities: Validation has the following three activities which are also known as the three levels of validation testing.

- **Unit Testing:** Unit testing is a basic level of testing which cannot be overlooked, and confirms the behavior of a single module according to its functional specifications.
- **Integration Testing:** It is a validation technique which combines all unit-tested modules and performs a test on their aggregation. But how do we integrate the units together? Is it a random process? It is actually a systematic technique for combining modules. In fact, interfacing among modules is represented by the system design. We integrate the units according to the design and availability of units. Therefore, the tester must be aware of the system design.
- **System Testing:** This testing level focuses on testing the entire integrated system. It incorporates many types of testing, as the full system can have various users in different environments. The purpose is to test the validity for specific users and environments. The validity of the whole system is checked against the requirement specifications.

Software testing techniques: Implement the test cases on the software. These techniques can be categorized into two parts: (a) static testing and (b) dynamic testing.

- **Static Testing:** It is a technique for assessing the structural characteristics of source code, design specifications or any notational representation that conforms to well-defined syntactic rules. It is called as static because we never execute the code in this technique. For example, the structure of code is examined by the teams but the code is not executed.
- **Dynamic Testing:** All the methods that execute the code to test a software are known as dynamic testing techniques. In this technique, the code is run on a number of inputs provided by the user and the corresponding results are checked. This type of testing is

further divided into two parts: (a) black-box testing and (b) white-box testing.

- **Black-box testing**: This technique takes care of the inputs given to a system and the output is received after processing in the system. What is being processed in the system? How does the system perform these operations? Black-box testing is not concerned with these questions. It checks the functionality of the system only. That is why the term black-box is used. It is also known as functional testing. It is used for system testing under validation.
- **White-box testing**: This technique complements black-box testing. Here, the system is not a black box. Every design feature and its corresponding code is checked logically with every possible path execution. So, it takes care of the structural paths instead of just outputs. It is also known as structural testing and is used for unit testing under verification.

Testing Tools:

Testing tools provide the option to automate the selected testing technique with the help of tools. A tool is a resource for performing a test process. The combination of tools and testing techniques enables the test process to be per- formed. The tester should first understand the testing techniques and then go for the tools that can be used with each of the techniques.

UNIT-II

2.1 VERIFICATION AND VALIDATION (V&V) ACTIVITIES

V&V activities can be understood in the form of SDLC phases.

Requirements gathering: The needs of the user are gathered and translated into a written set of requirements. These requirements are prepared from the user's viewpoint only and do not include any technicalities according to the developer.

Requirement specification or objectives: In this phase, all the user requirements are specified in developer's terminology. The specified objectives from the full system which is going to be developed are prepared in the form of a document known as software requirement specification (SRS).

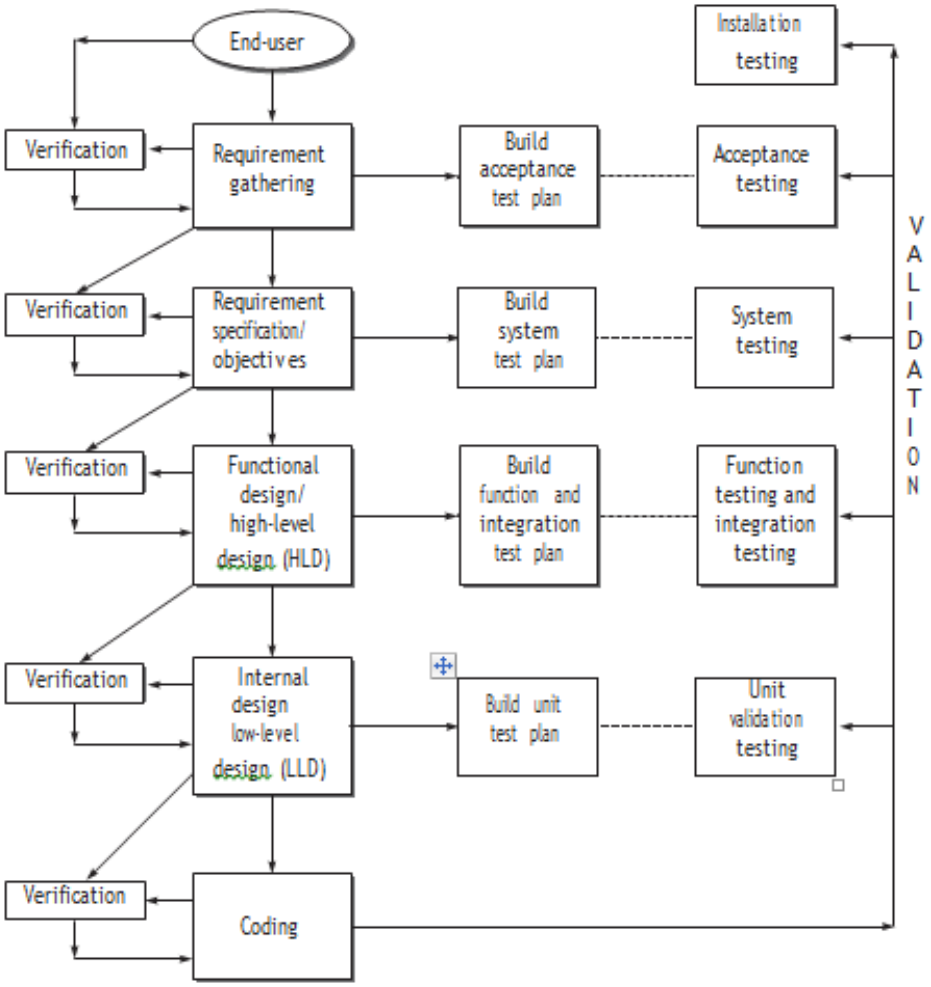
Functional design or high-level design: Functional design is the process of translating user requirements into a set of external interfaces. The output of the process is the functional design specification, which describes the product's behaviour as seen by an observer external to the product. The high-level design is prepared with SRS and software analysts convert the requirements into a usable product. In HLD, the software system architecture is prepared and broken into independent modules. Thus, an HLD document will contain the following items at a macro level:

- Overall architecture diagrams along with technology details
- Functionalities of the overall system with the set of external interfaces
- List of modules
- Brief functionality of each module
- Interface relationship among modules including dependencies between modules, database tables identified along with key elements

Internal design or low-level design: Since HLD provides the macro-level details of a system, an HLD document cannot be given to programmers for coding. So the analysts prepare a micro-level design document called internal design or low-level design (LLD). This document describes each and every module in an elaborate manner, so that the programmer can directly code the program based on this. There may be at least one separate document for each module.

Coding If an LLD document is prepared for every module, then it is easy to code the module. Thus in this phase, using design document for a module, its coding is done.

After understanding all the SDLC phases, we need to put together all the verification activities. As described earlier, verification activities are performed almost at every phase, therefore all the phases described above will be verified, as shown in Fig. Along with these verification activities performed at every step, the tester needs to prepare some test plans which will be used in validation activities performed after coding the system. These test plans are prepared at every SDLC phase.



V&V diagram

2.2 VERIFICATION ACTIVITIES

All the verification activities are performed in connection with the different phases of SDLC. The following verification activities have been identified:

1. Verification of Requirements and Objectives
2. Verification of High-Level Design
3. Verification of Low-Level Design
4. Verification of Coding (Unit Verification)

1. VERIFICATION OF REQUIREMENTS AND OBJECTIVES:

VERIFICATION OF REQUIREMENTS:

In this type of verification, all the requirements gathered from the user's viewpoint are verified. The tester works in parallel by performing the following two tasks:

- The tester reviews the acceptance criteria in terms of its completeness, clarity, and

testability. Moreover, the tester understands the proposed system well in advance so that necessary resources can be planned for the project.

- The tester prepares the Acceptance Test Plan which is referred at the time of Acceptance Testing

VERIFICATION OF OBJECTIVES

After gathering requirements, specific objectives are prepared considering every specification. These objectives are prepared in a document called software requirement specification (SRS). In this activity also, two parallel activities are performed by the tester:

1. The tester verifies all the objectives mentioned in SRS. The purpose ofthis verification is to ensure that the user's needs are properly under- stood before proceeding with the project.
2. The tester also prepares the System Test Plan which is based on SRS. This plan will be referenced at the time of System Testing.

Requirement and objectives verification has a high potential of detecting bugs. Therefore, requirements must be verified. As stated above, the testers use the SRS for verification of objectives. One characteristic of a good SRS is that it can be verified. An SRS can be verified, if and only if, every requirement stated herein can be verified. A requirement can be verified, if, and only if, there is some procedure to check that the software meets its requirement. It is a good idea to specify the requirements in a quantification manner.

It is clear now that verification starts from the requirement phase and every requirement specified in the SRS must be verified. But what are the points against which verification of requirement will be done? Following are the points against which every requirement in SRS should be verified:

Correctness: There are no tools or procedures to measure the correctness of a specification. The tester uses his or her intelligence to verify the correctness of requirements. Following are some points which can be adopted (these points can change according to situation):
Testers should refer to other documentations or applicable standards and compare the specified requirement with them.

Unambiguous: A requirement should be verified such that it does not provide too many meanings or interpretations.

- Every requirement has only one interpretation.
- Each characteristic of the final product is described using a single unique term.

Consistent: No specification should contradict or conflict with another. Conflicts produce bugs in the next stages, therefore they must be checked for the following:

- Real-world objects conflict, for example, one specification recommends mouse for input, another recommends joystick.
- Logical conflict between two specified actions, e.g. one specification requires the function

to perform square root, while another specification requires the same function to perform square operation.

- Conflicts in terminology should also be verified. For example, at one place, the term process is used while at another place, it has been termed as task or module.

Completeness: The requirements specified in the SRS must be verified for completeness. We must verify that all significant requirements such as functionality, performance, design constraints, attribute, or external interfaces are complete.

- Check whether responses of every possible input (valid & invalid) to the software have been defined.
- Check whether figures and tables have been labeled and referenced completely.

Updation:

- Requirement specifications are not stable, they may be modified or another requirement may be added later. Therefore, if any updation is there in the SRS, then the updated specifications must be verified.
- If the specification is a new one, then all the above mentioned steps and their feasibility should be verified.
- If the specification is a change in an already mentioned specification, then we must verify that this change can be implemented in the current design.

Traceability: The traceability of requirements must also be verified such that the origin of each requirement is clear and also whether it facilitates referencing in future development or enhancement documentation. The following two types of traceability must be verified:

- Backward traceability Check that each requirement references its source in previous documents.
- Forward traceability Check that each requirement has a unique name or reference number in all the documents. Forward traceability assumes more meaning than this, but for the sake of clarity, here it should be understood in the sense that every requirement has been recognized in other documents.

3. **VERIFICATION OF HIGH-LEVEL DESIGN**: Like the verification of requirements, the tester is responsible for two parallel activities in this phase as well:
   - The tester verifies the high-level design.
   - The tester also prepares a Function Test Plan which is based on the SRS. This plan will be referenced at the time of Function Testing (discussed later).
   - The tester also prepares an Integration Test Plan which will be referred at the time of integration testing (discussed later).

HOW TO VERIFY HIGH-LEVEL DESIGN?

Verification of Data Design: The points considered for verification of data design are as follows:

- Check whether the sizes of data structure have been estimated appropriately.
- Check the provisions of overflow in a data structure.
- Check the consistency of data formats with the requirements.
- Check whether data usage is consistent with its declaration.
- Check the relationships among data objects in data dictionary.
- Check the consistency of databases and data warehouses with the requirements specified in SRS.

Verification of Architectural Design: The points considered for the verification of architectural design are:

- Check that every functional requirement in the SRS has been take care of in this design.
- Check whether all exceptions handling conditions have been taken care of.
- Verify the process of transform mapping and transaction mapping, used for the transition from requirement model to architectural design.
- Since architectural design deals with the classification of a system into sub-systems or modules, check the functionality of each module according to the requirements specified.
- Check the inter-dependence and interface between the modules.
- In the modular approach of architectural design, there are two issues with modularity—Module Coupling and Module Cohesion. A good design will have low coupling and high cohesion. Testers should verify these factors, otherwise they will affect the reliability and maintainability of the system which are non-functional requirements of the system.

Verification of User-Interface Design: The points to be considered for the verification of user-interface design are:

- Check all the interfaces between modules according to the architecture design.
- Check all the interfaces between software and other non-human producer and consumer of information.
- Check all the interfaces between human and computer.
- Check all the above interfaces for their consistency.
- Check the response time for all the interfaces are within required ranges. It is very essential for the projects related to real-time systems where response time is very crucial.
- For a Help Facility, verify the following:
    - The representation of Help in its desired manner
    - The user returns to the normal interaction from Help
- For error messages and warnings, verify the following:
    - Whether the message clarifies the problem
    - Whether the message provides constructive advice for recovering from the error

4. **VERIFICATION OF LOW-LEVEL DESIGN:** In this verification, low-level design phase is considered. The abstraction level in this phase is low as compared to high-level design. In LLD, a detailed de- sign of modules and data are prepared such that an operational software is ready. For this, SDD is preferred where all the modules and their interfaces are defined. Every operational detail of each module is prepared. The details of each module or unit are prepared in their separate SRS and SDD.

HOW TO VERIFY LOW-LEVEL DESIGN?

- This is the last pre-coding phase where internal details of each design entity are described. For verification, the SRS and SDD of individual modules are referred to. Some points to be considered are listed below:
- Verify the SRS of each module.
- Verify the SDD of each module.
- In LLD, data structures, interfaces, and algorithms are represented by design notations; verify the consistency of every item with their design notations.
- Organizations can build a two-way traceability matrix between the SRS and design (both HLD and LLD) such that at the time of verification of de- sign, each requirement mentioned in the SRS is verified. In other words, the traceability matrix provides a one-to-one mapping between the SRS and the SDD.

5. **VERIFICATION OF CODE:** Coding is the process of converting LLD specifications into a specific language. This is the last phase when we get the operational software with the source code. People have the impression that testing starts only after this phase. However, it has been observed in the last chapter that testing starts as soon as the requirement specifications are given and testers perform parallel activities during every phase of SDLC. If we start testing after coding, then there is a possibility that requirement specifications may have bugs and these might have been passed into the design and consequently into the source code. Therefore, the operational software which is ready now is not reliable and when bugs appear after this phase, they are very expensive to fix.
   - Since low-level design is converted into source code using some language, there is a possibility of deviation from the LLD. Therefore, the code must also be verified. The points against which the code must be verified are:
   - Check that every design specification in HLD and LLD has been coded using traceability matrix.
   - Examine the code against a language specification checklist.

   . Some points against which the code can be verified are:
      - Misunderstood or incorrect arithmetic precedence
      - Mixed mode operations
      - Incorrect initialization
      - Precision inaccuracy

- Incorrect symbolic representation of an expression
- Different data types
- Improper or non-existent loop termination
- Failure to exit
- Two kinds of techniques are used to verify the coding:
- (a) static testing, and
- (b) dynamic testing.
    - **Static testing techniques:** This technique does not involve actual execution. It considers only static analysis of the code or some form of conceptual execution of the code.
    - **Dynamic testing techniques:** It is complementary to the static testing technique. It executes the code on some test data. The developer is the key person in this process who can verify the code of his module by using the dynamic testing technique.

2.3 VALIDATION

- Validation is a set of activities that ensures the software under consideration has been built right and is traceable to customer requirements. Validation testing is performed after the coding is over.
- What is the need for validation? When every step of SDLC has been verified, why do we want to test the product at the end? The reasons are:
- To determine whether the product satisfies the users' requirements, as stated in the requirement specification.
- To determine whether the product's actual behaviour matches the de- sired behaviour, as described in the functional design specification.
- It is not always certain that all the stages till coding are bug-free. The bugs those are still present in the software after the coding phase need to be uncovered.
- Validation testing provides the last chance to discover bugs, otherwise these bugs will move to the final product released to the customer.
- Validation enhances the quality of software.

VALIDATION ACTIVITIES: The validation activities are divided into Validation Test Plan and Validation Test Execution which are described as follows:

- **Validation Test Plan:** It starts as soon as the first output of SDLC, i.e. the SRS, is prepared. In every phase, the tester performs two parallel activities—verification at that phase and the corresponding validation test plan. For preparing a validation test plan, testers must follow the points described below.

- **Acceptance test plan**: This plan is prepared in the requirement phase according to the acceptance criteria prepared from the user feedback. This plan is used at the time of Acceptance Testing.

- **System test plan**: This plan is prepared to verify the objectives specified in the SRS. Here, test cases are designed keeping in view how a complete integrated system will work or behave in different conditions. The plan is used at the time of System Testing.
- **Function test plan:** This plan is prepared in the HLD phase. In this plan, test cases are designed such that all the interfaces and every type of functionality can be tested. The plan is used at the time of Function Testing.
- **Integration test plan**: This plan is prepared to validate the integration of all the modules such that all their interdependencies are checked. It also validates whether the integration is in conformance to the whole system design. This plan is used at the time of Integration Testing.
- **Unit test plan**: This plan is prepared in the LLD phase. It consists of a test plan of every module in the system separately. Unit test plan of every unit or module is designed such that every functionality related to individual unit can be tested. This plan is used at the time of Unit Testing.

Validation Test Execution

Validation test execution can be divided in the following testing activities:

- **Unit validation testing:** The testing strategy is to first focus on the smaller building blocks of the full system. One unit or module is the basic building block of the whole software that can be tested for all its interfaces and functionality.
- **Integration testing:** It is the process of combining and testing multiple components or modules together. The individual tested modules, when combined with other modules, are not tested for their interfaces. Therefore, they may contain bugs in integrated environment. Thus, the intention here is to uncover the bugs that are present when unit tested modules are integrated.
- **Function testing:** When the integrated system has been tested, all the specified functions and their external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications. An external specification is a precise description of the software behaviour from the viewpoint of the outside world (e.g. user). Thus, function testing is to explore the bugs related to discrepancies between the actual system behaviour and its functional specifications.
- **System testing**: It is different from function testing, as it does not test every function. System testing is actually a series of different tests whose primary purpose is to fully exercise a computer-based system [7]. System testing does not aim to test the specified function, but its intention is to test the whole system on various grounds where bugs may occur. For example, if the software fails in some conditions, how does it recover? Another example is protection from improper penetration, i.e. how secure the whole system is.
- **Acceptance testing**: In the requirement phase of SDLC, acceptance criteria for the system to be developed are mentioned in one contract by the customer. When the system is ready, it can be tested against

- **Installation testing**: Once the testing team has given the green signal for producing the software, the software is placed into an operational status where it is installed. The installation testing does not test the system, but it tests the process of making the software system operational. The installation process must identify the steps required to install the system. For example, it may be required that some files may have to be converted into another format. Or it may be possible that some operating software may be required. Thus, installation testing tests the interface to operating software, related software, and any operating procedures.

3. Dynamic Testing: Black-Box Testing Techniques

Black-box technique is one of the major techniques in dynamic testing for designing effective test cases. This technique considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered. Therefore, this is also known as functional testing. The software system is considered as a black box, taking no notice of its internal structure, so it is also called as black-box testing technique.

Black-box testing attempts to find errors in the following categories:

- To test the modules independently
- To test the functional validity of the software so that incorrect or missing functions can be recognized
- To look for interface errors
- To test the system behaviour and check its performance
- To test the maximum load or stress on the system
- To test the software such that the user/customer accepts the system with- in defined acceptable limits.

3.1 BOUNDARY VALUE ANALYSIS (BVA): BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain. For example, if A is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254). Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101).

BVA offers several methods to design test cases as discussed in the following sections.

BOUNDARY VALUE CHECKING (BVC)

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

(a) Minimum value (Min)

(b) Value just above the minimum value (Min^+)

(c) Maximum value (Max)

(d) Value just below the maximum value (Max^-)

Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following test cases can be designed:

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

It can be generalized that for n variables in a module, $4n + 1$ test cases canbe designed with boundary value checking method.

ROBUSTNESS TESTING METHOD

The idea of BVC can be extended such that boundary values are exceededas:

- A value just greater than the Maximum value (Max^+)
- A value just less than Minimum value (Min^-)

When test cases are designed considering the above points in addition toBVC, it is called *robustness testing*.

Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:

10. A_{max+}, B_{nom}
11. A_{min-}, B_{nom}
12. A_{nom}, B_{max+}
13. A_{nom}, B_{min-}

It can be generalized that for n input variables in a module, $6n + 1$ test casescan be designed with robustness testing.

WORST-CASE TESTING METHOD

We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called worst-case testing method.

Again, take the previous example of two variables, A and B. We can add the following test

cases to the list of 9 test cases designed in BVC as:

10. A_{min}, B_{min}
11. A_{min+}, B_{min}
12. A_{min}, B_{min+}
13. A_{min+}, B_{min+}
14. A_{max}, B_{min}
15. $A_{max–}$, B_{min}
16. A_{max}, B_{min+}
17. $A_{max–}$, B_{min+}
18. A_{min}, B_{max}
19. A_{min+}, B_{max}
20. A_{min}, $B_{max–}$
21. A_{min+}, $B_{max–}$
22. A_{max}, B_{max}
23. $A_{max–}$, B_{max}
24. A_{max}, $B_{max–}$
25. $A_{max–}$, $B_{max–}$

It can be generalized that for n input variables in a module, 5^n test cases can be designed with worst-case testing.

Example:

A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) Test cases using BVC Since there is one variable, the total number of test cases will be $4n + 1 = 5$.

In our example, the set of minimum and maximum values is shownbelow:

Min value = 1
Min^+ value = 2
Max value = 100
Max^- value = 99
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

(b)Test cases using robust testing Since there is one variable, the total number of test cases will be $6n + 1 = 7$. The set of boundary values is shown below:

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

(c)Test cases using worst-case testing since there is one variable, the total number of test cases will be $5^n = 5$. Therefore, the number of test cases will be same as BVC.

3.2 EQUIVALENCE CLASS TESTING: We know that the input domain for testing is too large to test every input. So we can divide or partition the input domain based on a common feature or a class of data. Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed.

Equivalence partitioning method for designing test cases has the following goals:

Completeness: Without executing all the test cases, we strive to touch the completeness of testing domain.

Non-redundancy: When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug. Thus, the goal of equivalence partitioning method is to reduce these redundant test cases.

To use equivalence partitioning, one needs to perform two steps:

1. **Identify equivalence classes:**
Two types of classes can always be identified as discussed below:

- Valid equivalence classes these classes consider valid inputs to the program.
- Invalid equivalence classes one must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program.

2. **Design test cases:** A few guidelines are given below to identify test cases through generated equivalence classes:
    - Assign a unique identification number to each equivalence class.
    - Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.
    - Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases.

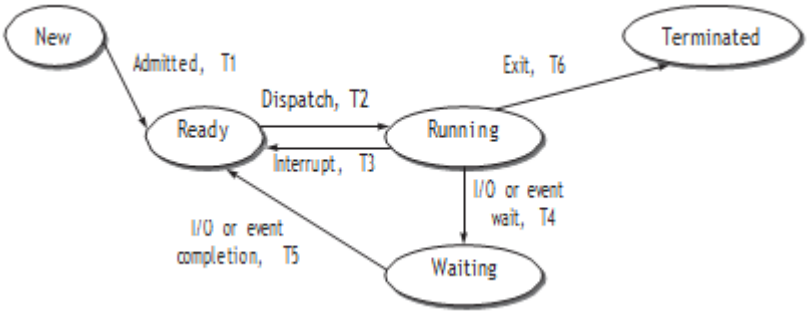
3.3 STATE TABLE-BASED TESTING: Tables are useful tools for representing and documenting many types of information relating to test case design. These are beneficial for the applications which can be described using state transition diagrams and state tables.

STATE TRANSITION DIAGRAMS OR STATE GRAPH

A system or its components may have a number of states depending on its input and time. For example, a task in an operating system can have the following states:

1. New State: When a task is newly created.
2. Ready: When the task is waiting in the ready queue for its turn.
3. Running: When instructions of the task are being executed by CPU.
4. Waiting: When the task is waiting for an I/O event or reception of asignal.
5. Terminated: The task has finished execution.

States are represented by nodes. Now with the help of nodes and transition links between the nodes, a state transition diagram or state graph is prepared. Whatever is being modeled is subjected to inputs. As a result of these inputs, when one state is changed to another, it is called a transition. Transitions are represented by links that join the nodes.



STATE TABLE: State graphs of larger systems may not be easy to understand. Therefore, state graphs are converted into tabular form for convenience sake, which are known as state tables. State tables also specify states, inputs, transitions, and outputs. The following conventions are used for state table

- Each row of the table corresponds to a state.
- Each column corresponds to an input condition.
- The box at the intersection of a row and a column specifies the next state (transition) and the output, if any.

State\Input Event	Admit	Dispatch	Interrupt	I/O or Event Wait	I/O or Event Wait Over	Exit
New	**Ready/ T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready/ T1	**Running/ T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running/T2	Running/ T2	**Ready / T3**	**Waiting/ T4**	Running/ T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting / T4	Waiting/T4	Waiting / T4	**Ready / T5**	Waiting / T4

The highlighted cells of the table are valid inputs causing a change of state.

The procedure for converting state graphs and state tables into test cases is discussed below.

- Identify the states: The number of states in a state graph is the number of states we choose to recognize or model.
- Prepare state transition diagram after understanding transitionsbetween states : After having all the states, identify the inputs on eachstate and transitions between states and prepare the state graph.
- Convert the state graph into the state table.
- Analyse the state table for its completeness.
- Create the corresponding test cases from the state table.

Deriving test cases from state table:

Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

3.4 DECISION TABLE-BASED TESTING: Decision table is another useful method to represent the information in a tabular method. It has the specialty to consider complex combinations of in- put conditions and resulting actions. Decision tables obtain their power from logical expressions. Each operand or variable in a logical expression takes on the value, TRUE or FALSE.

FORMATION OF DECISION TABLE

A decision table is formed with the following components

ENTRY

		Rule 1	Rule 2	Rule 3	Rule 4	…
Condition Stub	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

Decision table structure

Condition stub: It is a list of input conditions for which the complex combinationis made.

Action stub: It is a list of resulting actions which will be performed if a combination of input condition is satisfied.

Condition entry: It is a specific entry in the table corresponding to input conditions mentioned in the condition stub. When we enter TRUE or FALSE for all input conditions for a particular combination, then it is called a *Rule*. Thus, a rule defines which combination of conditions produces the resulting action.

Action entry: It is the entry in the table for the resulting action to be performed when one rule (which is a combination of input condition) is satisfied. 'X' denotes the action entry in the table.

TEST CASE DESIGN USING DECISION TABLE

For designing test cases from a decision table, following interpretations shouldbe done:

- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions, becomes the testcase itself.
- If there are *k* rules over *n* binary conditions, there are at least *k* test casesand at the most 2^n test cases.

Example:

A program calculates the total salary of an employee with the conditions thatif the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design test cases using decision table testing.

Solution: The decision table for the program is shown below:

ENTRY

		Rule 1	**Rule 2**	**Rule3**
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The test cases derived from the decision table are given below:

Test Case ID	**Working Hour**	**Day**	**Expected Result**
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.0 of salary

3.5 CAUSE-EFFECT GRAPHING BASED TESTING: As said earlier, boundary value analysis and equivalence class partitioning methods do not consider combinations of input conditions. Like decision tables, cause-effect graphing is another technique for combinations of input conditions. But cause-effect graphing takes the help of decision table to design a test case. Therefore, cause-effect graphing is the technique to represent the situations of combinations of input conditions and then we convert the cause- effect graph into decision table for the test cases.

The following process is used to derive the test cases:

- ***Identification of causes and effects:*** The next step is to identify *causes and effects* in the specifications. A cause is a distinct input condition identified in the problem. It may

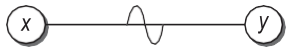
also be an equivalence class of input conditions. Similarly, an effect is an output condition.

- **Transformation of specification into a cause-effect graph**: Based on the analysis of the specification, it is transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph. Complete the graph by adding the constraints, if any, between causes and effects.
- **Conversion into decision table**: The cause-effect graph obtained is converted into a limited-entry decision table by verifying state conditions in the graph. Each column in the table represents a test case.
- **Deriving test cases:** The columns in the decision table are converted into test cases.

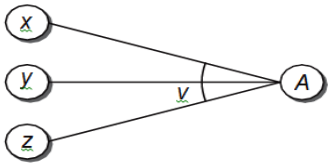
BASIC NOTATIONS FOR CAUSE-EFFECT GRAPH

Identity: According to the identity function, if *x* is 1, *y* is 1; else *y* is 0.

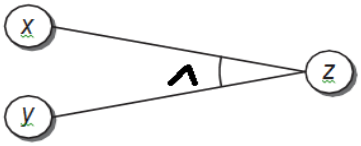
NOT: This function states that if *x* is 1, *y* is 0; else *y* is 1.



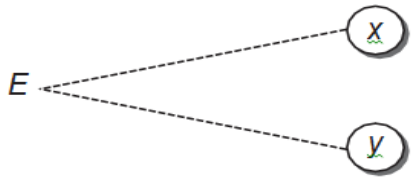
OR : The OR function states that if *x* or *y* or *z* is 1, *A* is 1; else *A* is 0.



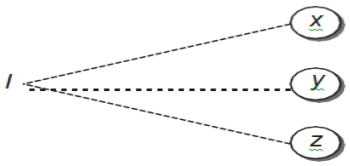
AND: This function states that if both *x* and *y* are 1, *z* is 1; else *z* is 0.



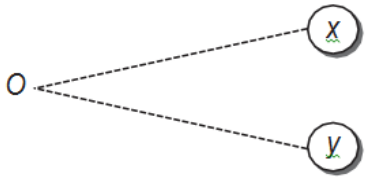
Exclusive: . According to this function, it always holds that either *x* or *y* can be 1, i.e. *x* and *y* cannot be 1 simultaneously.



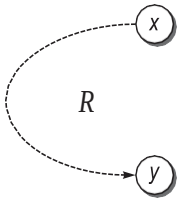
Inclusive: It states that at least one of *x*, *y*, and *z* must always be 1 (*x*, *y*, and *z* cannot be0 simultaneously).



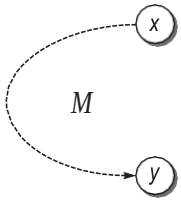
One and Only One: It states that one and only one of *x* and *y* must be 1.



Requires: It states that for *x* to be 1, *y* must be 1, i.e. it is impossible for *x* to be 1 and *y* to be 0.



Mask: It states that if *x* is 1, *y* is forced to 0.



Example: The "Print message" is software that reads two characters and, depending on their values, messages are printed.

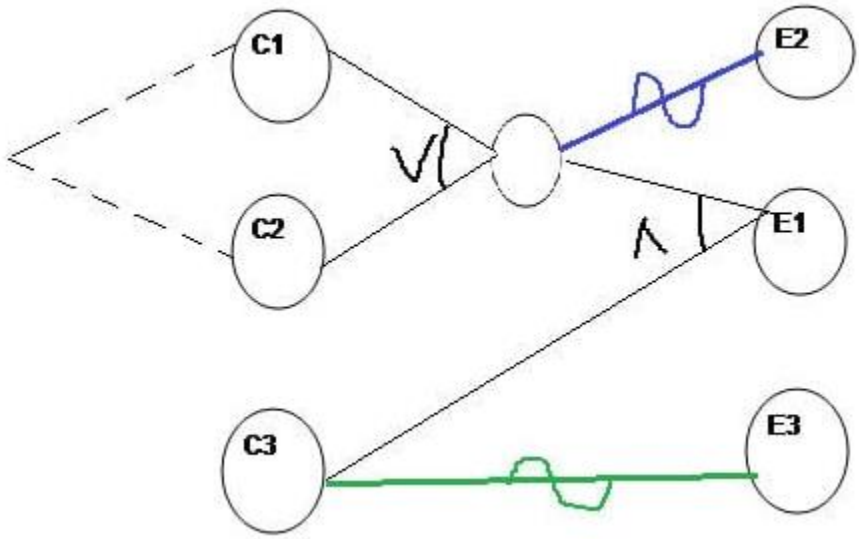
- The first character must be an "A" or a "B".
- The second character must be a digit.

- If the first character is an “A” or “B” and the second character is a digit, then the file must be updated.
- If the first character is incorrect (not an “A” or “B”), the message X must be printed.
- If the second character is incorrect (not a digit), the message Y must be printed.

Solution:

The Causes of this situation are:
C1 – First character is A
C2 – First character is B
C3 – the Second character is a digit

The Effects (results) for this situation are:
E1 – Update the file
E2 – Print message “X”
E3 – Print message “Y”



Writing Decision Table Based on Cause and Effect graph:

Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

Writing Test Cases From The Decision Table

Below is a sample test case for Test Case 1 (TC1) and Test Case 2 (TC2).

TC ID	TC Name	Description	Steps	Expected result
TC1	TC1_FileUpdate Scenario1	Validate that system updates the file when first character is A and second character is a digit.	1. Open the application. 2. Enter first character as "A" 3. Enter second character as a digit	File is updated.
TC2	TC2_FileUpdate Scenario2	Validate that system updates the file when first character is B and second character is a digit.	1. Open the application. 2. Enter first character as "B" 3. Enter second character as a digit	File is updated.

3.6 ERROR GUESSING:

Error guessing is the preferred method used when all other methods fail. Sometimes it is used to test some special cases. According to this method, errors or bugs can be guessed which do not fit in any of the earlier defined situations. So test cases are generated for these special cases.

It is a very practical case wherein the tester uses his intuition and makes a guess about where the bug can be. The tester does not have to use any particular testing technique. However, this capability comes with years of experience in a particular field of testing. This is the reason that experienced managers can easily smell out errors as compared to a novice tester.

The history of bugs can help in identifying some special cases in the project. There is a high probability that errors made in a previous project are repeated again. In these situations, error guessing is an ad hoc approach, based on intuition, experience, knowledge of project, and bug history. Any of these can help to expose the errors. The basic idea is to make a list of possible errors in error-prone situations and then develop the test cases. Thus, there is no general procedure for this technique, as it is largely an intuitive and ad hoc process.

UNIT-III

Dynamic Testing: White-Box Testing Techniques

White-box testing is another effective testing technique in dynamic testing. It is also known as glass-box testing. The entire design, structure, and code of the software have to be studied for this type of testing. It is obvious that the developer is very close to this type of testing. Often, developers use white-box testing techniques to test their own design and code. This testing is also known as structural or development testing. In white-box testing, structure means the logic of the program which has been implemented in the language code. The intention is to test this logic so that required results or functionalities can be achieved. Thus, white-box testing ensures that the internal parts of the software are adequately tested.

4.1 NEED OF WHITE-BOX TESTING:

Is white-box testing really necessary? Can't we write the code and simply test the software using black-box testing techniques? The supporting reasons for white-box testing are given below:

- In fact, white-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software. Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
- Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.
- Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
- Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

4.2 LOGIC COVERAGE CRITERIA: Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered. Therefore the intention in white-box testing is to cover the whole logic. Discussed below are the basic forms of logic coverage.

Statement Coverage: The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified.

Decision or Branch Coverage: Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once. as:

Condition coverage: states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement:

while ((I < 5) && (J < COUNT))

In this loop statement, two conditions are there. So test cases should be de- signed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

Test case 1: I <5, J < COUNT

Test case 2: I < 5, J > COUNT

Decision/condition Coverage: Condition coverage in a decision does not mean that the decision has been covered. If the decision

if (A && B)

is being tested, the condition coverage would allow one to write two test cases:

Test case 1: A is True, B is False.

Test case 2: A is False, B is True.

But these test cases would not cause the THEN clause of the IF to execute (i.e. execution of decision). The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once

Multiple condition coverage: In case of multiple conditions, even decision/ condition coverage fails to exercise all outcomes of all conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated. Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.

4.3 BASIS PATH TESTING: Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program. Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing.

The guidelines for effectiveness of path testing are discussed below:

1. Path testing is based on control structure of the program for which flow graph is prepared.
2. Path testing requires complete knowledge of the program's structure.
3. Path testing is closer to the developer and used by him to test his module.
4. The effectiveness of path testing gets reduced with the increase in size of software under test
5. Choose enough paths in a program such that maximum logic coverage is achieved.

4.3.1 CONTROL FLOW GRAPH: The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a graph.

- **Node:** It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
- **Edges or links:** they represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
- **Decision node:** A node with more than one arrow leaving it is called a decision node.
- **Junction node:** A node with more than one arrow entering it is called a junction.
- **Regions:** Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

PATH TESTING TERMINOLOGY

- **Path:** A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or

decisions, one or more times.

- **Segment:** Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction- process-decision, decision-process-junction, and decision-process-decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.

- **Path segment**: A path segment is a succession of consecutive links that belongs to some path.

- **Length of a path:** The length of a path is measured by number of nodes traversed. This method has some analytical and theoretical benefits. If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.

- **Independent path:** An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined.
- **Cyclomatic Complexity:** Based on the cyclomatic complexity, the following formulae are being summarized.

1. V(G) = e – n + 2p, where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.

2. V(G) = d + p, where d is the number of decision nodes in the graph.

3. V(G) = number of regions in the graph

Guidelines for Basis Path Testing: We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then test cases can be designed according to this basis set.

The following steps should be followed for designing test cases using path testing:

- Draw the flow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.
- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

Example:

Consider the following program segment:

```
main()
{
    int number, index;
1.  printf("Enter a number");
2.  scanf("%d, &number);
3.  index = 2;
4.  while(index <= number – 1)5.
5.  {
6.      if (number % index == 0)
7.      {
8.          printf("Not a prime number");
9.          break;
10.     }
11.     index++;
12.     }
13.     if(index == number)
14.                 printf("Prime number");
15.             } //end main
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

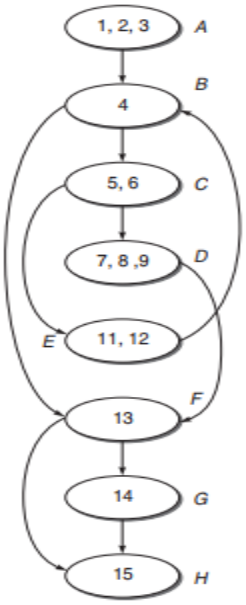
Solution

(a) DD graph

For a DD graph, the following actions must be done:

- Put the line numbers on the execution statements of the program, as shown in Fig. Start numbering the statements after declaring the variables, if no variables have been initialized. Otherwise, start from

the statement where a variable has been initialized.



- Put the sequential statements in one node. For example, statements 1, 2, and 3 have been put inside one node.
- Put the edges between the nodes according to their flow of execution.
- Put alphabetical numbering on each node like A, B, etc.

The DD graph of the program is shown in Figure 5.4.

(b) Cyclomatic complexity

(i) V(G) = e – n + 2 * p

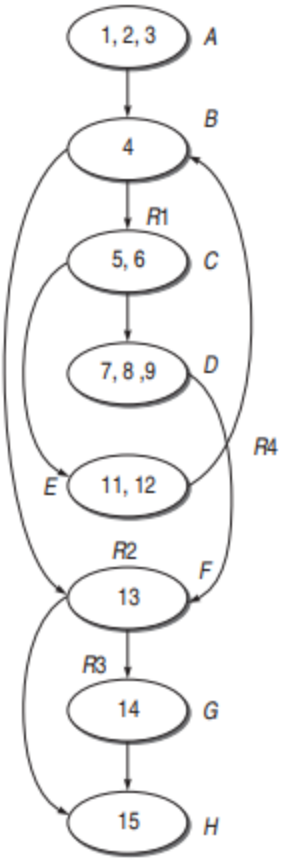
= 10 – 8 + 2

= 4

(ii) V(G) = Number of predicate nodes + 1

= 3 (Nodes B, C, and F) + 1

= 4



(ii)V(G) = Number of predicate nodes + 1

= 3 (Nodes B, C, and F) + 1

= 4

(iii) V (G) = Number of regions

= 4(R1, R 2, R 3, R 4)

(c) Independent paths: Since the cyclomatic complexity of the graph is 4, there will be 4 inde-pendent paths in the graph as shown below:

(i) A-B-F-H

(ii) A-B-F-G-H

(iii) A-B-C-E-B-F-G-H

(iv) A-B-C-D-F-H

(d) Test case design from the list of independent paths

Test case ID	Input num	Expected result	Independent paths coveredby test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

APPLICATIONS OF PATH TESTING: Path testing has been found better suitable as compared to other testing methods. Some of its applications are discussed below.

- Thorough testing / more coverage: Path testing provides us the best code coverage, leading to a thorough testing. Path coverage is considered better as compared to statement or branch coverage methods because the basis path set provides us the number of test cases to be covered which ascertains the number of test cases that must be executed for full coverage.
- Unit testing: Path testing is mainly used for structural testing of a module. In unit testing, there are chances of errors due to interaction of decision out- comes or control flow problems which are hidden with branch testing. Since each decision outcome is tested independently, path testing uncovers these errors in module testing and prepares them for integration.

- Integration testing: Since modules in a program may call other modules or be called by some other module, there may be chances of interface errors during calling of the modules. Path testing analyses all the paths on the interface and explores all the errors.

- Maintenance testing: Path testing is also necessary with the modified version of the software. If you have earlier prepared a unit test suite, it should be run on the modified software or a selected path testing can be done as a part of regression testing. In any case, path testing is still able to detect any security threats on the interface with the called modules.

- Testing effort is proportional to complexity of the software: Cyclomatic complexity number in basis path testing provides the number of tests to be executed on the software based on the complexity of the software. It means the number of tests derived in this way is directly proportional to the complexity of the software. Thus, path testing takes care of the complexity of the software and then derives the number of tests to be carried out.

- Basis path testing effort is concentrated on error-prone software: Since basis path

testing provides us the number of tests to be executed as a measure of software cyclomatic complexity, the cyclomatic number signifies that the testing effort is only on the error-prone part of the software, thus minimizing the testing effort.