Unit 1

Introduction:

Software is becoming complex, but the demand for quality in software products has been increased. This rise in customer awareness for quality increases the workload and responsibility of thesoftware development team. That is why software testing has gained so much popularity in the last decade. Jobs trends have shifted from development to software testing. Today software quality assurance and software testing courses are offered by many institutions. Organizations have separate testing groups with proper hierarchy. Software development is driven with testing outputs.

Evolution:

The different phases of software testing are:

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. Debugging was a more general term at that time. Till 1956, there was no clear distinction between software development, testing and debugging.

Demonstration – Oriented Phase (1957-1978):
In 1957, Charles Baker pointed out that the purpose of checkout is not only to run the software but also to determine 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 ofcheckout was to show the absence of errors.

Destruction – Oriented Phase (1979-1982):

In this phase testing process is changed from 'testing is to show the absence of errors' to 'testing is to find more and more errors'. This phase separated debugging from testing and stressed on the valuable test cases if they explore more bugs.

Evolution – Oriented Phase (1979-1982):
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.

Prevention – Oriented Phase (1979-1982):
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 we can prevent the in implementation or further phases. The prevention model includes test planning, test analysis, and test design activities playing a major role, while the evaluation phase mainly relies on analysis and reviewing techniques other than testing.

Process – Oriented Phase (1979-1982):
In this phase testing was established as a complete process rather than a single phase (performed after coding) in the SDLC. The testing process starts as soon as the requirements for a project are specified and its runs in 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).

The evolution of software testing was also discussed by ?Hung Q Nguyen and Rob Pirozzi in three phases namely Software Testing 1.0, Software Testing 2.0 and Software Testing 3.0.

Software Testing 1.0:
In this phase, Software Testing was just considered a single phase to be performed after coding of the software in SDLC. No test organization was there. A few testing tools were present but their use was limited due ti high cost. Management was not concerned with testing, as there was no quality goal.

Software Testing 2.0:
In this phase, Software Testing gained importance in SDLC and the concept of early testing also started. Testing was evolving in the direction of planning the test resources. Many testing tools were also available in this phase.

Software Testing 3.0:
In this phase, Software Testing is being evolved in the form of a process which is based on strategic effort. It means that there should be a process which gives us a roadmap of the overall testing process. Moreover, it should be driven by quality goals so that all controlling and monitoring activities and be performed by the mangers. Thus the management is actively involved in the testing phase.

Myths & Facts:

Myth: –Testing is a single phase in SDLC
Truth: It is a myth 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 continuous even after implementation of the software.

Myth: –Testing is easy
Truth: The general perception is that, software testing is an easy job, where in test cases are executedwith 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 through understanding of the project being developed with its overall design.

Myth: –Software development is worth more than testing
Truth: This myth prevails in the minds of every team member and even in freshers who are seeking job. 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. But testing has now become as established path for job-seekers. Testing is a complete process like development, so the testing tam enjoys equal status and importance as the development team.

Myth: –Complete testing is possible
Truth: 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. This is the reason why the term 'Complete Testing' has been replaced with 'Effective Testing'.

Myth: –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 the work of the tester begins as soon as we get the specifications. The tester performs testing at the end of every phase in SDLC in the form of verification and validation.

Myth: –The purpose of the 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. There are various things related to quality of the software, for which test cases must be executed.

Myth: –Anyone can be tester
Truth: As an established process, software testing as a career also needs training for various purposes such as to understand 1) Various phases of SDLC, 2) Recent techniques to design test cases, 3) various tools and how to work on them.

Goals of Software Testing:

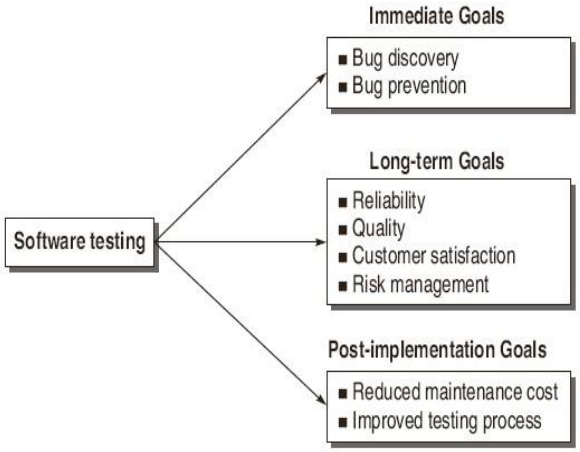


Figure 1.2 Software testing goals

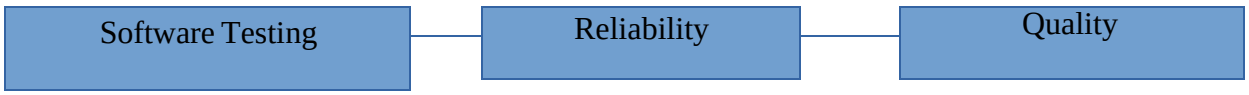
Short-Term or Immediate Goals: These goals are the immediate results after performing testing. These goals may be set in the individual phases of SDLC.

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 success rate of software testing.

Bug Prevention: 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 stages.

Long-Term Goals:

Quality:

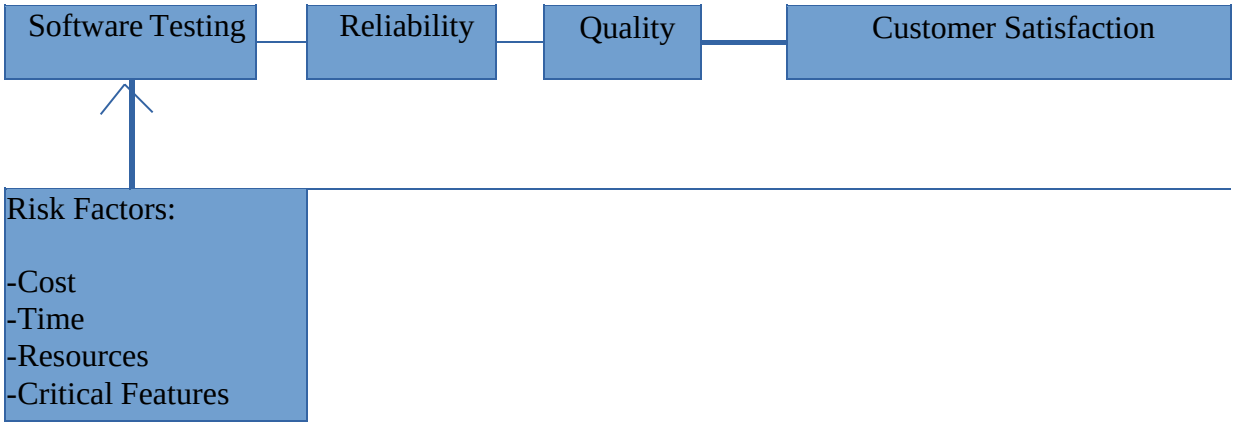


Since software is also a product, its quality is primary from the users point of view. Therefore, the first goal of understanding and performing the testing process is to enhance the quality of the software product. The software should be passed through a rigorous reliability analysis to attain high quality standards.

Customer Satisfaction:

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 the unspecified requirementswhich are otherwise understood.

Risk Management:



Risk is the probability that undesirable events will occur in a system. These undesirable events will prevent the organization from successfully implementing in business initiatives. It is the testers responsibility to evaluate business risks (such as cost, time, resources and critical features) and make the same a basis for testing choices. Testers should also categorize the levels of risk after their assessment (like low-level risk, high risk, moderate risk).

Post Implementation Goals:

Reduced Maintenance Cost: The maintenance cost of any software product is not 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.

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.

Psychology of 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 testing is to show that the Software performs its intended functions correctly. If testing is performed keeping this goal in mind, then we cannot achieve the desired goals as we will not able to test the software as a whole. This approach is based on the human psychology that human beings tend to work according to the goals fixed in their minds. If we have a pre-conceived assumption that the software is error free, then we will design the test cases to show that all the modules run smoothly.

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. It means now we don't think of testing only those features or specifications which may have mentioned in document like SRS, but we also think in terms of finding bugs in the features or domains which are understood but not specified. Thus Software testing may be designed as:

–Testing is the process of executing a program with the intent of finding errors"

Definition:

–Testing is the process of executing a program with the intent of finding errors"

–A Successful test is one that uncovers as as-yet-undiscovered error" --Myers

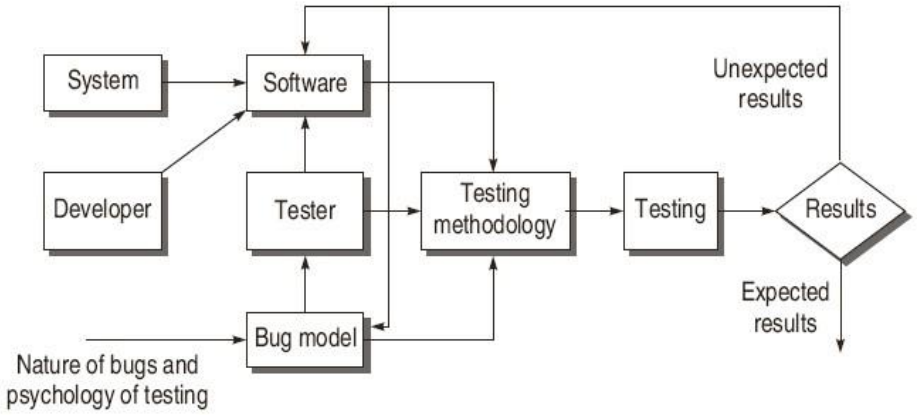
"Testing can show the presence of bugs but never their absence" *--W Dijkstra*

" Software Testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context which it is intended to operate" *--Cem Kaner*

–Software testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item" -- IEEE

Model for Software Testing:

The software is basically a part of a system for which it is being developed. The developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software, as badly written software may be difficult to test.



Testers are supposed to get on with their tasks as soon as the requirements are specified. With suitable testing techniques decided in the testing methodology, testing is performed on the software with a particular goal. The following describe the testing model:

Software and Software Model:

Software is built after analyzing the system in the environment. It is complex entity which deals with environment, logic, programmer psychology etc. Since in this model of testing, our aim is to concentrate on the testing on the testing process, therefore the software under consideration should not be complex such that it would not be tested. 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, tetsers can develop a testing methodology that incorporates both testing strategy and tactics. Testing staregy is the roadmap that gives us well-defined steps for the overall testing process. Once the planned steps of the testing process are prepared, software testing techniques and testing tools can be applied within these steps.

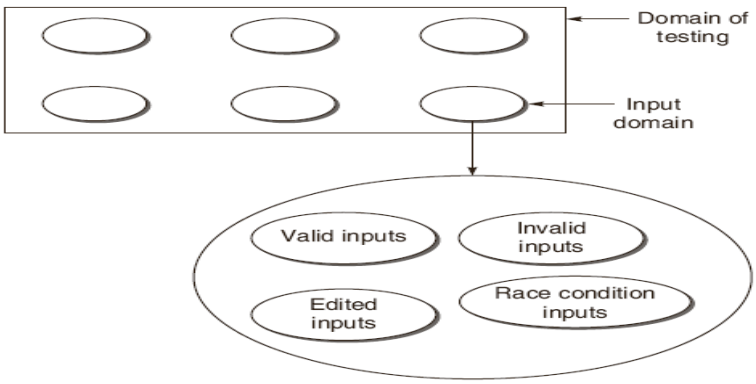
Effective Vs Exhaustive Software Testing:

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.

The testing process should be understood as a domain of possible tests. There are subsets of these possible tests. But the domain of possible tests becomes infinite, as we cannot test every possible combination.

Complete testing requires the organization to invest a long tine which is not cost-effective. Therefore, testing must be performed on selected subsets that can be performed within the constrained resources. This selected group of subsets, but not the whole domain of testing, makes effective testing.
Now let us see in detail 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:



Input Domain for 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 cases combinations will be 199*199=39601.

(b) Invalid Inputs: The important thing in this case is the behaviour of the program as to how it responds when a user feeds invalid inputs. If we consider 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.

(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 to the input, which are not visible to the user.

(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 proceeds B in most cases, but B can also come first in rare and restricted condition. This is the race condition whenever B proceeds A. Race conditions are among the least tested.

There are too many paths through the Program to Test:

A program path can be traced through the code from the start of a program to its termination. A testing person thinks that if all the possible paths of the control flow through the program are executed, then possibly the program can be said completely tested. However there are two flaws in the statement:

(i) Consider the following segment:

```
for (i=0;i<n: i++)
        {
                if(m>=0)
                        x[i] = x[i] + 10;
                else
                        x[i] = x[i] – 2;
        }
```

In our example, there are two paths in one iteration. Now the total number of paths will be 2^n +1, where n is the number of times the loop will be carried out , and 1 is added for ―for loop‖ to terminate. Thus if n is 20, the number of paths will be 1048577.

(ii) The complete path testing, if performed somehow, does not guarantee that here will not be errors. For example, if a programmer develops a descending order program in place of ascending order program then exhaustive path testing is of no use.

Every Design Error cannot be found:

How do we know that the specifications are achievable?. Its consistency and completeness must be provided, and in general that is a provably unsolvable problem.

Software Testing Terminology:

Definitions:

Failure: It is the inability of a system or component to perform required function according to its specification. Failure indicates that External behavior is incorrect.

Fault/Defect/Bug: Fault is a condition that in actual causes a system to produces failure. Fault is a synonymous with words defect or bug.

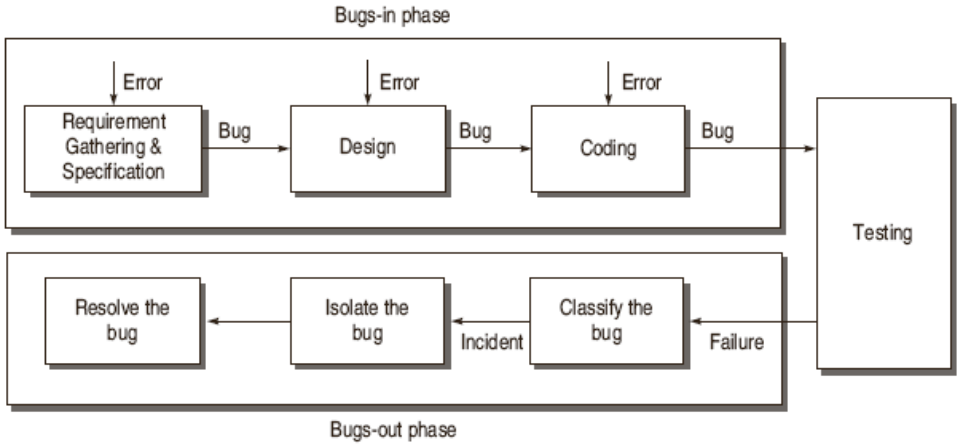
Error: Whenever a development team member makes a mistake in any phase of SDLC, errors are produced. It might be a typographical error, a misleading of a specification, a misunderstanding of what a subroutine does, and so on.

Test Case:

A test case is a document, which has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.

Test case parameters are Test case ID, Purpose, Preconditions, Inputs, Expected results.

Life Cycle of a Bug:



Bugs-In Phase: This phase is where the errors and bugs are introduced in the software. If the bug goes unnoticed in the starting stages then the bug is carried out to the subsequent stage. Thus, a phase may have its own errors as well as bugs received from the previous phase.

Bugs-Out Phase: When we observe failures, the following activities are performed to get rid of the bugs:

Bug Classification: In this part, we observe the failure and classify the bugs according to its nature. A bug can be critical or catastrophic or it may have no adverse effect on the behaviour of the software. This classification may help the developer to prioritize the handling of the bugs.

Bug Isolation: Bug Isolation is the activity by which we locate the module in which the bug appears.

Bug Resolution: Once we have isolated the bug, we back trace the design to pinpoint the location ofthe error. In this way, a bug is resolved when we have the exact location of its occurrence.

States of a Bug:

New: When the bug is posted for the first time, its state will be NEW means that the bug is not yet approved.

Open: After a tester has posted a bug, the test lead approves that the bug is genuine and he changes the state as OPEN.

Assign: Once the lead changes the state as OPEN, he assigns the bug to corresponding developer or development team and state of bug is changed to ASSIGN.

Test: Once the developer fixes the bug, he has to assign the bug to the testing tam for next round of testing. Before he releases the software with bug fixed, he changes the state to TEST. It means that bug has been fixed and is released to testing team.

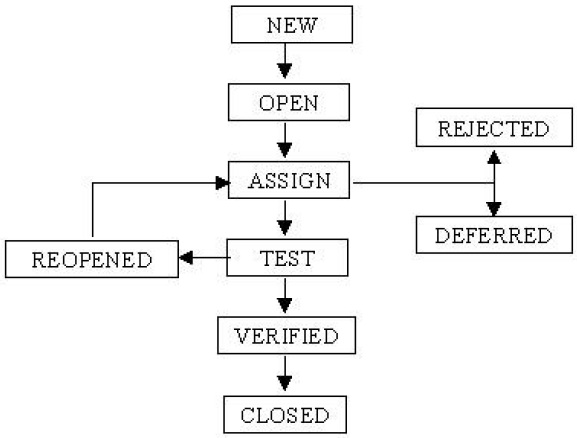
Deferred: The bug changed to DEFERRED state means the bug is expected to be fixed in next releases. There may be many factors for changing the bug to this state like priority of the bug may be low, lack of time for the release or the bug may not have major effect on the software.

Rejected: If the developer feels that the bug is not genuine, he rejects the bug. Then the state of the

bug is changed to REJECTED.

Verified/fixed: Once the bug is fixed and the status is changed to TEST, the tested tests the bug. If the bug is not present in the software, he approves that the bug is fixed and changes the status to VERIFIED.
Reopened: If the bug still exists even after the bug is fixed by the developer, the tester changes the status to REOPENED. The bug traverses the life cycle once again.

Closed: Once the bug is fixed, it is tested by the tester. If the tester feels that the bug no longer exists in the software, he changes the status of the bug to CLOSED. This state means that the bug is fixed, tested andapproved.



Why do Bugs Occur:

A software bug may be defined as a coding error that causes an unexpected defect, fault, flaw,or imperfection in a computer program. In other words, if a program does not perform as intended, it is most likely a bug.

There are bugs in software due to unclear or constantly changing requirements, software complexity, programming errors, timelines, errors in bug tracking, communication gap, documentation errors, deviation from standards etc.

- To Err is Human.
- In many occasions, the customer may not be completely clear as to how the product should ultimately function. Such cases usually lead to a lot of misinterpretations from any or both sides.
- Constantly changing software requirements cause a lot of confusion and pressure both on the development and testing teams. Often, a new feature added or existing feature removed can be linked to the other modules or components in the software. Overlooking such issues causes bugs.
- Also, fixing a bug in one part/component of the software might arise another in a different or same component.
- Designing and re-designing, UI interfaces, integration of modules, database management all these add to the complexity of the software and the system as a whole.
- Rescheduling of resources, re-doing or discarding already completed work, changes in hardware/software requirements can affect the software too. Assigning a new developer to the project in midway can cause bugs. This is possible if proper coding standards have not been

followed, improper code documentation, ineffective knowledge transfer etc. Discarding a portion of the existing code might just leave its trail behind in other parts of the software; overlooking or not eliminating such code can cause bugs. Serious bugs can especially occur with larger projects, as it gets tougher to identify the problem area.

- Programmers usually tend to rush as the deadline approaches closer. This is the time when most of the bugs occur. It is possible that you will be able to spot bugs of all types and severity.
- Complexity in keeping track of all the bugs can again cause bugs by itself. This gets harder when a bug has a very complex life cycle i.e. when the number of times it has been closed, reopened, not accepted, ignored etc goes on increasing.

Bugs Affect Economics of Software Testing:

Studies have demonstrated that testing prior to coding is 50% effective in detecting errors and after coding, it is 80% effective. Moreover, it is at least 10 times as costly to correct an error after coding as before and 100 times as costly to correct a post release error. This is how the bugs affect the economics of testing.

A bug found and fixed early stages when the specification is being written, cost very less. The same bug, if not found until the software is coded and tested, might cost ten times the cost in early stages.

Bug Classification based on Crticality:

A sample guideline for assignment of Priority Levels during the product test phase includes:

1. Critical — An item that prevents further testing of the product or function under test can be classified as Critical Bug. No workcaround is possible for such bugs. Examples of this include a missing menu option or security permission required to access a function under test.

2. Major — A defect that does not function as expected/designed or cause other functionalityto fail to meet requirements can be classified as Major Bug. The workaround can be provided for such bugs. Examples of this include inaccurate calculations; the wrong field being updated, etc.

3. Medium — The defects which do not conform to standards and conventions can be classified as Medium Bugs. Easy workarounds exists to achieve functionality objectives. Examples include matching visual and text links which lead to different end points.

4. Minor — Cosmetic defects which does not affect the functionality of the system can be classified as Minor Bugs.

Bug Classification based on SDLC:

Requirements and Specification Bugs:
The first type of bug in SDLC is in the requirement gathering and specification phase. Requirements and specifications developed can be incomplete ambiguous, or self-contradictory. They can be misunderstood or impossible to understand. The specifications that don't have flaws in them may change while the design is in progress. Some of the features are added, modified and deleted. Requirements, especially, as expressed in specifications are a major source of expensive bugs. The range is from a few percentage to more than 50%, depending on the application and environment. What hurts most about the bugs is that they are the earliest to invade the system and the last toleave.

Design Bugs:
Design bugs may be the bugs from the previous phase and in addition those errors which areintroduced in the current phase. The following design errors may be there:

Control flow bugs: Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, spaghetti code, and worst of all, pachinko code.

Logic bugs: Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and in combinations. Logic Bugs also includes evaluation of Boolean expressions in deeply nested IF-THEN-ELSE constructs.

Processing Bugs: Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing. Examples of processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of greater-than-or-equal etc

Data-Flow Bugs: Most initialization bugs are special case of data flow anomalies. A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediate use.

Error Handling Bugs: If the system fails, then there must be an error message or the system should handle the error in na appropriate way. If we forget to handle the exceptions, then error handling bugs will appear.

Boundary Related bugs: When the software fails at boundary values, then there will be Boundary related bugs. For example, there is one integer whose range is in between 1 to 100. User must test the program by entering 0 or 101.

User Interface bugs: Examples of UI bugs include inappropriate functionality of some feature, not doing what the user expects, wrong content in the help text etc....

Coding Bugs: Coding errors of all kinds can create any of the other kind of bugs. Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking. If a program has many syntax errors, then we should expect many logic and coding bugs. The documentation bugs are also considered as coding bugs which may mislead the maintenance programmers.

Interface and Integration Bugs:

Various categories of bugs in Interface, Integration are:

External Interfaces: The external interfaces are the means used to communicate with the world. These include devices, actuators, sensors, input terminals, printers, and communication lines. External interface bugs are: invalid timing or sequence assumptions related to external signals Misunderstanding external input or output formats. Insufficient tolerance to bad input data. *Internal Interfaces:* Internal interfaces are in principle not different from external interfaces but theyare more controlled. A best example for internal interfaces are communicating routines.

System Bugs: System bugs cover all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems. There can be no meaningful system testing until there has been thorough component and integration testing. System bugs are infrequent (1.7%)but very important because they are often found only after the system has been fielded.

Testing Bugs: Testers have no immunity to bugs. Tests require complicated scenarios and databases. They require code or the equivalent to execute and consequently they can have bugs.

Testing Principles:

--Effective Testing not Exhaustive Testing.

--Testing is not a single phase performed in SDLC.

--Destructive approach for constructive testing.

--Early testing is the best policy.

--Probability of existence of an error in a section of a program is proportional to the number oferrors already found in that section.

--Testing strategy should start at the smallest module level and expand towards the whole program.

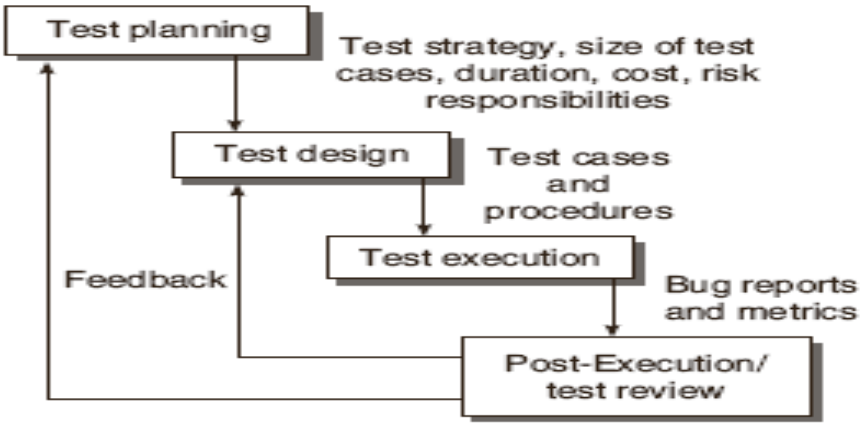
--Testing should be performed by an independent team.

--Everything must be recorded in software testing.

--Invalid inputs and unexpected behaviour have a high probability of finding an error.

--Testers must participate in specification and design reviews.

Software Testing Life Cycle:



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. STLC consists of following phases: Test Planning, Test Design, Test Execution, Post Execution / Test Review.

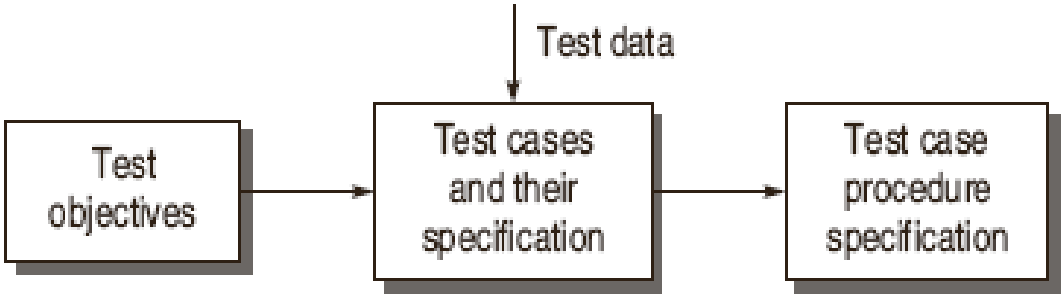
Test Planning:
Test planning, the most important activity to ensure that there is initially a list of tasks and

milestones in a baseline plan to track the progress of the project. It also defines the size of the test effort. The activities of test Plan are:

--To determine the scope and the risks that need to be tested and that are NOT to be tested.
--Documenting Test Strategy.
--Making sure that the testing activities has been included.
--Deciding Entry and Exit criteria.
--Evaluating the test estimate.
--Planning when and how to test and deciding how the test results will be evaluated, and defining test exit criterion.
--The Test artifacts delivered as part of test execution.
--Defining the management information, including the metrics required and defect resolution and risk issues.
--Ensuring that the test documentation generates repeatable test assets.

S.No.	Parameter	Description
1.	Test plan identifier	Unique identifying reference.
2.	Introduction	A brief introduction about the project and to the document.
3.	Test items	A test item is a software item that is the application under test.
4.	Features to be tested	A feature that needs to tested on the testware.
5.	Features not to be tested	Identify the features and the reasons for not including as part of testing.
6.	Approach	Details about the overall approach to testing.
7	Item pass/fail criteria	Documented whether a software item has passed or failed its test.
8	Test deliverables	The deliverables that are delivered as part of the testing process,such as test plans, test specifications and test summary reports.
9	Testing tasks	All tasks for planning and executing the testing.
10	Environmental needs	Defining the environmental requirements such as hardware, software, OS, network configurations, tools required.
11	Responsibilities	Lists the roles and responsibilities of the team members.
12	Staffing and training needs	Captures the actual staffing requirements and any specific skills and training requirements.
13	Schedule	States the important project delivery dates and key milestones.
14	Risks and Mitigation	High-level project risks and assumptions and a mitigating plan for each identified risk.
15	Approvals	Captures all approvers of the document, their titles and the sign off date.

Test Design:



--*Determining the test objectives and their properties*: The test objectives reflects the fundamental elements that need to be tested to satisfy an objective. For this purpose, you need to gather refernece materials like SRS and design documentation. Then on the basis of reference materials a team of experts compile a list of test objectives.

--*Preparing items to be tested*
--*Mapping Items to the test cases:* After making a list of test items to be tested, there is a need to identify the test cases. A matrix can be created for this purpose , identifying which test case will be covered by which item. The existing test cases can also be used for this mapping. This matrix will help in:
(a) Identifying the major test scenarios.
(b) Identifying and reducing the redundant test cases.
(c) Identifying the absence of a test case for a particular objective and as a result, creating them.

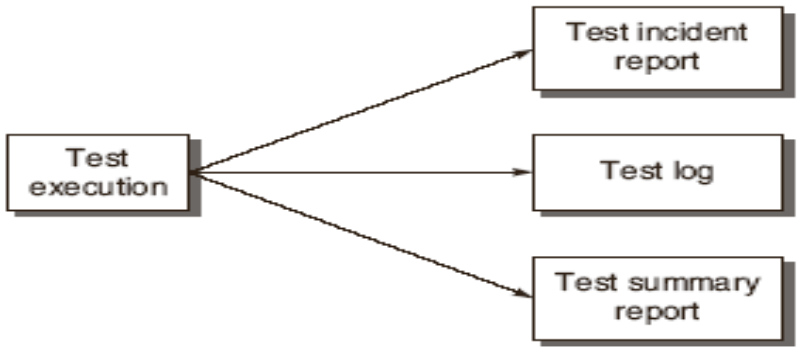
Selection of test case design techniques: While designing test cases, there are two broad categories, namely **black-box testing** and **white-box testing.**

Creating test cases and test data: The test cases mention the objective under which a test case is being designed, the inputs required, and the expected outputs. While giving input specifications, test data must also be chosen.

Setting up the test environment and supporting tools: Details like hardware configuration, testers, interfaces, operating systems and manuals must be specified during this phase.

Creating test procedure specification: It shows the description of how the test cases will be run. It is in the form of sequenced steps. This procedure is actually used by the tester at the time of execution of test cases.

Test Execution:



In this phase, all the 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:

This phase is to analyze bug-related issues and get feedback that maximum number of bugs can be removed. As soon the developer gets the bug report, he performs the following activities: Understanding the bug, Reproducing the bug, Analyzing the nature and cause of the bug.

Relating test life cycle to development life cycle:

S. No.	Phase	SDLC - Software Development Life cycle	STLC - Software Test Life Cycle
1	Requirements Gathering	Requirements gathering is done by business analyst. Development team analyze the requirements from the design, architecture & coding perspective.	Testing team also review & analyze the requirements. Testing team identifies the testing requirements like what types of testing will be required and review the requirements
2	Design	Technical architect works for the high level & low design of the software. Business analyst works for the UI design of the application	Here, test architect generally the test lead/manager, does the test planning, identify high level testing points.
3	Coding or development	Development team does the actual coding based on the designed architecture.	Testing team write the detailed test cases.
4	Testing	In SDLC, actual testing is carried out in this phase. It includes unit testing, integration testing & system testing etc..	Test Execution and bug reporting, manual testing, automation testing is done, defects found are reported. Re-testing and regression testing is also done in this phase.

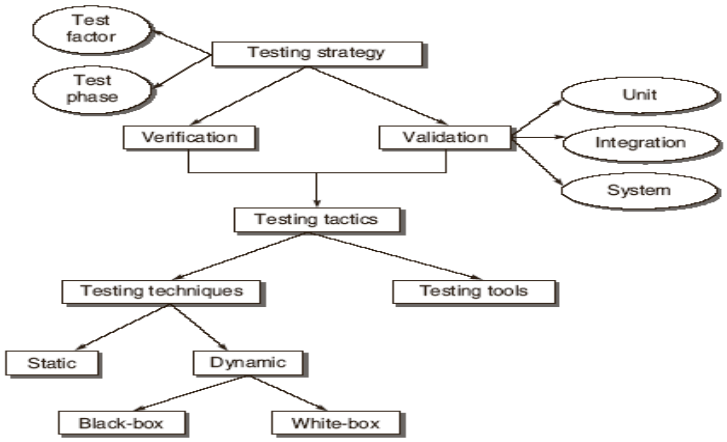
5	Deployment	Application is deployed on production environment for real end users.	Final testing and implementation is done in this phase and final test report is prepared.
6	Maintenance	Maintenance testing is carried out in this phase.	

Software Testing Methodology:Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved.

Software Testing Strategy: Testing strategy is the planning of the whole testing process into a well planned series of steps. The components of testing strategy are:

Test Factors: Test factors are risk factors or issue related to the system under development. Risk factors need to be selected and ranked according to a specific system under development.

Test Phase: It refers to the phases of SDLC where testing will be performed.



Test Strategy Matrix:

Test starategy matrix:

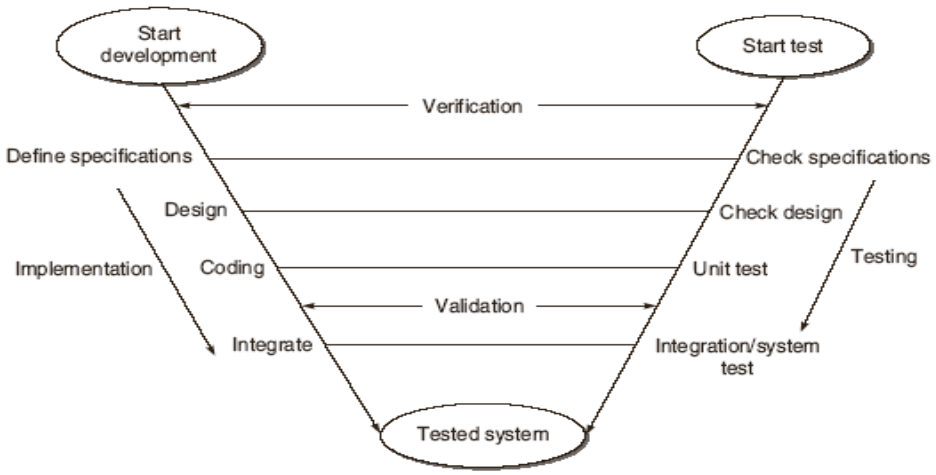
A test strategy matrix identifies the concerns that will become the focus of test planning and execution. The matrix is prepared using test factors and test phase. The steps to prepare the test matrix are:
--select and rank test factors,
--Identify system development phases,
--Identify risks associated with the system under development.

Development of Test Strategy: A test strategy includes testing the components being built for the system, and 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.

Verification	**Validation**
Evaluates the intermediary products to check whether it meets the specific requirements of the particular phase	Checks whether the product is built as per the

specified requirement and design specification.	Evaluates the final product to check whether it meets the business needs. It determines whether the software is fit for use and satisfies the business need.
Checks —Are we building the product right‖?	Checks —Are we building the right product‖?
This is done without executing the software	Is done with executing the software
Involves all the static testing techniques	Includes all the dynamic testing techniques.
Examples includes reviews, inspection and walkthrough	Example includes all types of testing like smoke, regression, functional, system.

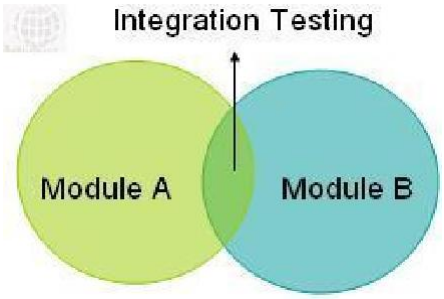
Testing Life cycle Model: Verification and Validation (V & V) are building blocks of a testing process. The formation of test strategy is based on these two terns only. V & V can be best understood when these are modeled in the testing process.



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 software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing is often automated but it can also be done manually.

Integration testing: Integration testing tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardware or interfaces between systems. Also after integrating two different components together we do the integration testing. As displayed in the image below when two different modules _Module A' and _Module B' are integrated then the integration testing is done.



System Testing (ST):

System Testingis a black box testing technique performed to evaluate the complete system the system's compliance against specified requirements. In System testing, the functionalities of the system are tested from an end-to-end perspective.

Testing Tactics: The testing techniques can be categorized into two parts: (a) Static Testing, (b) Dynamic Testing.

Static Testing: Static testing is the testing of the software work products manually, or with a set of tools, but they are not executed. It starts early in the Life cycle and so it is done during the verification process. It does not need computer as the testing of program is done without executing the program. For example: reviewing, walk through, inspection, etc.

Dynamic Testing: Dynamic Testing is a kind of software testing technique using which the dynamic behaviour of the code is analysed. For Performing dynamic, testing the software should be compiled and executed and parameters such as memory usage, CPU usage, and response time of the software are analyzed. Dynamic testing is further divided into : (i) Black-box testing (ii) White-box testing

(i) Black-box testing: Black-box testing is a method of software testing that examines the functionality of an application based on the specifications. It is also known as Specifications based testing. This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing.

(ii) White-box testing: White box testing is a testing technique, that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

Considerations in Developing Testing Methodologies: The different considerations for developinga test strategy are:

--Determine Project Risks

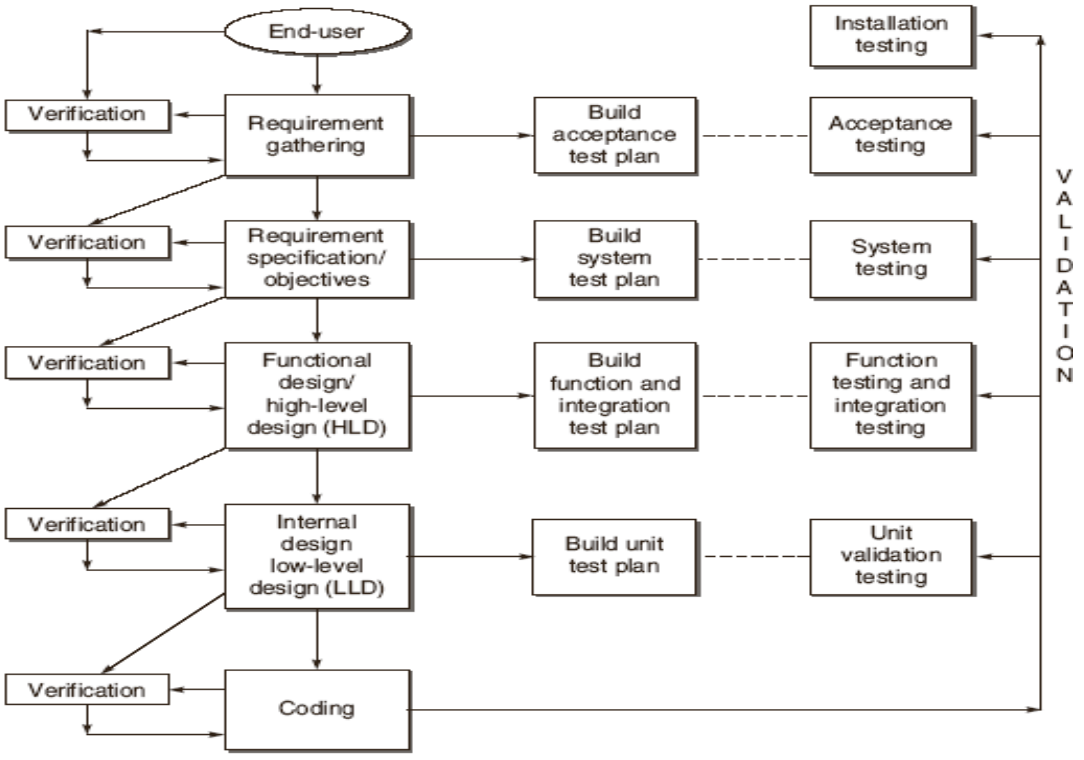
--Determine the Type of Development Project

--Identify test Activities according to SDLC Phase

--Build the Test Plan.

Verification and Validation

Verification & Validation Activities (V&V):



V&V activities can be best understood with the help of phases of SDLC activities:

Requirements Gathering: It is an essential part of any project and project management. Understanding fully what a project will deliver is critical to its success.

Requirement Specification or Objectives: Software Requirements Specification (SRS) is a description of a software system to be developed, laying out functional and non-functional requirements, and may include a set of use cases that describe interactions the users will have with the software. Software requirements specification establishes the basis for an agreement between

customers and contractors or suppliers on what the software product is to do as well as what it is not expected to do.

High Level Design or Functional Design: A functional design assures that each modular part of a device has only one responsibility and performs that responsibility with the minimum of side effects on other parts. Functionally designed modules tend to have low coupling. High-level design (HLD) explains the architecture that would be used for developing a software product. The architecture diagram provides an overview of an entire system, identifying the main components that would be developed for the product and their interfaces. The HLD uses possibly nontechnical to mildly technical terms that should be understandable to the administrators of the system.

Low-Level Design (LLD): It is a component-level design process that follows a step-by-step refinement process. This process can be used for designing data structures, required software architecture, source code and ultimately, performance algorithms. Overall, the data organization may be defined during requirement analysis and then refined during data design work. Post-build, each component is specified in detail. The LLD phase is the stage where the actual software components are designed.

Coding: The goal of the coding phase is to translate the design of the system into code in a given programming language. The coding phase affects both testing and maintenance profoundly. A well written code reduces the testing and maintenance effort. Since the testing and maintenance cost of software are much higher than the coding cost, the goal of coding should be to reduce the testingand maintenance effort. Hence, during coding the focus should be on developing programs that are easy to write. Simplicity and clarity should be strived for, during the coding phase.

Verification: Verification is done at the starting of the development process. It includes reviews and meetings, walkthroughs, inspection, etc. to evaluate documents, plans, code, requirements andspecifications. It answers the questions like:

--Am I building the product right?

--Am I accessing the data right (in the right place; in the right way).

--It is a Low level activity

According to the Capability Maturity Model(CMMI-SW v1.1) we can also define verification as"the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [IEEE-STD-610]".

Advantages of Software Verification :

--Verification helps in lowering down the count of the defect in the later stages of development.

-- Verifying the product at the starting phase of the development will help in understanding the product in a better way.

--It reduces the chances of failures in the software application or product.
--Ithelps in building the product as per the customer specifications and needs.

The goals of verification are:

--Everything must be verified.

--Results of the verification may not be binary.

--Even implicit qualities must be verified.

Verification of Requirements:

In this type of verification, all the requirements gathered from the users viewpoint sre 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 testers works in parallel by performing the following two tasks:

1. The tester reviews the acceptance criteria in terms of its completeness, clarity and testability.
2. The tester prepares the Acceptance Test Plan which is referred at the time of Acceptance Testing.

Verification of Objectives: After gathering of objectives specific objectives are prepared consedering every specification. These objectives are prepared in a document called SRS. In this activity the tester performs two parallel activities:

1. The tester verifies all the objectives mentioned in SRS.
2. The tester also prepares the System Test Plan which is based on SRS.

***How to verify Requirements and Objectives**:*

An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement. In general any ambiguous requirement is not verifiable. Non verifiable requirements include statements such as “works well", “good human interface", and “shall usually happen". These requirements cannot be verified becauseit is impossible to define the terms “good", “well", or “usually". An example of a verifiablestatement is *“**Output of the program shall be produced within 20 s of event x 60% of the time; and shall be produced within 30 s of event x 100% of the time".*** This statement can be verified because it uses concrete terms and measurable quantities. If a method cannot be devised to determine whether the software meets a particular requirement, then that requirement should be removed or revised. Following are the points against which every requirement in SRS should be verified.

Correctness: An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet. However, generally speaking there is no tool or procedure to ensure correctness. That’s why the SRS should be compared to superior documents (including the System Requirements Specification, if exists) during a review process in order to filter out possible contradictions and inconsistencies. Reviews should also be used to get a feedback from the customer side on whether the SRS correctly reflects the actual needs. This process can be made easier and less error-prone by traceability.

Unambiguity. An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. As a minimum, this requires that each characteristic of the final product be described using a single unique term. In cases where a term used in a particular context could have multiple meanings, the term should be included in a glossary where its meaning is made more specific.

Completeness. An SRS is complete if, and only if, it includes the following elements:
--All significant requirements imposed by a system specification should be acknowledged andtreated.
--Full labels and references to all figures, tables, and diagrams in the SRS and definition of all termsand units of measure.

However, completeness is very hard to achieve, especially when talking about business systemswhere the requirements always change and new requirements raise.

Consistency: Consistency refers to internal consistency. If an SRS does not agree with some higher-level document, such as a system requirements specification, then it is a violation of correctness. An SRS is internally consistent if, and only if, no subset of individual

requirements described in it conflict. The three types of likely conflicts in an SRS are as follows:

--The specified characteristics of real-world objects may conflict. For example, the format of an output report may be described in one requirement as tabular but in another as textual or one requirement may state that all lights shall be green while another may state that all lights shall beblue.

--There may be logical or temporal conflict between two specified actions. For example, one requirement may specify that the program will add two inputs and another may specify that the program will multiply them or one requirement may state that “A" must always follow “B", whileanother may require that “A and B" occur simultaneously.

--Two or more requirements may describe the same real-world object but use different terms for thatobject. For example, a program’s request for a user input may be called a “prompt" in one requirement and a “cue" in another. The use of standard terminology and definitions promotes consistency.

Modifiability or Updation: An SRS is modifiable if, and only if, its structure and style are such thatany changes to the requirements can be made easily, completely, and consistently while retaining the structure and style. Modifiability generally requires an SRS to

a. Have a coherent and easy-to-use organization with a table of contents, an index, and explicitcross referencing;
b. Not be redundant (i.e., the same requirement should not appear in more than one place in theSRS);
c. Express each requirement separately, rather than intermixed with other requirements.

Traceability*:* An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. The following two types of traceability are recommended:

a. Backward traceability (i.e., to previous stages of development). This depends upon eachrequirement explicitly referencing its source in earlier documents.
b. Forward traceability (i.e., to all documents spawned by the SRS). This depends upon eachrequirement in the SRS having a unique name or reference number.

Verification of High Level Design*:*

Like the verification of requirements, the tester is responsible for two parallel activities in this placeas well:

1. The tester verifies the high level design. Since the system has been decomposed in a number of sub-system or components, the tester should verify the functionality of these components and their interfaces. Every requirement in SRS should map the design.
2. 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.

How to verify High Level Design:

Verification of Data Design

--Check whether 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 requirements in SRS.

Verification of Architectural Design

--Check that every functional requirement in SRS has been take care in this design.

--Check whether all exceptions handling conditions have been taken care.

--Verify the process of transform mapping and transaction mapping used for transition from therequirement model to architectural design.

--Check the functionality of each module according to the requirements specified.

--Check the inter-dependence and interface between the modules.

--Coupling and Module Cohesion.

Verification of User-Interface Design

--Check all the interfaces between modules according to architecture design.

--Check all the interfaces between software and other non-human producer and consumer ofinformation.

--Check all the interfaces between human and computer.

--Check all the above interfaces for their consistency.

--Check that the response time for all the interfaces are within required ranges.

--Help Facility error messages and warnings.

***Verification of Low Level Design (LLD)**:*

In LLD,a detailed design of modules and data are prepared such that an operational software is ready. The details of each module or units is prepared in their separate SRS and SDD (Software Design Documentation). Testers also perform the following parallel activities in this phase:

--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.

--The tester also prepares the Unit Test Plan which will be referred at the time of Unit Testing.

***How to verify Low Level Design (LLD)**:*

--Verify the SRS of each module.

--Verify the SDD of each module.

In LLD, data structures, interfaces and algorithms are represented by design notations; so verify theconsistency of every item with their design notations.

***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. Since LLD 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.

--Verify every statement, control structure, loop, and logic.

--Misunderstood or incorrect Arithmetic precedence.

--Mixed mode operations

--Incorrect initialization

--Precision Inaccuracy

--Incorrect symbolic representation of an expression

--Different data types

--Improper or nonexistent loop termination

--Failure to exit.

Validation:

Validation is a set of activities that ensures the software under consideration has been built right andis traceable to customer requirements. The reason for doing validation are:

--Developing tests that will determine whether the product satisfies the users' requirements, asstated in the requirement specification.
--Developing tests that will determine whether the product's actual behavior matches the desiredbehavior, as described in the functional design specification.
--The bugs, which are still existing in the software after coding need to be uncovered.
--Last chance to discover the bugs otherwise these bugs will move to the final product released tothe customer.
--Validation enhances the quality of software.

Validation Activities*:*

The Validation activities are divided into ***Validation Test Plan*** and ***Validation Test Execution*** whichare 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 and Validation. For preparing a validation test plan, testers must follow the following points:
--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 preparethe related test plans which are used at the time of validation testing.

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

Acceptance Test Plan: The test plan document describes a series of operations to perform on the software. Whenever there is a visible change in the state of the software (new window, change in the display, etc.) the test plan document asks the testers to observe the changes, and verify that they are as expected, with the expected behavior described precisely in the test plan.

System Test Plan**:** This plan i s prepared to verify the objectives specified in the SRS. The plan isused at the time of System Testing.

Fuction Test Plan**:** The functional test plan is not testing the underlying implementation of the application components. It is testing the application from the customer's viewpoint. The functionaltest is concerned with how the application is meeting business requirements**.**

Integration Test Plan**:** Integration test planning is carried out during the design stage. An integration test plan is a collection of integration tests that focus on functionality.

Unit Test Plan**:** This document describes the Test Plan in other words how the tests will be carried out. This will typically include the list of things to be Tested, Roles and Responsibilities, prerequisites to begin Testing, Test Environment, Assumptions, what to do

after a test is successfully carried out, what to do if test fails, Glossary and so on.

Validaion Test Execution:

***Unit Validation Testing*:** A unit is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.

- Unit tests are basically written and executed by software developers to make sure that code meetsits design and requirements and behaves as expected.
- The goal of unit testing is to segregate each part of the program and test that the individual partsare working correctly.
- This means that for any function or procedure when a set of inputs are given then it should returnthe proper values. It should handle the failures gracefully during the course of execution when anyinvalid input is given.
- A unit test provides a written contract that the piece of code must assure. Hence it has severalbenefits.

***Integration Testing*:** Integration testing tests integration or interfaces between components, interactions to different parts of the system such as an operating system, file system and hardwareor interfaces between systems.

- Also after integrating two different components together we do the integration testing. Asdisplayed in the image below when two different modules 'Module A' and 'Module B' areintegrated then the integration testing is done.

In system testing the behavior of whole system/product is tested as defined by the scope of thedevelopment project or product.

- It may include tests based on risks and/or requirement specifications, business process, use cases, or other high level descriptions of system behavior, interactions with the operating systems, and system resources.

***Functional Testing*:** Functional testing verifies that each **function** of the software application operates in conformance with the requirement specification. This testing mainly involves black box testing and it is not concerned about the source code of the application. Each and every functionality of the system is tested by providing appropriate input, verifying the output and comparing the actual results with the expected results. This testing involves checking of User Interface, APIs, Database, security, client/ server applications and functionality of the Application under Test. The testing can be done either manually or using automation

***System Testing*:** System testing is most often the final test to verify that the system to be deliveredmeets the specification and its purpose. System testing is carried out by specialist testers or independent testers. System testing should investigate both functional and non-functional requirements of the testing.

***Acceptance testing*:** After the system test has corrected all or most defects, the system will be delivered to the user or customer for acceptance testing. Acceptance testing is basically done by the user or customer although other stakeholders may be involved as well. The goal of acceptance testing is to establish confidence in the system. Acceptance testing is most often focused on avalidation type testing.