Module 1: Testing Methodology

-by
Rohini M. Sawant

INTRODUCTION

Software testing has always been considered a single phase performed after coding.

But time has proved that our failures in software projects are mainly due to the fact that we have not realized the role of software testing as a process.

Software is becoming complex, but the demand for quality in software products has increased.

This rise in customer awareness for quality increases the workload and responsibility of the software development team.

That is why software testing has gained so much popularity

SOFTWARE TESTING DEFINITIONS

- Testing is the process of executing a program with the intent of finding errors-Myers.
- A successful test is one that uncovers an as-yet-undiscovered error.- Myers
- Testing can show the presence of bugs but never their absence- W. Dijkstra
- Program testing is a rapidly maturing area within software engineering that is receiving increasing notice both by computer science theoreticians and practitioners. Its general aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances- E. Miller

Myths about Software Testing

- Testing is a single phase in SDLC .
- Testing is easy.
- Software development is worth more than testing.
- Complete testing is possible. (Hence we go for Effective testing)
- Testing starts after program development.
- The purpose of testing is to check the functionality of the software.

GOALS OF SOFTWARE TESTING

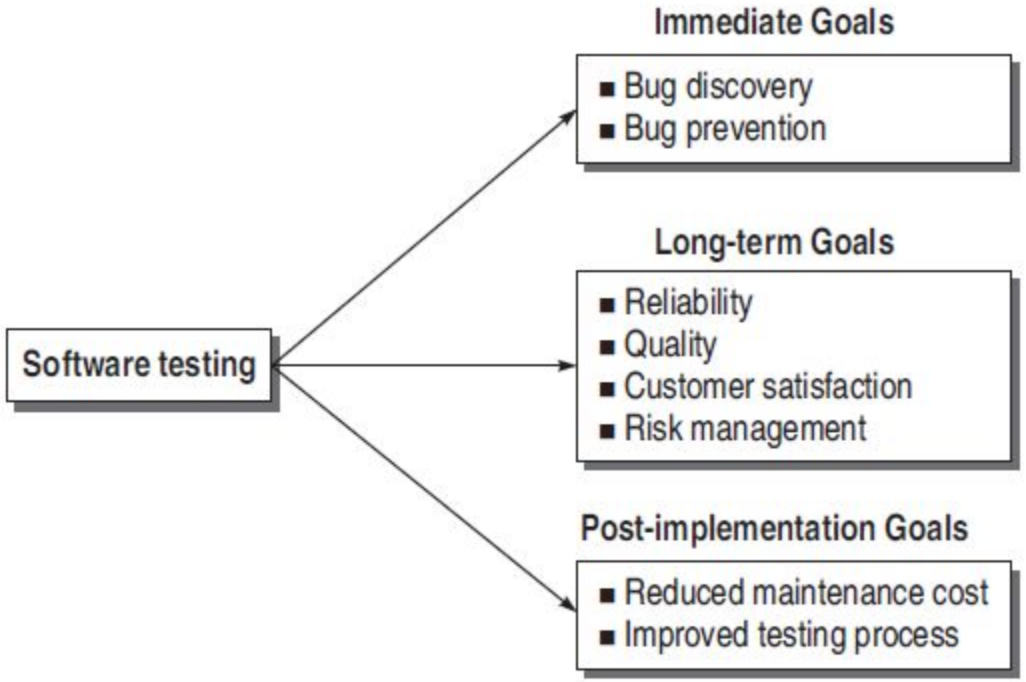


Figure 1.2 Software testing goals

GOALS OF SOFTWARE TESTING

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 behaviour 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.Though errors cannot be prevented to zero, they can be minimized. In this sense, bug prevention is a superior goal of testing.

GOALS OF SOFTWARE TESTING

Long-term goals: These goals affect the product quality in the long run, Some of them are discussed here:

- **Reliability**
- **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.

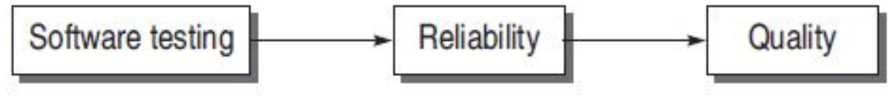


Figure 1.3 Testing produces reliability and quality

- **Customer satisfaction: Software Testing > Reliability > Quality > Customer Satisfaction**

  A complete testing process achieves reliability, reliability enhances the quality, and quality in turn, increases the customer satisfaction.

- **Risk management:** The purpose of software testing as a control is to provide information to management so that they can better react to risk situations. Risk Factors like Cost, Time, Resources. 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.

GOALS OF SOFTWARE TESTING

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.
- **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 analysed to find out snags in the present testing process, which can be rectified in future projects.

MODEL FOR SOFTWARE TESTING

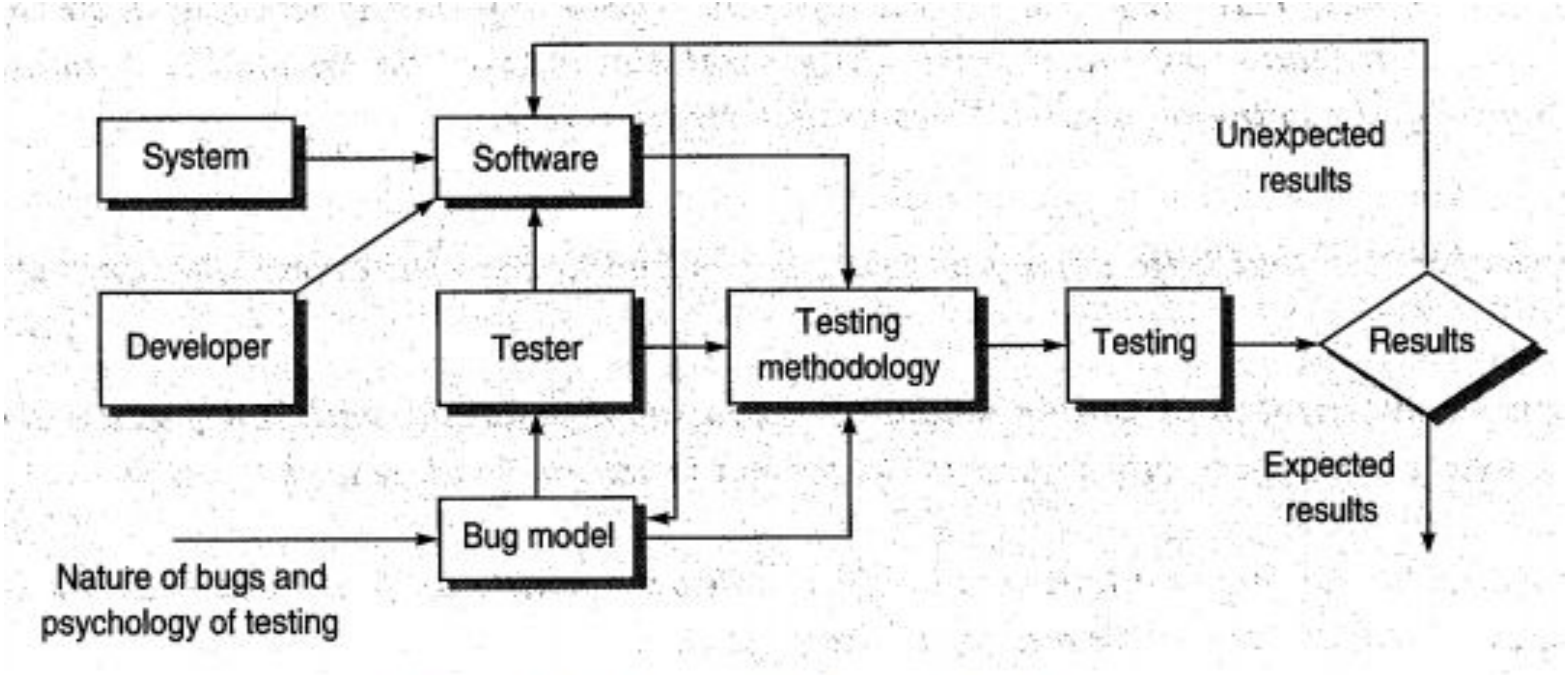


Figure 1: Software Testing Models

MODEL FOR SOFTWARE TESTING

- The software is basically a part of a system for which it is being developed.
- Systems consist of hardware and software to make the product run.
- Testability is a major issue for the developer while developing the software, as a badly written software may be difficult to test. 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

MODEL FOR SOFTWARE TESTING

- **Software and Software Model:** 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:** It 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.

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

Effective ST vs. Exhaustive ST:

- **Exhaustive/Complete Testing** which means Every statement in the program and every possible path combination of data must be executed.
- Exhaustive software testing refers to a theoretical approach where every possible input, combination, and state of a software application is tested.
- This includes testing different input values, sequences of operations, and various configurations to ensure that the software behaves correctly under all expected conditions.
- Since this is not possible, we should concentrate on **Effective Testing** which uses efficient techniques to test the software in such a way that all the important features are tested and endorsed.

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

EXHAUSTIVE 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.
- This combination of possible tests is infinite in the sense that the processing resources and time are not sufficient for performing these tests.
- Computer speed and time constraints limit the possibility of performing all the tests.
- Complete testing requires the organization to invest a long time which is not cost-effective.
- However, due to the complexity and size of modern software systems, exhaustive testing is rarely feasible or cost-effective. Instead, software testing typically focuses on strategically selecting representative test cases that are likely to uncover the most significant defects or vulnerabilities.
- Therefore, testing must be performed on selected subsets that can be performed within the constrained resources.

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

Reasons 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 domain of input data has four sub-parts:

(a) valid inputs,

(b) invalid inputs,

(c) edited inputs, and

(d) race condition inputs

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

a) 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?

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

b) 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 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: (i) Numbers out of range (ii) Combination of alphabets and digits (iii) Combination of all alphabets (iv) Combination of control characters (v) Combination of any other key on the keyboard

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

c) 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.
- The behaviour of users cannot be judged.
- They can behave in a number of ways, causing defect in testing a program. That is why edited inputs are also not tested completely

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

d) 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

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

There are too Many Possible Paths Through the Program to Test

- A program path can be traced through the code from the start of a program to its termination. Two paths differ if the program executes different statements in each, or executes the same statements but in different order.
- If there are two paths in one iteration. Now the total number of paths will be 2n + 1, where n is the number of times the loop will be carried out.
- Therefore, all these paths cannot be tested, as it may take years.
- The complete path testing, if performed somehow, does not guarantee that there will not be errors.

Every Design Error Cannot be Found

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

EFFECTIVE TESTING

- This selected group of subsets, but not the whole domain of testing, makes effective software testing.
- Effective testing can be enhanced if subsets are selected based on the factors which are required in a particular environment.
- Effective testing provides the flexibility to select only the subsets of the domain of testing based on project priority such that the chances of failure in a particular environment is minimized.

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

EFFECTIVE TESTING IS HARD

- Effective testing, though not impossible, is hard to implement.
- But if there is careful planning, keeping in view all the factors which can affect it, then it is implementable as well as effective.
- To achieve that planning, we must understand the factors which make effective testing difficult.
- At the same time, these factors must be resolved.
- These are described as follows.

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

a) Defects are hard to find

- The major factor in implementing effective software testing is that many defects go undetected due to many reasons, e.g. certain test conditions are never tested.
- Secondly, developers become so familiar with their developed system that they overlook details and leave some parts untested.
- So a proper planning for testing all the conditions should be done and independent testing, other than that done by developers, should be encouraged

EFFECTIVE SOFTWARE TESTING VS. EXHAUSTIVE SOFTWARE TESTING

b)When are we done with testing

- This factor actually searches for the definition of effective software testing. Since exhaustive testing is not possible, we don’t know what should be the criteria to stop the testing process.
- A software engineer needs more rigorous criteria for determining when sufficient testing has been performed.
- Moreover, effective testing has the limiting factor of cost, time, and personnel.
- In a nutshell, the criteria should be developed for enough.
- For example, features can be prioritized which must be tested within the boundary of cost, time, and personnel of the project.

Effective Testing vs Exhaustive Testing

Aspect	Effective Testing	Exhaustive Testing
Objective	Identify critical defects	Uncover all potential defects
Scope	Focus on high-priority areas	Cover all possible scenarios
Time and Resources	Less time-consuming	Highly time-consuming and resource-intensive
Feasibility	Practical for most applications	Impractical for large systems
Use Cases	Suitable for most applications	Ideal for critical systems
Reliability	Good reliability	Maximum reliability
Maintenance	Easier to maintain	High maintenance overhead
Cost	Lower Cost	Higher Cost

SOFTWARE TESTING TERMINOLOGY

- **Failure-** When the software is tested, failure is the first term being used. It means the inability of a system or component to perform a required function according to its specification. 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. It is the reason embedded in any phase of SDLC and results in failures. Fault is synonymous with the words defect or bug.
- **Error** Whenever a development team member makes a mistake in any phase of SDLC, errors are produced. It might be a typographical error, a misleading of a specification, a misunderstanding of what a subroutine does, and so on. Error is a very general term used for human mistakes. Thus, an error causes a bug/fault and the bug in turn causes failures

SOFTWARE TESTING TERMINOLOGY

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 a program behaviour. The primary purpose of designing a test case is to find errors in the system. For designing the test case, it needs to provide a set of inputs and its corresponding expected outputs.

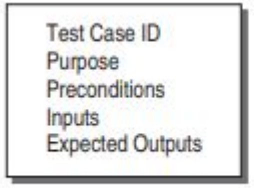

Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs

Figure 2.3 Test case template

SOFTWARE TESTING TERMINOLOGY

- **Testware** The documents created during testing activities are known as testware. These are the documents that a test engineer produces. It may include test plans, test specifications, test case design, test reports, etc. Testware documents should also be managed and updated like a software product.
- **Incident** An incident is the symptom(s) associated with a failure that alerts the user about the occurrence of a failure.
- **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.

SOFTWARE TESTING LIFE CYCLE (STLC)

- The testing process divided into a well-defined sequence of steps is termed as software testing life cycle (STLC).
- The major contribution of STLC is to involve the testers at early stages of development.
- This has a significant benefit in the project schedule and cost. The STLC also helps the management in measuring specific milestones.
- The Software Testing Life Cycle (STLC) outlines the various phases and activities involved in testing software applications.
- It provides a structured approach to ensure that software testing is thorough, systematic, and effective.

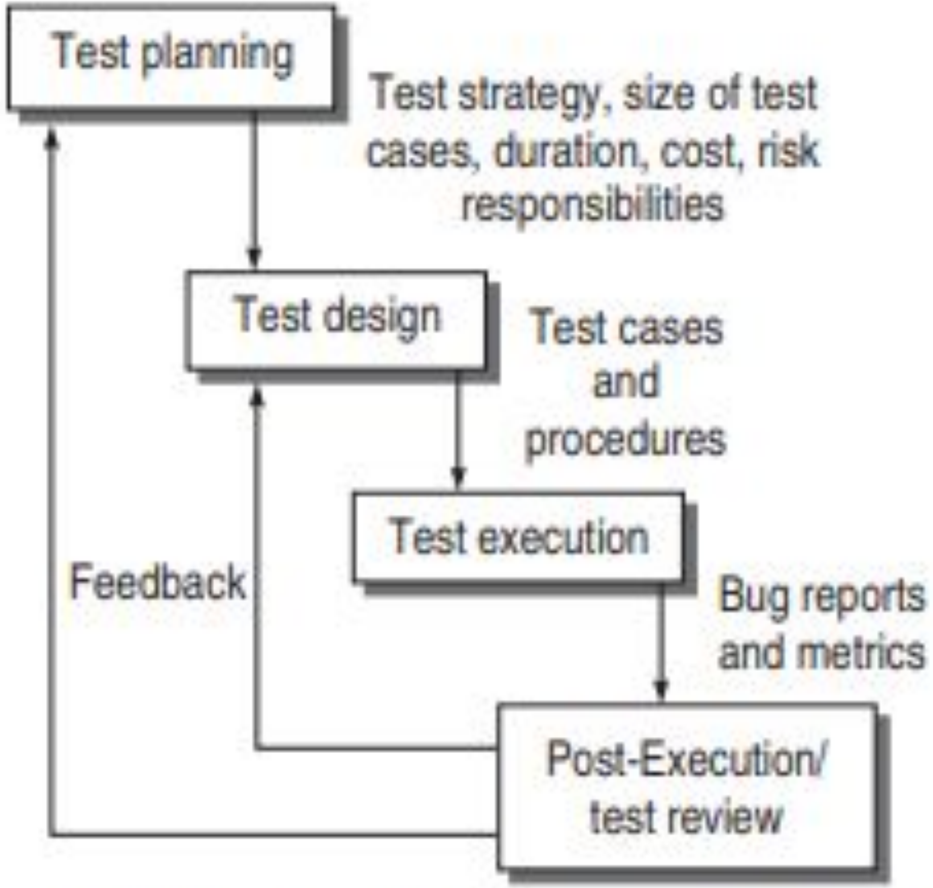

Test planning
Test strategy, size of test cases, duration, cost, risk responsibilities
Test design
Test cases and procedures
Test execution
Bug reports and metrics
Post-Execution/ test review
Feedback

Figure 2.8 Software testing life cycle

SOFTWARE TESTING LIFE CYCLE (STLC)

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.
- Identifying reporting procedures, bug classification, databases for testing, bug severity levels, and project metrics.

The major output of test planning is the test plan documents.

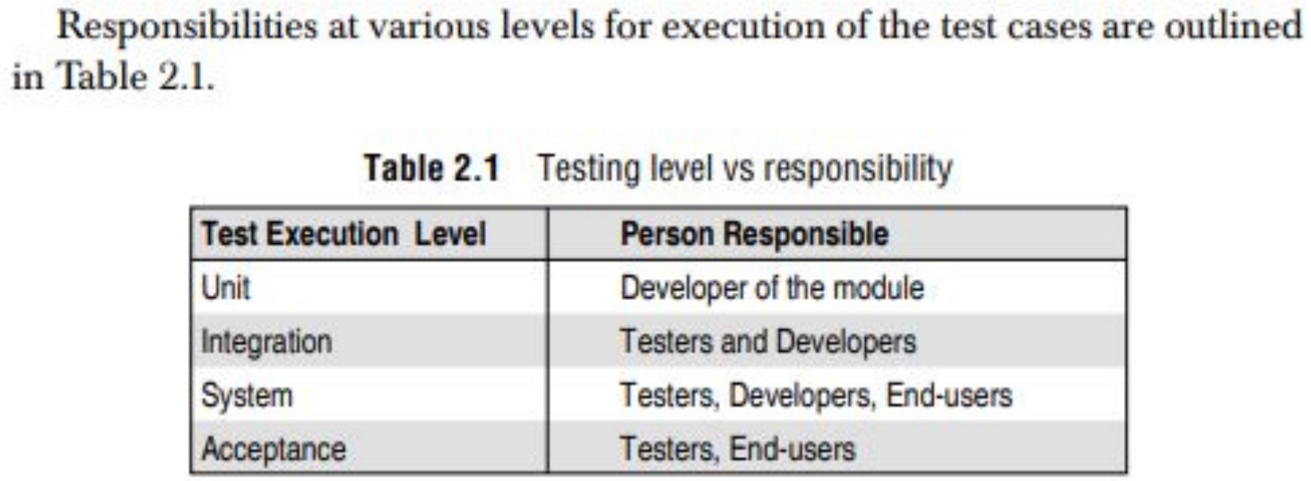
SOFTWARE TESTING LIFE CYCLE (STLC)

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. The test design is an important phase after test planning. It includes the following critical activities:

- Determining the test objectives and their prioritization
- Preparing list of items to be tested
- Mapping items to test cases
- Selection of test case design techniques
- Creating test cases and test data
- Setting up the test environment and supporting tools
- Creating test procedure specification

SOFTWARE TESTING LIFE CYCLE (STLC)

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. It is the decision of the test team to opt for automation or manual execution. Test results are documented in the test incident reports, test logs, testing status, and test summary.



Responsibilities at various levels for execution of the test cases are outlined in Table 2.1.

Table 2.1 Testing level vs responsibility

Test Execution Level	Person Responsible
Unit	Developer of the module
Integration	Testers and Developers
System	Testers, Developers, End-users
Acceptance	Testers, End-users

SOFTWARE TESTING LIFE CYCLE (STLC)

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 feedback so that maximum number of bugs can be removed. This is the primary goal of all test activities done earlier. As soon as the developer gets the bug report, he performs the following activities:

- Understanding the bug
- Reproducing the bug
- Analysing the nature and cause of the bug

LIFE CYCLE OF A BUG

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 Fig. 2.11.

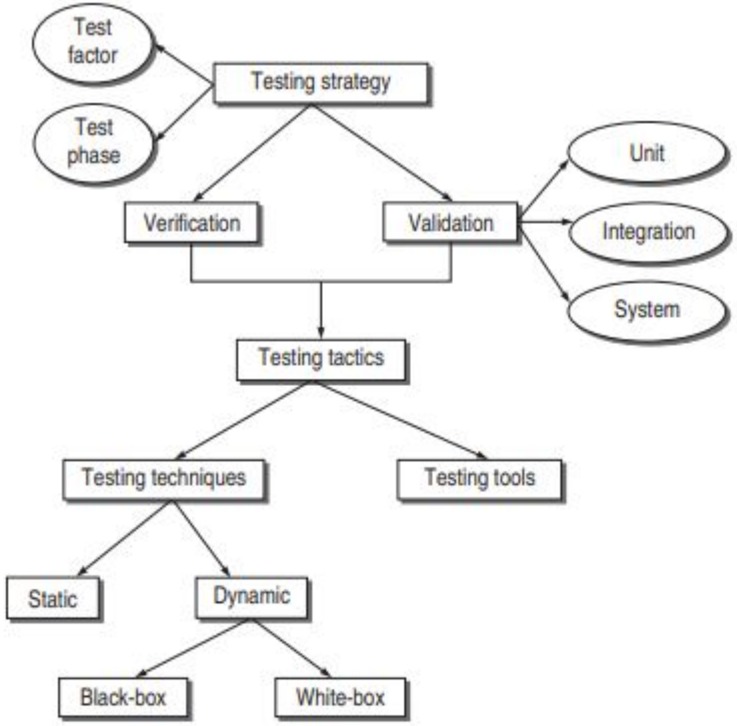


Figure 2.11 Testing methodology

SOFTWARE 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

TEST STRATEGY MATRIX

Test Strategy Matrix: It identifies the concerns that will become the focus of test planning and execution. In this way, this matrix becomes an input to develop the testing strategy. The matrix is prepared using test factors and test phase. The steps to prepare this matrix are discussed below.

- Select and rank test factors: Based on the test factors list, the most appropriate factors according to specific systems are selected and ranked from the most significant to the least. These are the rows of the matrix.
- Identify system development phases: Different phases according to the adopted development model are listed as columns of the matrix. These are called test phases.
- Identify risks associated with the system under development: In the horizontal column under each of the test phases, the test concern with the strategy used to address this concern is entered. The purpose is to identify the concerns that need to be addressed under a test phase.The risks may include any events, actions, or circumstances that may prevent the test program from being implemented or executed according to a schedule, such as late budget approvals, delayed arrival of test equipment, or late availability of the software application

Table 2.2 Test strategy matrix

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test
Portability	Is portability feature mentioned in specifications according to different hardware?					Is system testing performed on MIPS and INTEL platforms?
Service Level	Is time frame for booting mentioned?	Is time frame incorporated in design of the module?				

DEVELOPMENT OF TEST STRATEGY

- When the project under consideration starts and progresses, testing too starts from the first step of SDLC.
- Therefore, the test strategy should be such that the testing process continues till the implementation of project.
- Moreover, the rule for development of a test strategy is that testing ‘begins from the smallest unit and progresses to enlarge’.
- This means the testing strategy should start at the component level and finish at the integration of the entire system.
- Thus, a test strategy includes testing the components being built for the system, and slowly shifts towards testing the whole system.
- This gives rise to two basic terms—Verification and Validation—the basis for any type of testing. It can also be said that the testing process is a combination of verification and validation
  - **Verification is ‘Are we building the product right?’**
  - **Validation is ‘Are we building the right product?’**

V-Testing Life Cycle Model

- In V-testing concept, as the development team attempts to implement the software, the testing team concurrently starts checking the software.
- When the project starts, both the system development and the system test process begin.
- The team that is developing the system begins the system development process and the team that is conducting the system test begins planning the system test process, as shown in Fig.
- Both teams start at the same point using the same information. If there are risks, the tester develops a process to minimize or eliminate them.

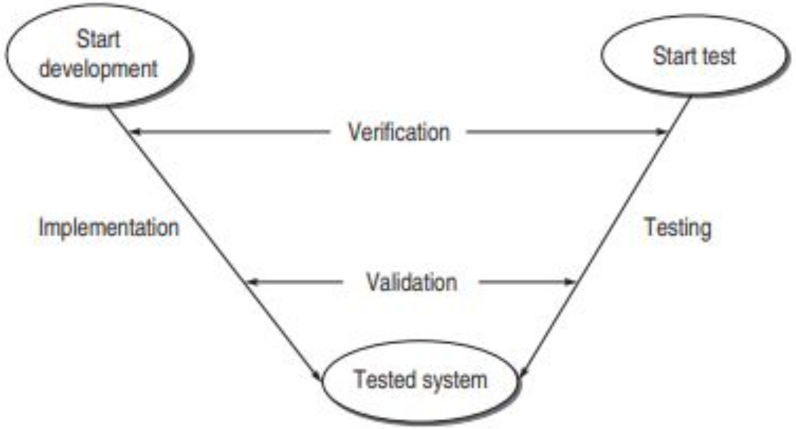


Figure 2.12 V-testing model

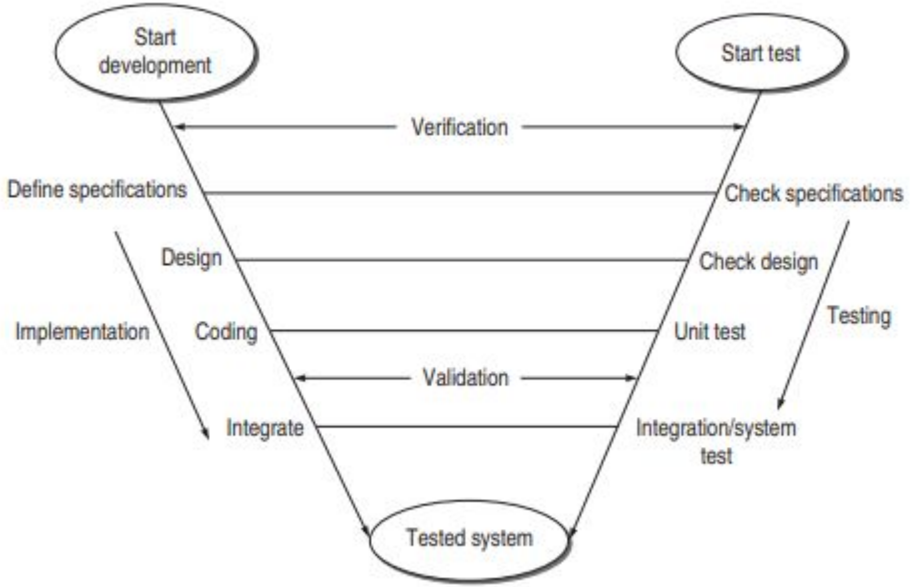


Figure 2.13 Expanded V-testing model

VALIDATION ACTIVITIES

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

- Unit Testing: It is a major validation effort performed on the smallest module of the system. If avoided, many bugs become latent bugs and are released to the customer. Unit testing is a basic level of testing which cannot be overlooked, and confirms the behaviour 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. When we unit test a module, its interfacing with other modules remain untested. When one module is combined with another in an integrated environment, interfacing between units must be tested.

VALIDATION ACTIVITIES

- 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

TESTING TACTICS

Software Testing Techniques:

- At this stage, testing can be defined as the design of effective test cases such that most of the testing domains will be covered detecting the maximum number of bugs.
- Therefore, the next objective is to find the technique which will meet both the objectives of effective test case design, i.e. coverage of testing domain and detection of maximum number of bugs. The technique used to design effective test case is called Software Testing Technique.
- These techniques can be categorized into two parts: (a) static testing and (b) dynamic testing.

TESTING TACTICS

- 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:
-

TESTING TACTICS

- 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 TACTICS

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 performed. The tester should fi rst understand the testing techniques and then go for the tools that can be used with each of the techniques.

TACTICAL TEST PLAN

A tactical test plan is required to start testing. This test plan provides:

- Environment and pre-test background
- Overall objectives
- Test team
- Schedule and budget showing detailed dates for the testing
- Resource requirements including equipment, software, and personnel
- Testing materials including system documentation, software to be tested, test inputs, test documentation, test tools
- Specified business functional requirements and structural functions to be tested
- Type of testing technique to be adopted in a particular SDLC phase or what are the specifi c tests to be conducted in a particular phase

VERIFICATION & VALIDATION

A V-diagram provides the following insights about software testing:

- Testing can be implemented in the same flow as for the SDLC.
- Testing can be broadly planned in two activities, namely verification and validation.
- Testing must be performed at every step of SDLC.
- V-diagram supports the concept of early testing.
- V-diagram supports parallelism in the activities of developers and testers.
- The more you concentrate on the V&V process, more will be the cost effectiveness of the software.
- Testers should be involved in the development process.

VERIFICATION & VALIDATION ACTIVITIES

V&V activities can be understood in the form of a diagram which is described here. To understand this diagram, we first need to understand SDLC phases. After this, verification and validation activities in those SDLC phases will be described.

- 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: 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).

VERIFICATION & VALIDATION ACTIVITIES

- Functional design or high-level design: Functional design is the process of translating user requirements into a set of external interfaces. The high-level design is prepared with SRS and software analysts convert the requirements into a usable product.
- An HLD document will contain the following items at a macro level:

1. Overall architecture diagrams along with technology details

2. Functionalities of the overall system with the set of external interfaces

3. List of modules

4. Brief functionality of each module

5. Interface relationship among modules including dependencies between modules, database tables identified along with key elements

VERIFICATION & VALIDATION ACTIVITIES

- Internal design or low-level design: 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.
- 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. 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.

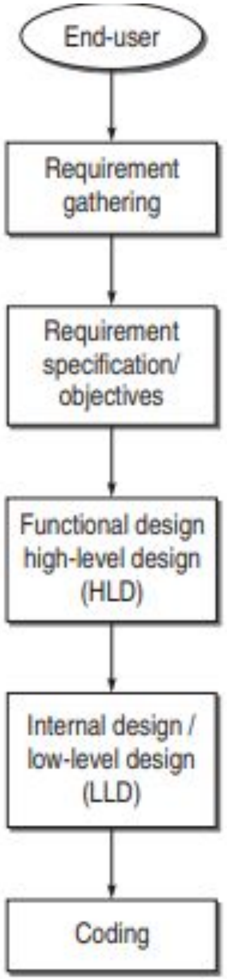
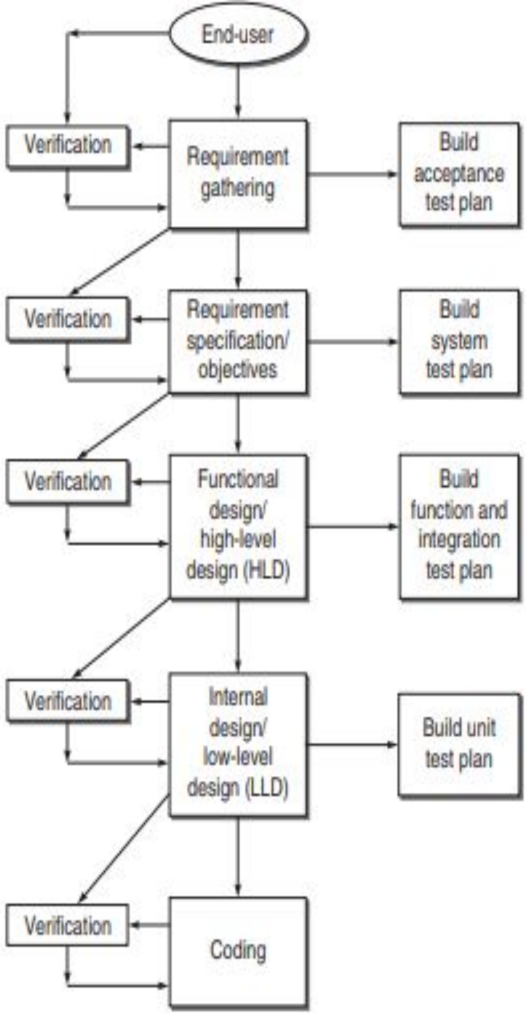

End-user
Requirement gathering
Requirement specification/ objectives
Functional design high-level design (HLD)
Internal design / low-level design (LLD)
Coding

Figure 3.2 SDLC Phases


End-user
Verification
Requirement gathering
Build acceptance test plan
Verification
Requirement specification/ objectives
Build system test plan
Verification
Functional design/ high-level design (HLD)
Build function and integration test plan
Verification
Internal design/ low-level design (LLD)
Build unit test plan
Verification
Coding

igure 3.3 Tester performs verification and prepares test plan during every SDLC phase

- When the coding is over for a unit or a system, and parallel verification activities have been performed, then the system can be validated.
- It means validation activities can be performed now.
- These are executed with the help of test plans prepared by the testers at every phase of SDLC.
- This makes the complete V&V activities diagram

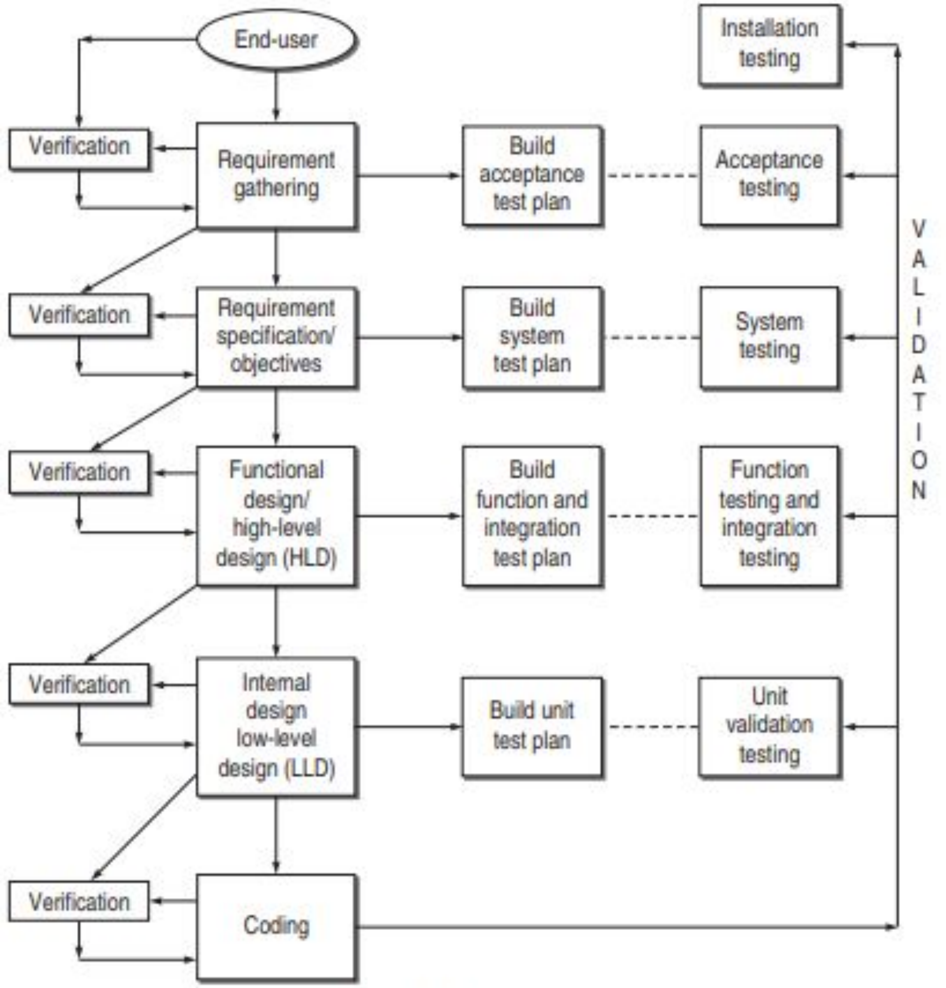
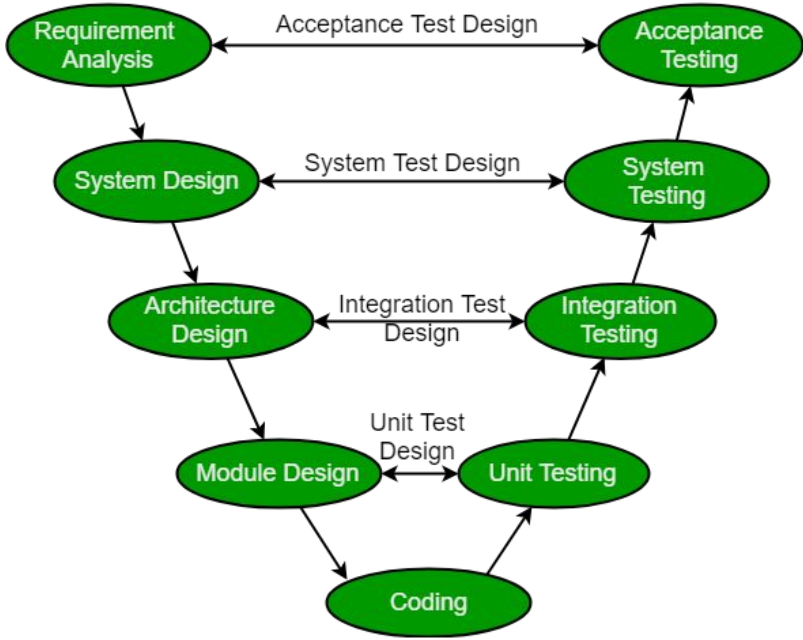

End-user
Verification
Requirement gathering
Build acceptance test plan
Acceptance testing
Installation testing
Verification
Requirement specification/ objectives
Build system test plan
System testing
Verification
Functional design/ high-level design (HLD)
Build function and integration test plan
Function testing and integration testing
Verification
Internal design low-level design (LLD)
Build unit test plan
Unit validation testing
Verification
Coding
VALIDATION

Figure 3.4 V&V diagram

The V-Model

- The V-model is a type of SDLC model where process executes in a sequential manner in V-shape.
- It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage.
- Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.
- **Verification:** It involves static analysis technique (review) done without executing code. It is the process of evaluation of the product development phase to find whether specified requirements meet.
- **Validation:** It involves dynamic analysis technique (functional, non-functional), testing done by executing code. Validation is the process to evaluate the software after the completion of the development phase to determine whether software meets the customer expectations and requirements.



The V-Model

Design Phase:

- **Requirement Analysis:** This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.
- **System Design:** This phase contains the system design and the complete hardware and communication setup for developing product.
- **Architectural Design:** System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.
- **Module Design:** In this phase the system breaks down into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD)

Testing Phases:

- **Unit Testing:** Unit Test Plans are developed during module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.
- **Integration testing:** After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.
- **System Testing:** System testing test the complete application with its functionality, inter dependency, and communication.It tests the functional and non-functional requirements of the developed application.
- **User Acceptance Testing (UAT):** UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user's requirement and system is ready for use in real world.

VERIFICATION & VALIDATION ACTIVITIES

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

- **Verification of Requirements and Objectives**
- **Verification of High-Level Design**
- **Verification of Low-Level Design**
- **Verification of Coding (Unit Verification)**

VERIFICATION OF REQUIREMENTS

- In this type of verification, all the requirements gathered from the user’s viewpoint are verified.
- For this purpose, an acceptance criterion is prepared.
- An acceptance criterion defines the goals and requirements of the proposed system and acceptance limits for each of the goals and requirements.

The tester works in parallel by performing the following two tasks:

1. 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.

2. 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 of this verification is to ensure that the user’s needs are properly understood 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

HOW TO VERIFY REQUIREMENTS AND OBJECTIVES?

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:

(a) Testers should refer to other documentations or applicable standards and compare the specified requirement with them.

(b) Testers can interact with customers or users, if requirements are not well-understood.
(c) Testers should check the correctness in the sense of realistic requirement. If the tester feels that a requirement cannot be realized using existing hardware and software technology, it means that it is unrealistic. In that case, the requirement should either be updated or removed from SRS.

HOW TO VERIFY REQUIREMENTS AND OBJECTIVES?

Unambiguous: A requirement should be verified such that it does not provide too many meanings or interpretations. It should not create redundancy in specifications. The following must be verified:

(a) Every requirement has only one interpretation.

(b) 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: (a) Real-world objects conflict

(b)Logical conflict between two specified actions

(c) Conflicts in terminology should also be verified.

HOW TO VERIFY REQUIREMENTS AND OBJECTIVES?

Completeness: The requirements specified in the SRS must be verified for completeness. We must

(a) Verify that all significant requirements such as functionality, performance, design constraints, attribute, or external interfaces are complete.

(b) Check whether responses of every possible input (valid & invalid) to the software have been defined.

(c) Check whether figures and tables have been labeled and referenced completely.

HOW TO VERIFY REQUIREMENTS AND OBJECTIVES?

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.

(a) If the specification is a new one, then all the above mentioned steps and their feasibility should be verified.

(b) 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.

VERIFICATION OF HIGH-LEVEL DESIGN

- The architecture and design is documented in another document called the software design document (SDD).
- Like the verification of requirements, the tester is responsible for two parallel activities in this phase as well:

1. The tester verifies the high-level design. Since the system has been decomposed in a number of subsystems or components, the tester should verify the functionality of these components. The tester verifies that all the components and their interfaces are in tune with requirements of the user. Every requirement in SRS should map the design.

2. The tester also prepares a Function Test Plan and Integration Test Plan which is based on the SRS.

Verification of High Level Design happens at Data, Architecture and Interface Level.

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 subsystems 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.

Verification of User-Interface Design

- Check all the interfaces between modules according to the architecture design.
- 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.
- For a Help Facility, verify the following: (i) The representation of Help in its desired manner (ii) The user returns to the normal interaction from Help
- For error messages and warnings, verify the following: (i) Whether the message clarifies the problem (ii) Whether the message provides constructive advice for recovering from the error `

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 design 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 is prepared in their separate SRS and SDD.
- Testers also perform the following parallel activities in this phase:
- 1. The tester verifies the LLD. The details and logic of each module is verified such that the high-level and low-level abstractions are consistent.
- 2. The tester also prepares the Unit Test Plan.

VERIFICATION OF 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 design, each requirement mentioned in the SRS is verified.

HOW TO VERIFY 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.
- 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.

 Code verification can be done most efficiently by the developer, as he has prepared the code. He can verify every statement, control structure, loop, and logic such that every possible method of execution is tested.

- Two kinds of techniques are used to verify the coding: (a) static testing, and (b) dynamic testing.

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.

The reasons for validation 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 desired behaviour, as described in the functional design specification.

 It is not always certain that all the stages till coding are bug-free. The bugs that 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:

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.

 Testers must understand the current SDLC phase.

 Testers must study the relevant documents in the corresponding SDLC phase.

 On the basis of the understanding of SDLC phase and related documents, testers must prepare the related test plans which are used at the time of validation testing. Under test plans, they must prepare a sequence of test cases for validation testing.

VALIDATION ACTIVITIES

The following test plans have been recognized which the testers have already prepared with the incremental progress of SDLC phases:

- 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.

VALIDATION ACTIVITIES

- 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 ACTIVITIES

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. Thus, unit testing is a process of testing the individual components of a system. A unit or module must be validated before integrating it with other modules. Unit validation is the first validation activity after the coding of one module is over.

VALIDATION ACTIVITIES

- 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. The objective of function test is to measure the quality of the functional components of the system.

VALIDATION ACTIVITIES

- 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. 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: As explained earlier, 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 the acceptance criteria contracted with the customer. This is called acceptance testing.
- 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. Installation testing tests the interface to operating software, related software, and any operating procedures.