UNIT – 1

1. **Introduction**: Software is an essential component of any electronic device or system. The demand for quality assurance (QA) in software products is dynamic and software testing (ST) has thus gained importance since the last decade.

   Software QA (**SQA**) is an important part of any software project and organisations have separate testing & debugging departments/teams. The presence of bugs in 'ready-to-be-released' software will delay it release, degrade its value in the market and ultimately the project vaporises.

   It should be realized that job trends are shifting towards testers rather than developers and with the evolution of any new concept, it should be tested first. The academia is not moving in parallel to the industry and its requirements. To bridge this gap, a study and understanding of ST as a full-fledged concept has to be taken up as a separate course.

2. Industry mainly relies upon automated testing tools to speed up the activity. But there exist umpteen situations where a product can't be tested through automated testing – it may not even save the project. So, all the testers and learners should also utilize the concept of designing the test cases, executing them and preparing the Test/Bug Reports. A testing team should be in constant touch with a developer team and go in parallel to make the project a grand success.

   The testing process should also be 'measured' using models & metrics to make sure of its quality and levels. The goals are also to be reached within the stipulated time. Capability Maturity Model (CMM) has measured the development process (1-5 scale) and companies are rushing for the highest scale.

   But many people in the industry/academia don't realise the need for measuring the testing, its level, and its place in the model being used. A TMM also exists that measures the maturing status of the testing process.

   Hence, the academia has to mould itself to the demands of industry and lend a helping hand to them. New testing metrics have to be researched and produced which should meet the industry needs and set standards for the testing process.

3. Testing is an important part of the software development life cycle (SDLC) and should be given adequate importance. Separate testing groups should come into existence, more in number, to divide the tasks among them so as to identify the errors in a pin-pointed manner and also suggest the remedies to get rid of them.

   It should also be noted that testing and debugging can't be carried out 100%. It is better to come out with *effective testing* instead of *exhaustive testing*.

The psychology of the tester should be in a 'negative' manner – bugs should be tried to found out and the program should be crashed. The program/project has to withstand this aspect and come out clean. A tester should try to show the errors – not their absence.

4. **Evolution of ST**: Previously, ST was considered only as a debugging process for removing errors, after the software development. By 1978, Myers stressed that ST should be dealt separately and ST should be done to find the errors, not to prove their absence. Testing tools appeared in market by 1990s but could not solve all the problems or replace testing itself with a pre-planned & automated process.

Debugging Oriented Phase	**Demonstration Oriented Phase**	**Destruction Oriented Phase**	**Evaluation Oriented Phase**	**Prevention Oriented Phase**	**Process Oriented Phase**
Checkout getting the system to run	Checkout of a program increased from program runs to program correctness	Separated Debugging from testing	Quality of the software	Bug prevention rather than bug detection	Testing is a process rather than a single phase
Debugging		Testing is to show the absence of errors	Verification and validation techniques		
		Effective testing			

1957 1979 1983 1988 1996

Fig Evolution phases of software testing

5. (a) **Debugging Oriented Phase**: In this early period of testing, ST basics were unknown. Programs were written, run, tested and debugged by programmers until they were sure that all the bugs were removed. The term 'checkout' was used for testing. There was no clear distinction between the terms software development, testing and debugging.

(b) **Demonstration-Oriented Phase**: It was noted in this phase that the purpose of 'checkout' is not only to run the program (without obstacles) but also to demonstrate its correctness. The importance of 'absence of errors' was also recognised. Note that there was there was no stress on test case design here.

(c) **Destruction-Oriented Phase**: This phase is a revolutionary turning point in the ST history. Myers changed the view of testing from 'showing the absence of errors' to 'detection of errors'. Effective testing was given more importance in comparison to exhaustive testing.

(d) **Evaluation-oriented Phase**: This phase gives more weight to SQA of the product at an early stage, in the form of verification and validation activities. Standards for V & V were also set and released so that the software can be evaluated at each step of software development.

(e) **Prevention-Oriented Phase**: This phase more importance to the point that *instead of detecting the bugs, it is better to prevent them* by experience. The prevention model includes test planning, test analysis and test design activities.

NOTE: Evaluation model relied more on analysis and review techniques rather than testing directly.

(f) **Process-oriented Phase**: Here, testing was *established* as a complete process rather than a single phase in Software Development Life Cycle (SDLC). The testing process starts at the requirements phase of the SDLC and runs in parallel. A model for measuring the performance of testing was also developed: CMM and TMM.

6. **Another angle of ST evolution** states 3 phases:

(a) **ST 1.0**: ST was just a single phase in SDLC. No testing firm/organisation existed but a few testing tools were present.

(b) **ST 2.0**: ST gained prominence in SDLC and the concept of ‘early testing’ started. Planning of test resources also gained ground and the testing tools increased in number and quality.

(c) **ST 3.0**: ST became a matured process based on strategic effort. It was established that an overall process should be present to give a roadmap of all testing techniques and methodologies. It should be driven by quality goals and monitored by the managers.

7. **ST-Myths and Facts**:

- **Myth1**: Testing is a single phase of SDLC.
- **Truth1**: In reality, testing starts as soon as we obtain the requirements/specifications. Testing continues throughout the SDLC and even after the implementation of the software.
- **Myth2**: Testing is easy.
- **Truth2**: Testing tools automate some of the tasks but not test cases’ designing. A tester has to design the whole testing process and develop the test cases manually – which means that he has to understand the whole project and process. Typically, a tester’s job is harder than that of a developer.
- **Myth3**: Software development is more important than testing.
- **Truth3**: This myth exists from the beginning of a career and testing is considered a secondary job. These days, testing surfaced as a complete process, same as that of development. A testing team enjoys equal status to that of a development team.
- **Myth4**: Complete Testing is possible.

- **Truth4**: Any person who has not worked on test cases might feel this. In reality, complete testing is never possible. All the inputs can’t be perceived and provided since it makes the testing process too large. Hence, ‘complete/exhaustive’ testing has been replaced by ‘effective’ testing.
  NOTE: Effective testing selects and runs some important test cases so that severe bugs are uncovered first.
- **Myth5**: Testing starts after program development.
- **Truth5**: This idea is not proper. A tester performs testing at the end of every phase of SDLC in the form of verification and validation testing. Detailed test plans and cases are prepared and reports are forwarded to the developers’ team. Testing after coding is just a small part of the testing process.
- **Myth6**: The purpose of testing is to check the functionality of the software.
- **Truth6**: The goal of testing is to ensure quality of the software. Quality doesn’t mean checking the functionalities of all the modules; many other points are also to be taken into account.
  NOTE: Quality => Standards; functionality => different operations to be carried out
- **Myth7**: Anyone can be a tester.
- **Truth7**: Testing can be performed only after a tester is properly trained for various purposes like understanding the SDLC phases designing the test cases and learning about various testing tools. Companies give preference to certified testers.

8. **Goals of ST**:

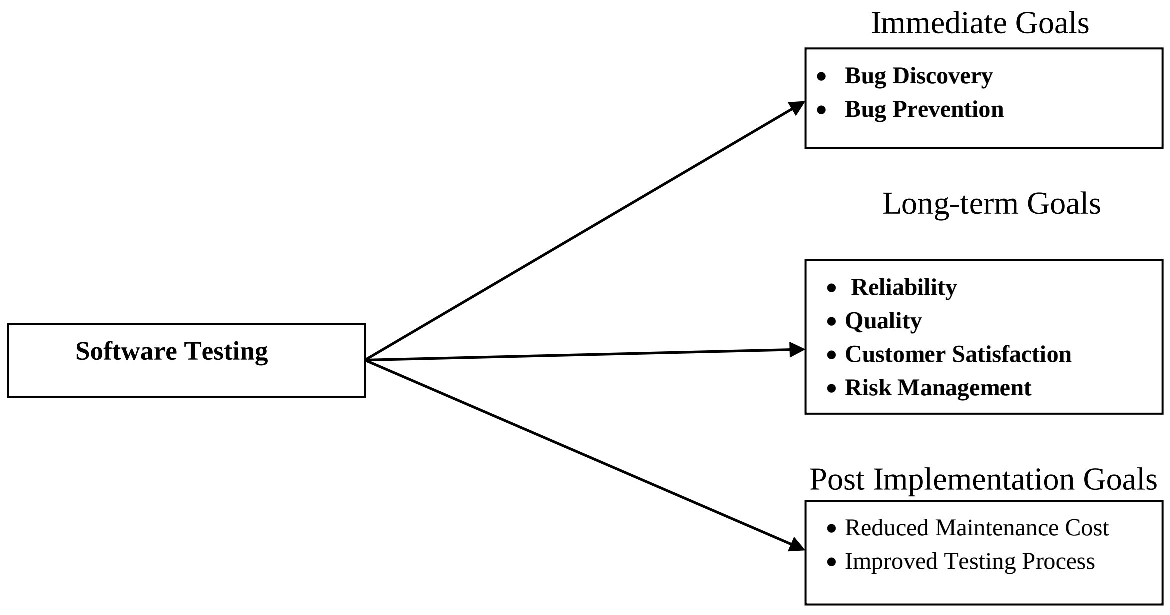

Fig Software Testing Goals

The goals of ST can be classified into three major categories:

(a) **Short Term or Immediate Goals**: These goals are the immediate results after performing testing procedure. They can be set in the individual phases of SDLC.

- **Bug Discovery**: Find errors at any stage of software development. More bugs discovered at an early stage => more success rate of the project.
- **Bug Prevention**: It is the consequent (result) action of bug discovery. From the discovery, behaviour, and interpretation of the bugs unearthed, we get to know how to design the code safely so that the bugs may not be repeated. Bug prevention is a superior goal of testing.

(b) **Long Term Goals**: These affect the 'product quality' in the long run, when a cycle of the SDLC is completed.

- **Quality**: through testing ensures superior quality of the software. Thus the first goal of a tester is to enhance the quality of the software product. Quality depends on various factors like technical correctness, integrity, efficiency and so on but reliability is the major factor in this idea. It is a matter of confidence that the software will not fail – and confidence increases with rigorous testing, thus increasing reliability and quality.

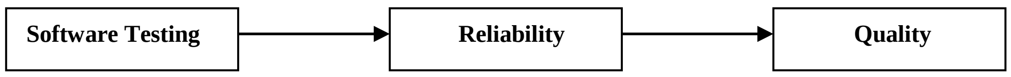

Fig : Testing produces reliability and quality

- **Customer Satisfaction**: Testing must satisfy the user for all specified requirements and other unspecified requirements also.

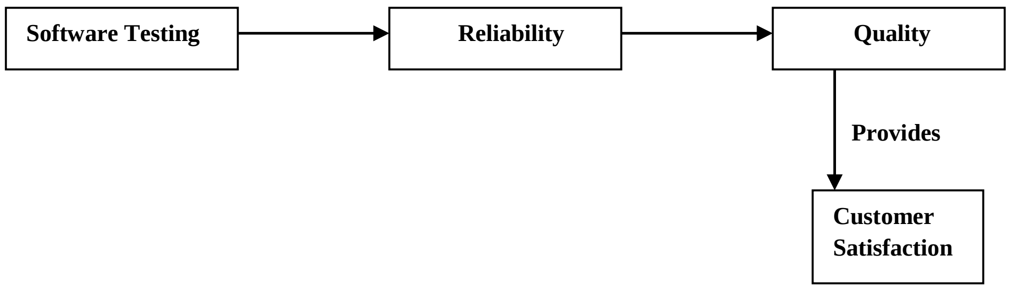

Fig : Quality leads to customer Satisfaction

- **Risk Management:** Risk is the probability that undesired events will occur in a system leading to unsuccessful termination of goals or project. Risks must be controlled by ST (as a control) which can help in minimizing or terminating risks.

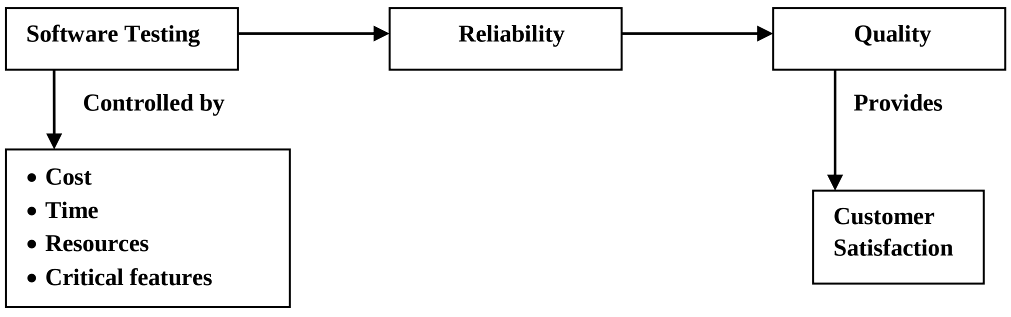

Fig: Testing controlled by risk factors

The purpose of ST as a control is to provide information to management so that they can react in a better way to risk situations. Ex: Time is less, more chance of finding bugs etc.

A tester's responsibility is to evaluate business risks and make them a basis for choosing the tests. Levels of the risks should be categorised as high, moderate, low so that it becomes a foundation for designing the test cases. Risk management becomes a long term goal for ST and has to be dealt with carefully to extract positive results.

(c) **Post-implementation Goals**: [After the product is released]

- **Reduced maintenance Cost**: In software market, maintenance cost is the cost of correcting newly surfaced errors. Post release errors are more costly since that are difficult to trace and fix. If testing had been done rigorously, chances of failures are minimized, thus reducing maintenance cost.
- **Improved ST Process:** The bug history (of previous projects) and post-implementation results should be analysed to predict the time and place of bug surfacing in the current project. Note that this methodology will be useful for the upcoming projects.

9. **Psychology for ST**: ST is directly related to human psychology. It is defined as a process of demonstrating that 'no errors are present'. [positive approach]

If testing is performed with this kind of approach, humans tend to produce test cases that go in the same kind of route as that of the software and the final result will be the software is fully bug-free.

If this process of testing is reversed, presuming that bugs exist everywhere, the domain of testing becomes larger and the chance of unearthing the bugs increases. This 'negative approach' is taken up for the benefit of both developer and tester and the result is that many unexpected bugs might get found. This methodology is *effective testing*.

Thus ST can be defined as a process of executing a program with the intent of finding maximum errors.

A developer should never be embarrassed if bugs are found – humans tend to make errors. On the other hand, a tester should concentrate on revealing maximum number of errors and try to bring the program to a halt.

Testing can be compared with the analogy of medical diagnostics of a patient. If lab tests are negative, it can't be regarded as a successful test. If they are positive, then the doctor can start appropriate treatment immediately.

Hence successful test => errors have been found
Unsuccessful test => found no errors.

10. **Definition of ST**:
    (a) Testing is the process of executing a program with the intent of finding errors.
    (b) A successful test uncovers an undiscovered/new error.
    (c) Testing can show the presence of bugs but never their absence.
    (d) The aim of testing is to verify the quality of the software by running the software in controlled circumstances.
    (e) ST is an investigation conducted to provide information about the quality of the product WRT the operating context.
    (f) ST is a concurrent lifecycle process of engineering, using, and maintenance of testing methodologies in order to improve the quality of the concerned software.

NOTE: Quality is the primary goal of testing; testing team is not responsible for QA. Finally, ST can be defined as a process that detects important bugs with the objective of having better quality software.

11. **Model for Software Testing**:

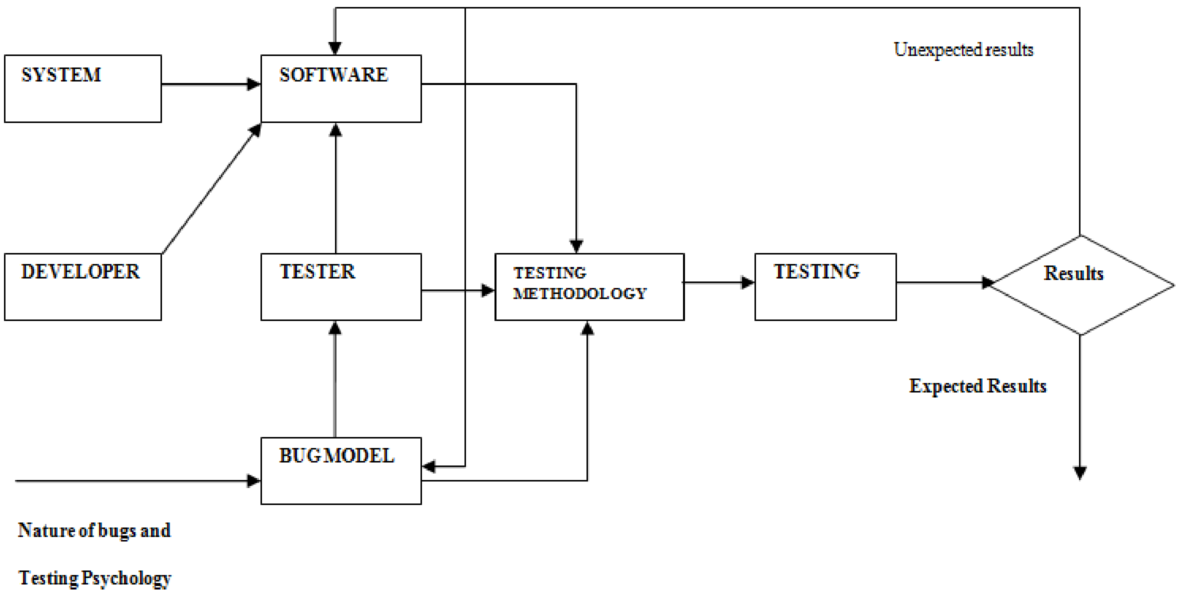

Fig: Software Testing Model

Since testing plays an important role in the success of s/w project, it should be performed in a planned manner. To attain this, all aspects related to ST should be considered in a suitable approach.

The s/w is a part of the total system for which it would be used. The developer produces the s/w considering its testability – a deficiently planned s/w may be difficult to test. Testers start their work as soon as requirements are specified and they work on the basis of a bug model.

NOTE: Testability is the level up to which s/w system supports ST.

A bug model classifies the bugs based on their importance or the SDLC phase in which testing is to be performed.

A testing methodology is finalised based on s/w type and bug model and it specifies how the testing process is to be carried out.

If the results are closer to the desired goals, it is deduced that testing is successful; else the s/w, or the bug model, or the testing methodology itself may have to be modified. This process continues till we obtain satisfactory results.

12. It should be supervised that the s/w under construction is not too complex for testing. The s/w should be testable at every point.

    A bug model provides an expectation about what type of bugs might surface. It should help in designing a testing strategy. Every bug can't be predicted thus making modifications necessary when bugs appear at a higher (or lower) rate than the expected one. Ex: Boundary errors, control flow errors, data errors, hardware errors, loading/memory errors, testing errors (no indication of important test cases, no bug reports, misinterpretations, can't reduce the problems etc.).

    Test plan/methodology is based on both the s/w (SDLC) and bug models. It gives well-defined steps for the overall testing process, taking care of the risk factors also. Once a testing methodology is in place, test cases can be designed and testing tools can be applied at various steps. If we don't obtain the expected results, modifications are apparently necessary.
13. **Effective ST vs. Exhaustive ST**:
    Exhaustive/Complex Testing: Every statement in the program and every possible path combination of data must be executed. 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.

    NOTE: Testing process is a domain of possible tests where subsets exist.

    Available computer speed and the time constraints limit the possibility of performing all the tests. Hence, testing must be performed on selected subsets with constrained resources.

    Effective testing can be enhanced if subsets are selected based on the features that are important for the concerned s/w environment.

    To prove that that a testing domain is too large to deal with, the following parts are touched.

Valid Inputs: Add two numbers of two-digits; range is from -99 to 99 (total numbers are 199). Test case combinations = 199 * 199 = 39,601. Similarly, for four digit numbers, test case combinations = 399,960,001. Is it possible to write those many test cases? Certainly not.

Invalid Inputs: Invalid inputs can also be used to observe the s/w response to them. It should be noted that the set of invalid inputs is also very large to test. We might have to consider numbers out of range, combination of alphabets and digits, combination of all alphabets, control characters and so on.

Edited Inputs: If we can edit inputs just before providing them, unexpected events will take place. Ex: Add invisible spaces in input data, press a number, press backspace continuously and press another key. This will result in buffer overflow and system hangs.

Race Condition Inputs: The timing variation between two or more inputs is also one of the limitations of testing.
Ex: A, B are the two inputs to be provided; A comes before B every time. If B comes before A suddenly, the system might crash. This is called race condition bug. Race conditions are among the least tested bugs.

14. **There are too many paths in the program to test**: A program path can be traced through the code from the start to termination. Two paths differ if the program executes different statements in each of them or same statements in different order. If all paths are executed, a tester might think that the program is completely tested. But problems still remain:

(a) A do-while loop with 20 statements iterating 20 times.
(b) If a nested loop exists, it's virtually impossible to test all the paths.

```
Ex: for(int i=0;i<n;i++)         1
      {                          2
        if(m>=0)                 3
           x[i] = x[i] + 10;     4
        else                     5
            x[i] = x[i] – 2;     6
       }                         7
```

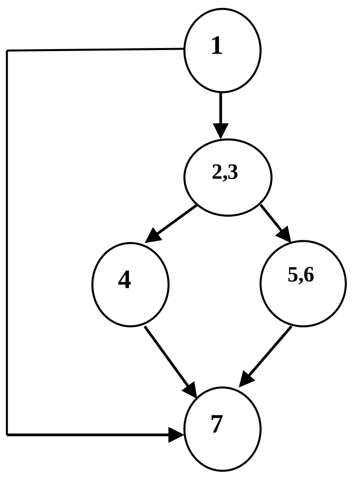

Fig: Flowchart

There are two paths in one iteration at (2,3).
∴ Total no. of paths = 2^n+1, where n → no. of times the loop is executed
1 is added since the ‘for’ loop will exit after its looping ends and terminates.
If n=20 (say), no. of paths = $2^{20}+1$ = 1048577.
∴ All the paths can’t be tested.

(c) Also note that the complete path testing (CPT) (even if somehow carried out), can’t ensure that no errors will be present.
Ex: If the program is accidentally written in descending order (of execution) instead of ascending order, CPT is of no use.

(d) Exhaustive testing can’t detect missing paths.

15. **Every Design Error can’t be found**: We can never be sure that the provided specifications are entirely correct. Specification errors are one of the major reasons that produce faulty s/w.

Ex: User narrates his measurements with meters in his mind but the developer uses inches.

Finally, it can be stated that the domain of testing is infinite. It should be realised that importance must be shifted to effective testing instead of exhaustive testing.

Effective testing requires careful planning and is hard to implement. A proper boundary must be found between time, cost and energy spending.

16. Bugs caught at the starting stages of SDLC are more economical to debug than those at further stages.

17. **ST Terminology**: Terms like error, bug, failure, defect etc. are not synonymous and should be used carefully as the situation demands.

(a) **Failure**: It is the inability of a system/component to perform a required function according to its specification. It means results of a test are different from the expected ones. Failure depicts problems in a system on the output side.
(b) **Fault/Defect/Bug**: Fault is a condition that causes the system to produce a failure. Fault is synonymous with the words defect and bug.

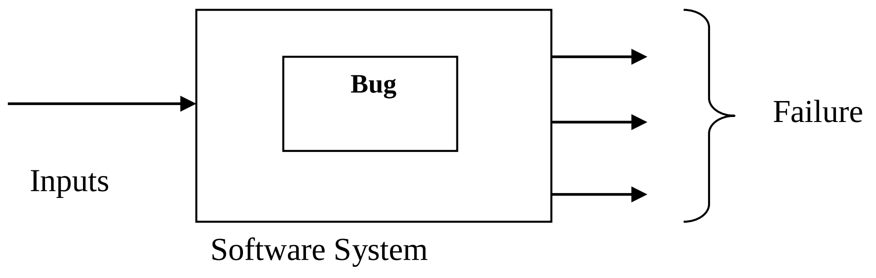

Fig : Testing Terminology

Fault is the reason embedded in any phase of SDLC and results in a failure.

It should also be noted that hidden bugs (unexecuted ones) may also lead to failures in future.

(c) **Error**: If mistakes are made at any phase of the SDLC, errors are produced. It is a term generally used for human mistakes.

Error → Bug → Failure(s) } **Fig: Flow of Faults**

```
Module A ( )
{
  .....
  While (a > n + 1);
  {
    ....
    Printf (“The value of x is:”, x);
  }
  ....
}
```

Expected Output: Value of x is printed.
Obtained Output: Value of x is not printed; may get warning/error.

This is a **failure** of the program.
Reason: The ‘while’ loop is not executed due to the presence of a semi-colon at the end of the statement. This is an **error**.

(d) **Test Case**: It is a well-documented procedure designed to test the functionality of a feature in the system. A test case (TC) has an identity and is associated with program behaviour.

Test case ID
Purpose
Preconditions
Inputs
Expected Outputs

Fig : Test Case Template

- **Test case ID**: Identification Number given to each test case.
- **Purpose**: Defines why the TC is being designed.
- **Preconditions**: Defining the inputs that may be used.

- **Inputs**: They should be real, instead of typical.
- **Expected Outputs**: Outputs that should be produced when there is no failure.

(e) **Testware**: These are the documents created during testing activities and those that a test engineer produces. They include test plans, test specifications, test case design, test reports etc. All the Testware should be managed and updated like a software product.

(f) **Incident**: Incident is a symptom that indicates a failure has taken place or is going to take place. It alerts a user about the future coming failure.

(g) **Test Oracle**: A means to judge the success/failure of a test i.e., the correctness of a test. Simplest oracle compares actual results and expected results by hand. To save time, they have also been automated.

18. Life Cycle of a Bug:

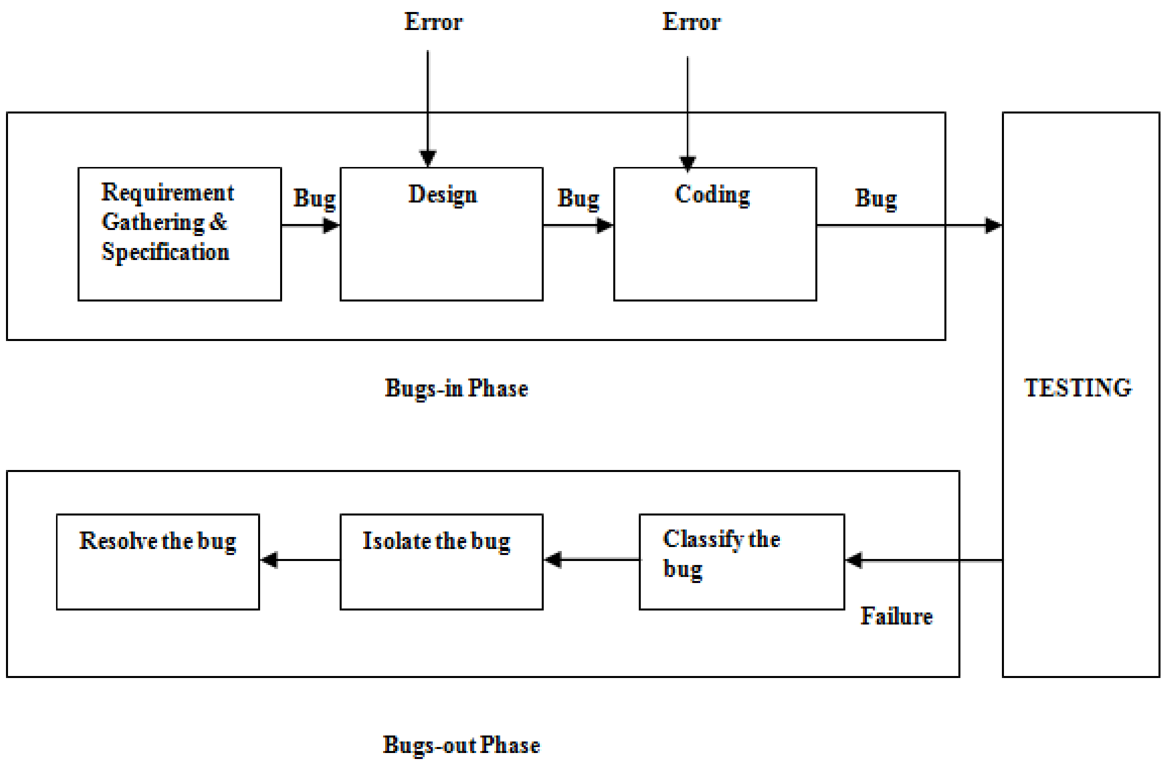

Fig : Life cycle of a Bug

If an error has been produced in a phase of the SDLC and is not detected, it results in a bug in the next phase.

19. **Bugs-in phase**: In this phase, errors and bugs are introduced in the software. A mistake creates error(s) and if this error goes unnoticed, it will cause failures and ends up as a bug in the software. Still yet, this bug is carried on to the next phase of the SDLC making it difficult to be unearthed and resolved.

NOTE: A phase may have its own errors as well as bugs received from previous phases.

20. **Bugs-out Phase**: If failures are noticed, it may be concluded that bugs are around. Note that some bugs may be present already but don't lead to failures as of now. In this phase, failures are observed and the following activities are performed to get rid of the bugs.

- **Bug Classification**: A failure is observed and bugs are classified depending on their nature. Ex: Critical, catastrophic etc.
  But a tester may not have sufficient time to classify each and every bug. At the same time, classification of bugs helps in the fact that serious bugs are first removed and trivial bugs may be dealt with later.
- **Bug Isolation**: Here the exact (module) location of the bug can be found out. Through the symptoms observed, the concerned function/module can be pin-pointed; this is known as the isolation of the bug.
- **Bug Resolution**: After isolation, the design can be back-traced (reverse engineering) to locate the bug and resolve it.

21. **States of a Bug**:

(1) New: Reported first time by a tester.

(2) Open: Note that 'new' state doesn't verify the genuineness (authenticity) of the bug. If the test team leader approves that the bug is genuine, it becomes an 'open' bug.

(3) Assign: Here, the developers' team checks the validity of the bug. If it is valid, a developer is assigned the job of fixing the bug.

(4) Deferred: The developer checks the validity and priority of the bug. If the bug is not much important or time (for release into the market) is less, the bug is 'deferred' and can be expected to be fixed in the next version/patch.

(5) Rejected: If the developer considers that the bug is NOT genuine, he may 'reject' it.

(6) Test: After fixing the (valid) bug, the developer sends it back to the testing team for the next round of checking. Now the bug's state becomes 'test'. (Fixed but not yet retested).

(7) Verified/Fixed: The tester tests the software to verify whether the bug is (fully) fixed or not. If all is well, the status is now 'verified'.

(8) Reopened: If the bug still exists even after fixing it, the tester changes its state to 'reopen' and the bug goes through the whole life cycle again. [A bug closed earlier may also be reopened if it makes its appearance again].

(9) Closed: Once the testing team is sure that the bug is fully fixed and eliminated, its status is set to be 'closed'.

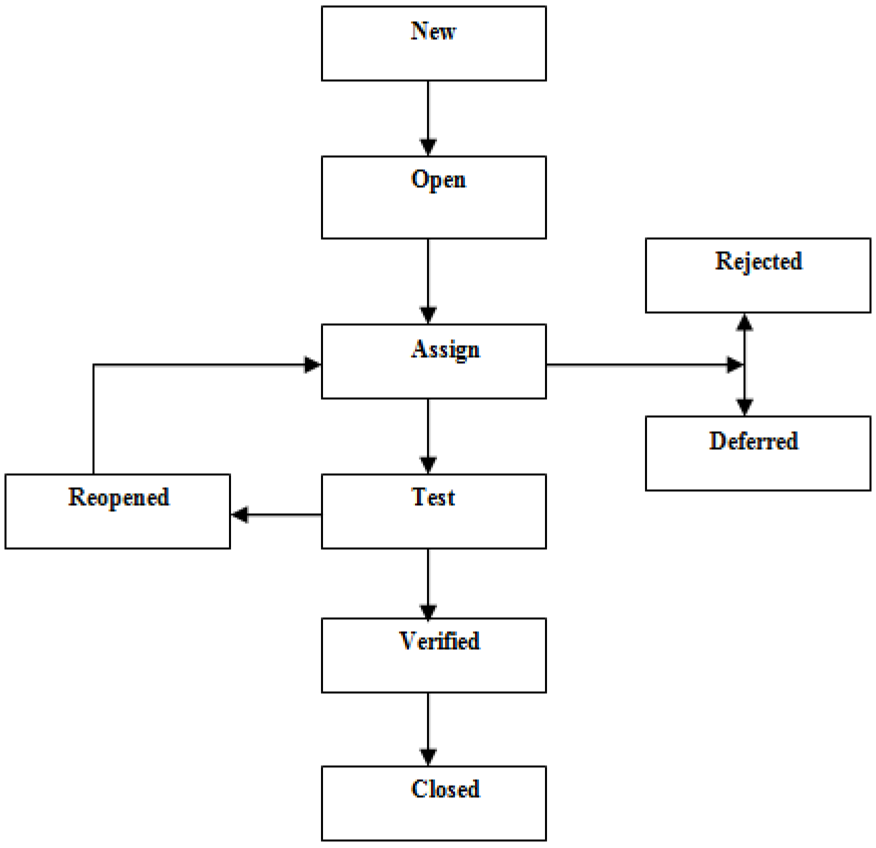

Fig : States of a Bug

22. **Reasons for Bug Occurrences**:
    (a) Human Errors
    (b) Bugs in Earlier States go Undetected: Ex: Miscommunication in gathering requirements, changing of requirements from time to time, changes in a phase affecting another, rescheduling of resources, complexity of maintaining the bug life cycle.

23. **Bugs affect Economics of ST**: It has been deduced that testing prior to coding is 50% effective in detecting errors; after coding it is 80% effective. Correction of bugs just before release of the product into market or after releasing (maintenance) is very costly.

    Cost of a bug = Detection cost + correction cost

     Cost of fixing a bug in the early stages (of SDLC) is lesser than that at later stages.

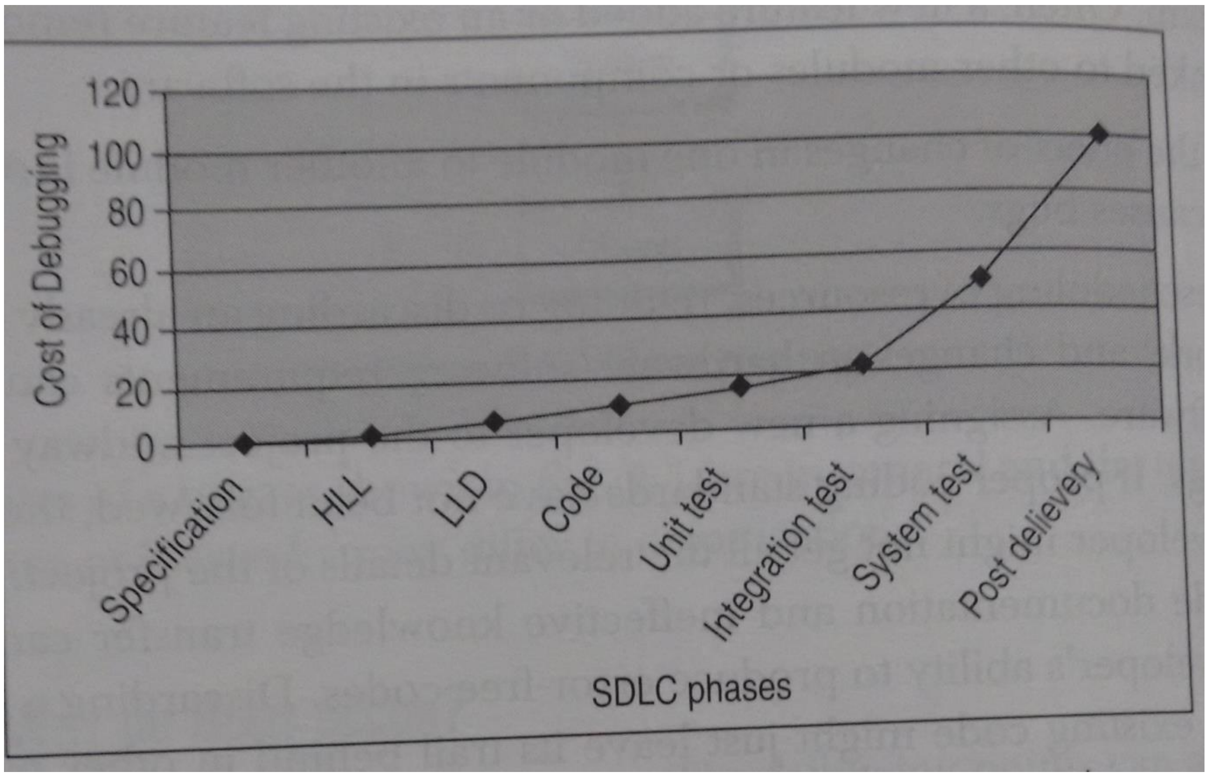

Fig: Cost of debugging increases if bug propagates

HLL => Higher Level Language; LLD=> Low Level Design (Design is a phase in SDLC)
Cost of debugging increases if the bug propagates through.

24. **Bug Classification** (based on criticality):
- **Critical Bug**: This type has the worst effect on the system, stops or hangs it completely. A user becomes helpless. Ex: In a mathematical program, the system might hang after taking in all input integers.
- **Major Bug**: It doesn’t stop the software to function but it might cause a part of it to fail or malfunction. Ex: The output is displayed, but it is wrong.
- **Medium Bug**: Medium bugs are less critical in nature as compared to critical or major bugs. Ex: A redundant or truncated output.
- **Minor Bug**: This type doesn’t affect the functionality of the software even if they occur. Ex: Typographical errors.

25. **Bug Classification based on SDLC**:

(1) Requirements and Specifications Bugs: These are the first type of bugs when they are observed WRT the SDLC phases. If undetected, the bugs of this first phase may pass-on to the future phases and become more difficult to locate and erase.

(2) Design Bugs: These are the bugs from the previous phase or those that came up here itself.
- Control flow bugs: Ex: Missing paths, unreachable paths etc.
- Logic bugs: Logical mistakes. Ex: Missing cases, wrong semantics, improper cases etc.
- Processing bugs: Arithmetic errors, incorrect conversions etc.

- Data flow bugs: Un-initialised data, data initialised in wrong format etc.
- Error Handling Bugs: Mishandling of errors, no error messages etc.
- Race Condition Bugs: Ex: Every time, the value of A is provided before B. Suddenly, if we give B before A, the software system crashes.
- Boundary related bugs: What happens to the program if the input is just less than minimum or just greater than maximum?
- User Interface bugs: Ex: Inappropriate functionality of some interfaces (wrong error messages), not matching user's expectations, causing confusion etc.

(3) **Coding Bugs**: Ex: Undeclared data, undeclared routines, dangling code (pointers pointing to an invalid address/virtual address), typographical errors etc.

(4) **Interface & Integration Bugs**: Invalid timing or sequence assumptions related to external signals, misunderstanding of external formats, inadequate protection against corrupt data, wrong sub-routine call sequence, misunderstood external parameter values etc.

(5) **System Bugs**: Invalid stress testing, compatibility errors, maximum memory limit etc.

(6) **Testing Bugs**: Testing mistakes like failure to notice problems, no fixes, no reports etc.

26. **Testing Principles**:

(1) Effective testing, not exhaustive testing: The tester's approach should be based on effective testing to adequately cover program logic and all conditions in the component level design.

(2) Testing is not a single phase in SDLC: Testing is not an activity in the SDLC. Testing starts as soon as specifications are prepared and continues till the product release into the market.

(3) Destructive approach for constructive testing: ST is a destructive process conducted in a constructive way. A tester should be wary of the presence of bugs and should proceed in a negative approach. The criterion to a successful test is to discover more bugs – not to show that bugs are not present.

(4) Early testing is best: Start testing as early as possible to discover bugs easily, reduce the cost of finding bugs and not making them difficult to discover.

(5) Probability of existence of an error in a section of a program is proportional to the no. of errors already found: Let module A has 50 discovered errors, module B 20 and module C 3. After a software update, all modules have come for testing again. Where should we be more careful or expect more errors to be found? Obviously module A. This can be stated in another way: error-prone sections should be dealt carefully since new bugs there are always expected.

(6) Testing strategy should start at the smallest module level and expand throughout the program. (Incremental testing) Testing must start at individual modules and integrate the modules for further testing.

(7) Testing should be performed by an independent team: Programmers may not have destructive approach; testers independently should take up the work.

(8) Everything must be recorded (filed) in ST.

(9) Invalid inputs and unexpected behaviour can find more errors (negative approach).

(10) Testers must participate in specification and design reviews

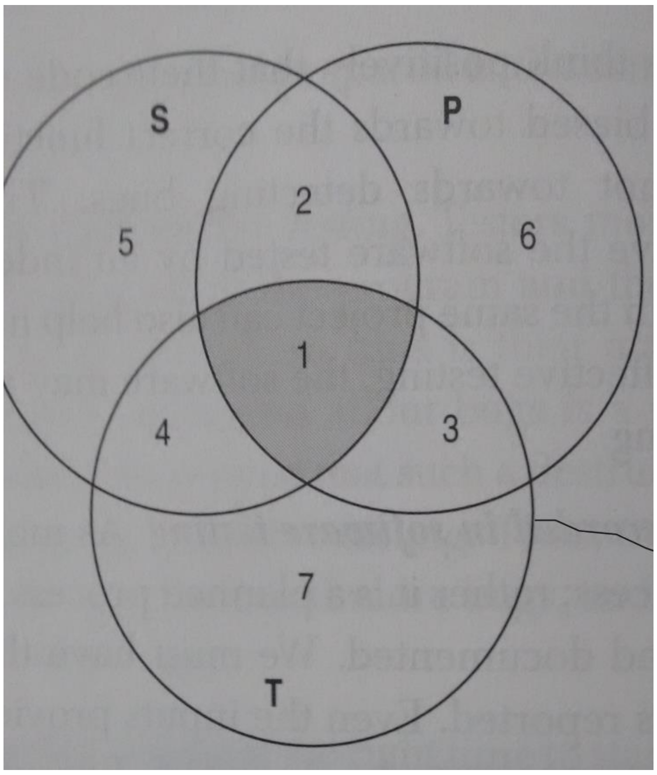

Fig : Venn diagram for S, P, T

S is the set of specified behaviour of the program
P is the implementation of the program
T is the set of test cases.

The good view of testing is to enlarge area of region 1. S, P and T must overlap each other such that all specifications are implemented and all of these are tested. This is possible only when the test team members participate in all discussions regarding specifications and design.

27. **Software Testing Life Cycle (STLC)**: Since testing has been identified as a process SDLC, there exists a need for well-defined series of steps to ensure effective testing.

The testing process divided into a precise sequence of steps is termed as ST life cycle (STLC). This methodology involves the testers at early stages of development, gives financial benefits and helps to reach significant milestones.

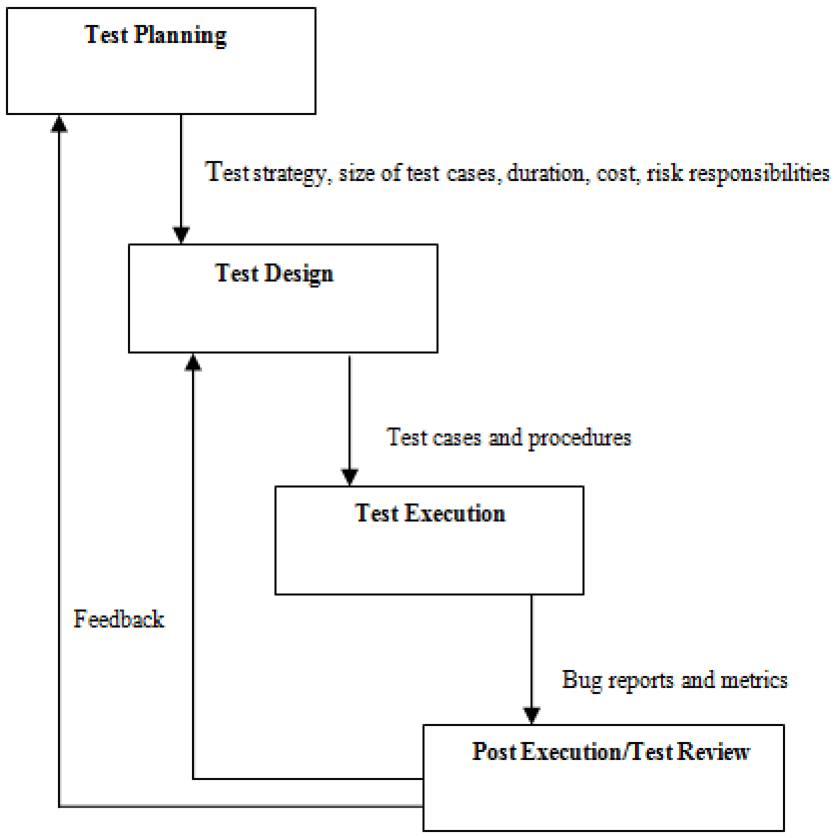
Test Planning
Test strategy, size of test cases, duration, cost, risk responsibilities
Test Design
Test cases and procedures
Test Execution
Feedback
Bug reports and metrics
Post Execution/Test Review

Fig : STLC

28. **Test Planning**: The goal of test planning is to consider important issues of testing strategy which are: resources, schedules, responsibilities, risks and priorities. The activities here are listed below:

- Define the test strategy.
- Estimate the number of test cases, their duration, and cost.
- Plan the resources like manpower, tools etc.
- Identify the risks.
- Define the test completion criteria.
- Identify the methodologies, and techniques for various test cases.
- Identify reporting procedures, bug classification, databases [for testing], bug severity levels and project metrics.

The output of test planning is the test planning document, which specifies about test case formats, test case formats, test cases at different phases of SDLC, different types of testing etc.

29. **Test Design**: The activities here are:

- Determining the test objectives and their prioritization: Test Objectives=> Elements that need to be tested to satisfy an objective. Reference material is needed to be gathered and design documentation is to be prepared. Depending on this material the experts prepare a list of objectives and prioritization [required] is also made up.
- Preparing list of items to be tested

- Mapping items to test cases => for this, a matrix is prepared using the items and test cases. This helps to identify what tests are to be carried out for the listed items, removing redundant test cases, identifying the absence of test cases for an error and designing them.
- **NOTE**: The rule in designing test cases is: cover all functions but do not make too many of them.

30. Some attributes of a 'good' test case are given below:
    - It is designed considering the importance of test risks, thus resulting in good prioritization.
    - It has high probability of finding an error.
    - It reduces redundancy, overlapping and wastage of time.
    - It covers all features of testing of the concerned software, but in a modular approach.
    - A 'successful' test case is one that has discovered a completely new and unknown error.
31. **Selection of TC design technique**: While designing the test cases, there are two categories: BBT and WBT. BBT concentrates on results only – not on what is going inside the program while WBT does the opposite.

32. **Creating Test Cases and Test Data**: Objectives of the test cases are identified and the type of testing (positive or negative) is also decided for the input specifications.

33. **Selecting the testing environment and supporting tools**: Details like hardware configurations, testers, interfaces, operating systems etc. are specified in this phase.

34. **Creating the procedure specification**: This is a description of how the test case will be run. This form of sequenced steps is used by a tester at the time of executed test cases.

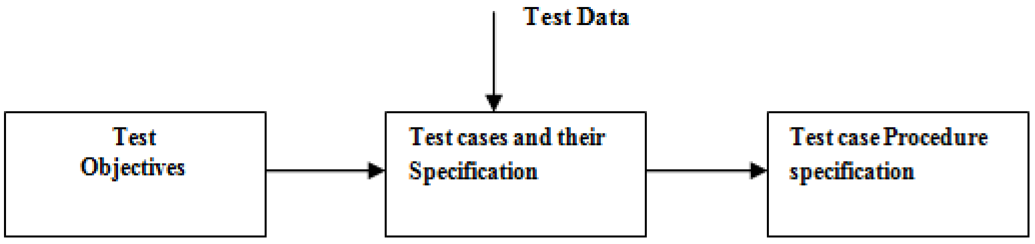
Test Data
Test Objectives
Test cases and their Specification
Test case Procedure specification

Fig : Test Case Design Steps

35. **Test Execution**: In this phase, all test cases are executed.
    - **Verification TCs:** Started at the end of each phase of SDLC.
    - **Validation TCs:** Started after the completion of a module.

Test results are documented as reports, logs and summary reports.

Test Execution Level	Responsibility
Unit	Developer
Integration	Testers & Developers
System	Testers, developers, users
Acceptance	Testers, end-users

Fig (Tab): Testing level Vs. Responsibility

36. **Test Review/Post-Execution**: This phase is to analyze bug related issues and obtain the feedback. As soon as the developer gets the bug-report, he performs the following activities:
   (a) Understanding the bug
   (b) Reproducing the bug: This is done to confirm the bug position and its whereabouts so as to avoid the failures.
   (c) Analyse the nature and cause of a bug.

   After these, the results from manual and automated testing can be collected and the following activities can be done: reliability analysis (are the reliability goals being met or not), coverage analysis and overall defect analysis.

37. **STM**: It is the organisation of ST through which the test strategy and test tactics are achieved as shown in the Figure given below.
38. **ST Strategy**: It is the planning of the whole testing process into a well-planned series of steps. Strategy provides a plan that includes specific activities that must be performed by the test team to achieve certain goals.
   (a) Test Factors: These are risk factors or issues related to the concerned system. These factors are needed to be selected and ranked according to a specific system under development. Testing process should reduce these test factors to a prescribed level.
   (b) Test Phase: It refers to the phases of SDLC where testing will be performed. Strategy of testing might change for different models of SDLC. (Ex: Waterfall, spiral etc.)

39. **Test Strategy Matrix**: A test strategy matrix identifies the concerns that will become the focus of test planning and execution. It is an input to develop the testing strategy.

Test Factors	Test Phase					
	Requirements	Design	Code	Unit Test	Integration Test	System Test
1. (most imp.)						
2. (next imp)						
3. (next imp.)						

Fig.: Test Strategy Matrix

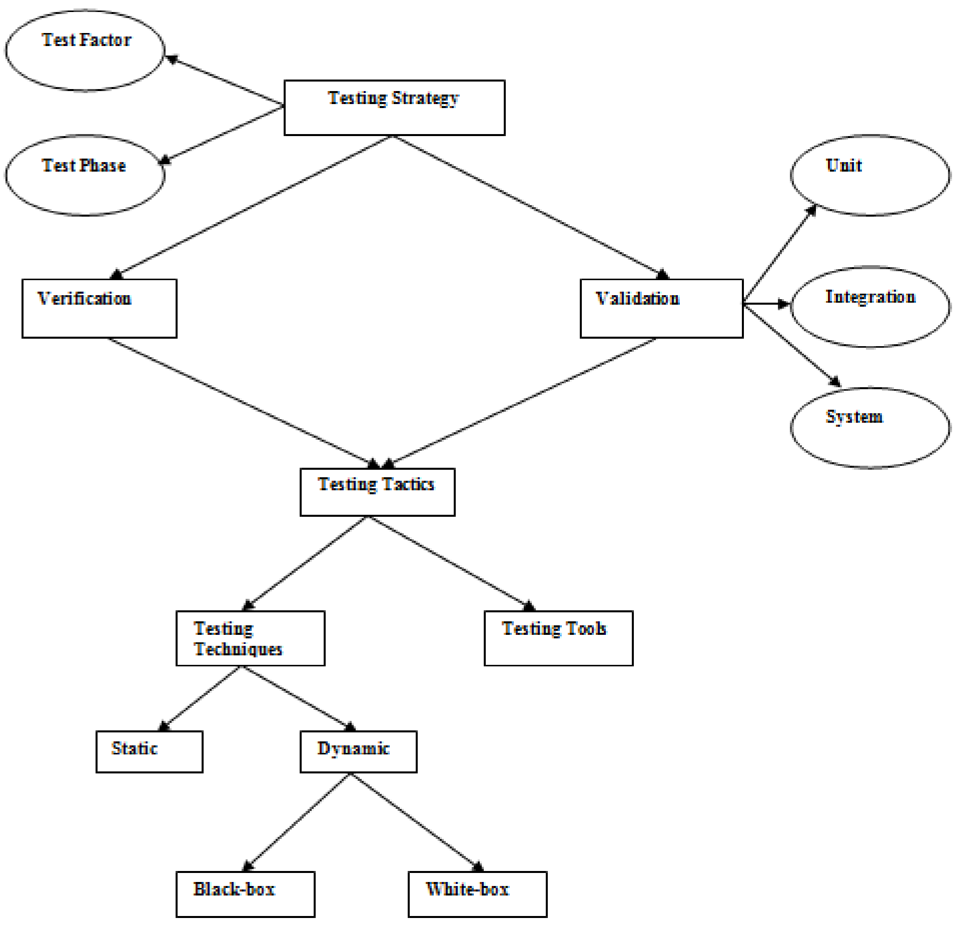

Fig: Testing Methodology

40. Steps to prepare the TS Matrix:

- Select and rank test factors (rows of the matrix)
- Identify system development phases (columns)
- Identify risks associated with the system: The purpose is to identify the problems that need to be solved under a test phase. Ex: Events, actions, or circumstances that may prevent the test program being implemented or executed according to a schedule. The concerns should be expressed as questions.

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? MIPS => (Microprocessor without interlocked pipeline stages)
Service Level	Is time frame for booting mentioned?	Is time frame included in the design of the module				

Fig : Example Test Strategy Matrix

41. **Creating a Test Strategy**: As shown in Fig given above, a project of designing a new OS is taken as an example. The steps to be used are:
    - Select and rank test factors: Portability is an important factor to be checked out for an OS. This factor is most important since an OS has to be compatible for different kinds of hardware configurations.
    - Identify the Test Phases: In this step, all test phases affected by the selected test factors are identified; these can be seen in Fig given above.
    - Identify the Risks: Risks are basic concerns associated with each factor in a phase and are expressed in the form of questions like: “Is testing performing successfully on INTEL, MIPS, and AMD H/W platforms?”
    - Plan the test strategy for every risk identified: After identifying the risks, a plan strategy to tackle them is to be developed so that risks are mitigated.

42. **Development of Test Strategy**: Test strategy should be such that testing starts at the very first phase of SDLC and continues till the end. Terms here are:

    (a) Verification: The purpose here is to check the software at every development phase of SDLC in such a way that any defect can be detected at early stages and stopped. Verification can be stated as a set of activities that ensures correct implementation of functions in software.

    (b) Validation: It is used to test the software as a whole in accordance with the user’s expectations/specifications.

Barry Boehm says:

Verification means ‘are we building the product right?’
Validation means ‘are we building the right product?’

In another way,
Verification checks are we working in the right way and
Validation ensures that the required goals have been achieved.

43. **Testing Life Cycle Model**: The formation of test strategy based on the two terms verification and validation. Life cycle involves continuous testing of the system during the development process (SDLC). At predetermined points, the results of the development process are inspected to determine the correctness of implementation. They also identify the errors as early as possible.

(a) V-Testing Life Cycle Model: Dev. Team and testing team start their work at the start. If there are risks, the tester develops a process to minimize or eliminate them.

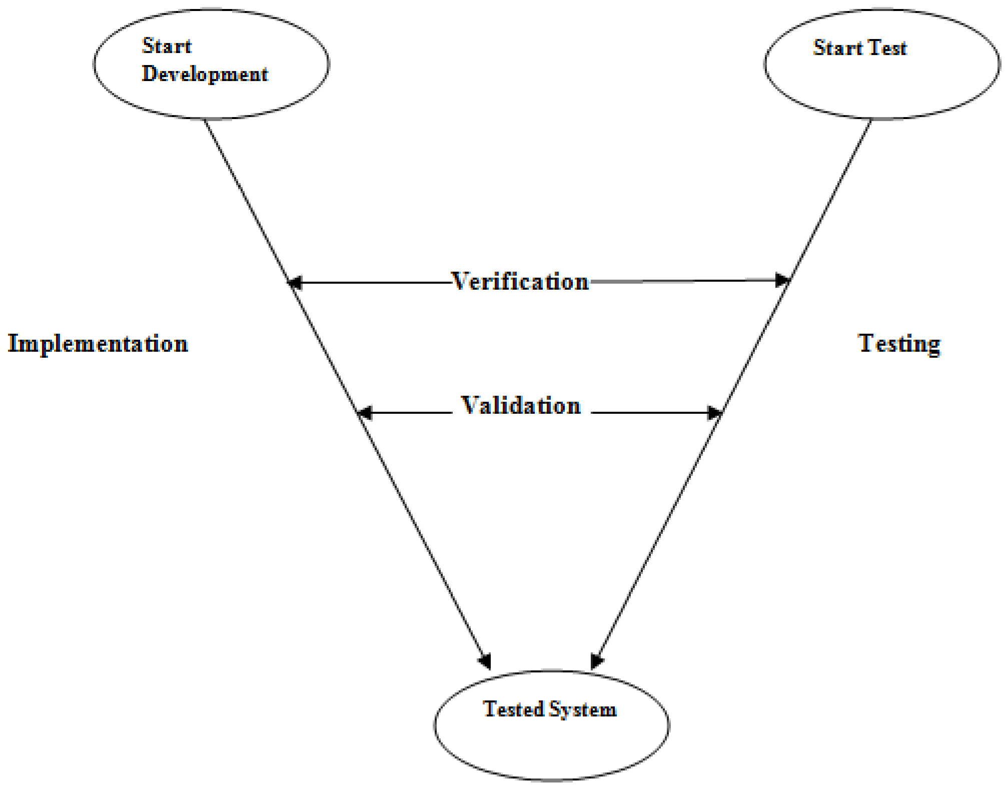

Fig : V-testing Model

It should be noted that the Figure given above can be expanded and elaborated further.

(b) Expanded V-Testing Model:

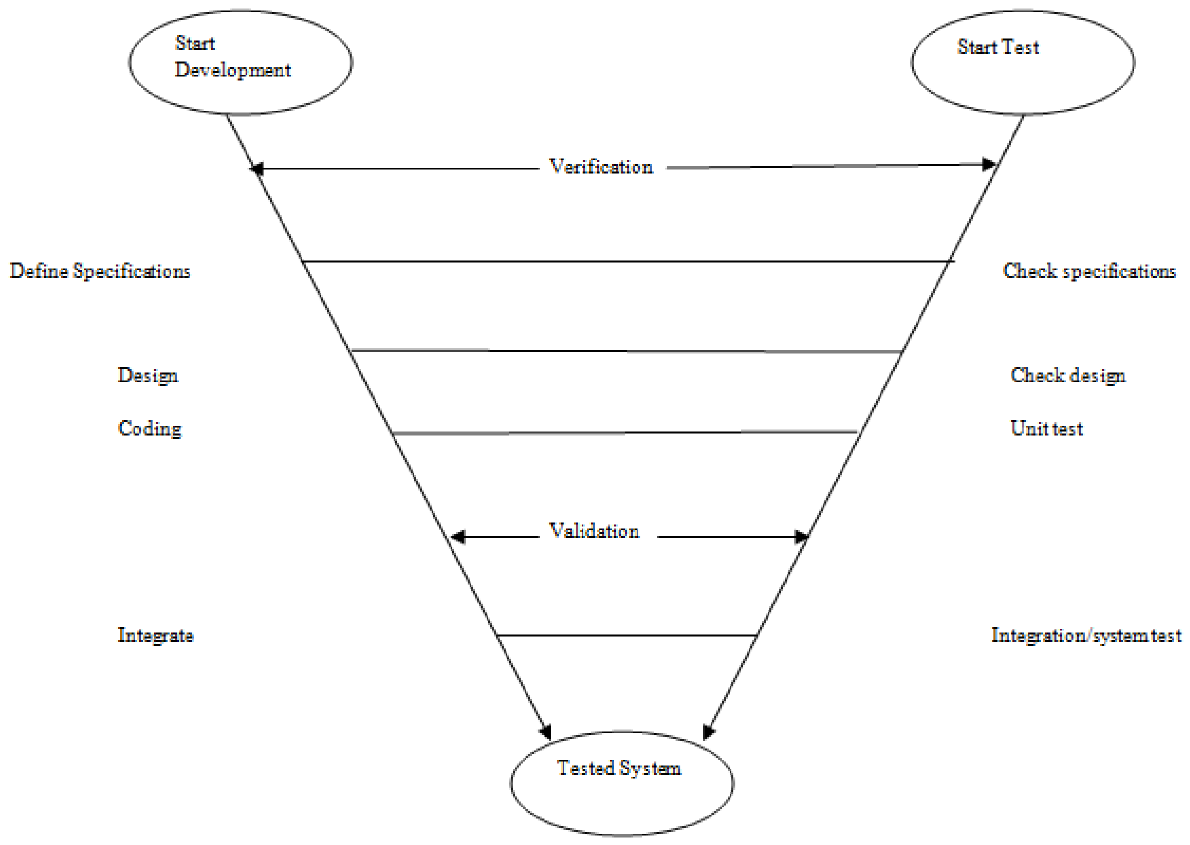

Fig: Expanded V-testing Model

44. The V & V process involves (i) verification of every step of SDLC and (ii) validation of the verified system at the end.

45. **Validation Activities**: It has three known activities:
    - **Unit Testing**: It is a validation effort performed on the smallest module of the system. It confirms the behaviour of a single module according to its functional specifications.
    - **Integration Testing**: It combines all unit tested modules and performs a test on their aggregation (interfaces between modules). Common factors among the modules, DS, messages etc. are given more preference.
    - **System Testing**: This level concentrates on testing the entire integrated system. The purpose is to test the validity for specific users and environments. The validity of the whole system is checked against the specifications of requirements.

46. **Testing Tactics**:
    - **Software Testing Techniques:** At this stage, testing can be defined as the design of effective test cases where most of the testing domains will be covered detecting the maximum number of bugs. The technique used to design effective test case(s) is called ST techniques.

- **Static Testing:** It is the technique for assessing the structural characteristics of source, design specifications or any notational representation that conforms to well-defined syntactic rules. Code is never executed in this methodology; they are only examined.
- **Dynamic Testing:** All the methods that execute the code to test software are known as dynamic testing techniques.
  - **BBT:** This takes care of the inputs given to a system and the output is received after processing in the system. BBT is not concerned with what is happening inside the program and the methods being used. It only checks the functionality of the program (functional testing).
  - **WBT:** Every design feature is checked logically and all possible paths are executed with different inputs. It is also called structural testing since it is concerned with the program structure.

47. **Testing Tools**: TTs provide the option to automate the selected testing technique with the help of tools. A tool is a resource for performing a test process. A tester should understand the automation process and its usage before actually utilising it.

48. **Considerations in developing testing methodologies**:

(a) Determine Project risks: A test strategy is developed with the help of another team familiar with the business risks associated with the software.
(b) Determine the type of development project: The environment or methodology to develop software also affects the testing risks. Note that risks associated with a new project will be different from that of purchased software.
(c) Identify test activities according to SDLC Phase
(d) Build the Test plan: A test plan provides environment and pre-test background, objectives, test team, schedule and budget, resource requirements, testing materials (documentation, software, inputs, test doc and results), functional requirements and structural functions and type of testing technique that is to be used.

Course (Unit-1) Objectives:

- Introduce the student to the emphasis to be given for software testing
- Provide understandability on the evolvement of software testing from the part of a phase to a total process.
- Offer the student the real process of types of testing (positive and negative).
- Make the student to understand the parallelism between development of software and its testing.
- Make clear the pros and cons of effective testing and exhaustive testing.
- Increase the student's capacity to write P/N test cases manually.

****