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 1.1 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**:

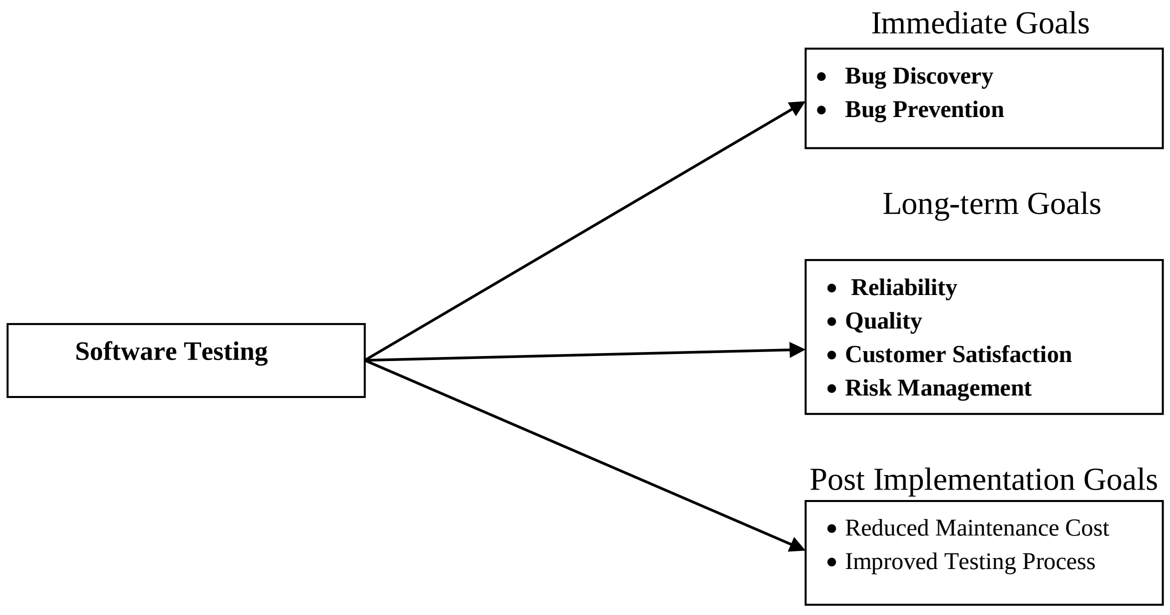


Fig 1.1: Software Testing Goals

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.

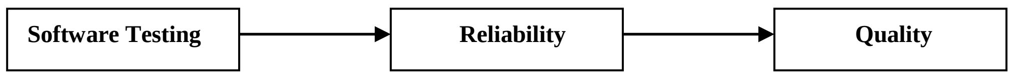


Fig 1.2: Testing produces reliability and quality

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

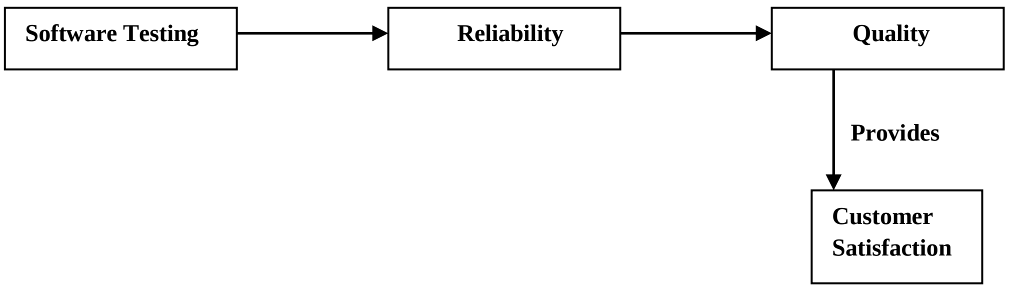


Fig 1.3: Quality leads to customer Satisfaction

- **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.

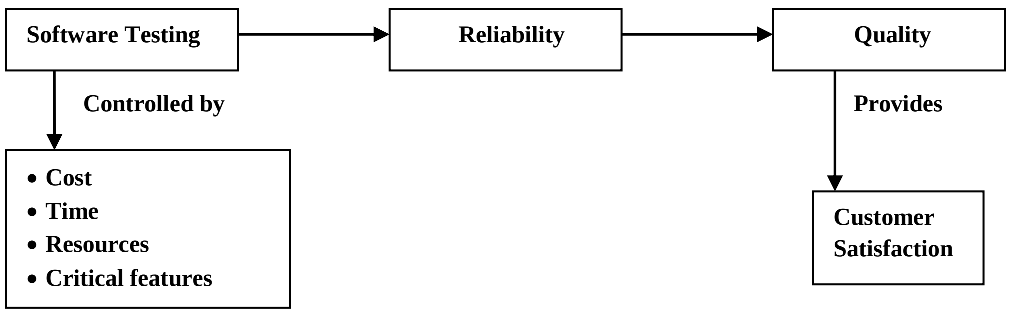


Fig 1.4: Testing controlled by risk factors

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**:

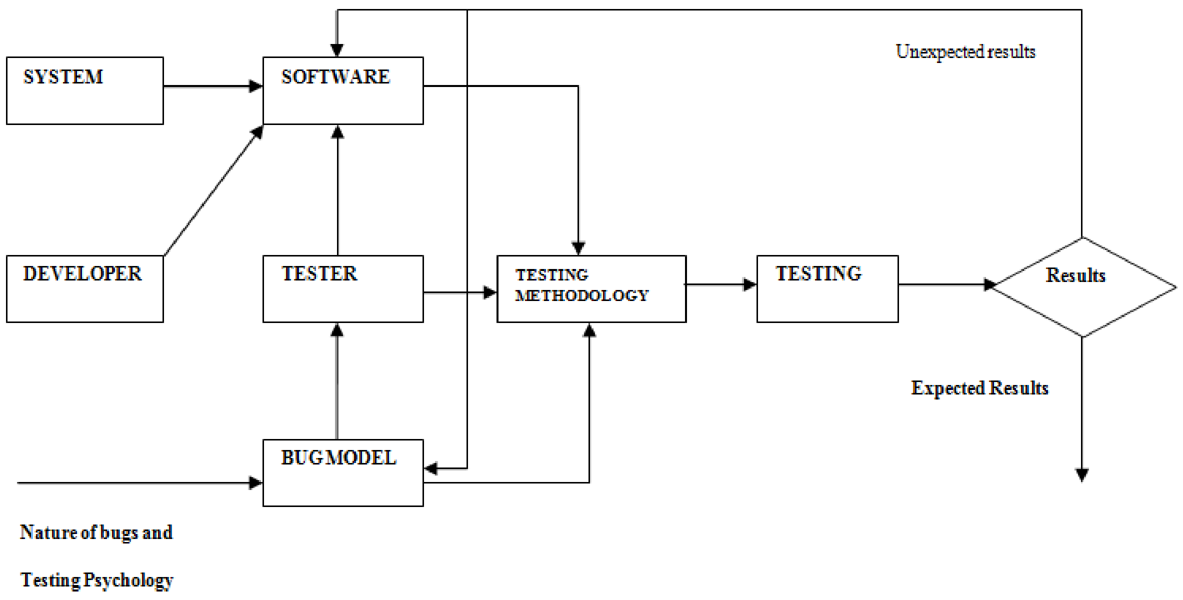


Fig 1.5: Software Testing Model

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
```

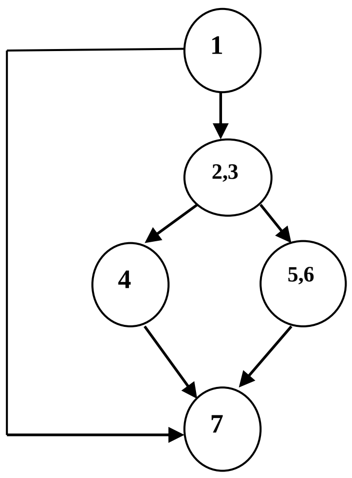


Fig 1.6: Flowchart

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.

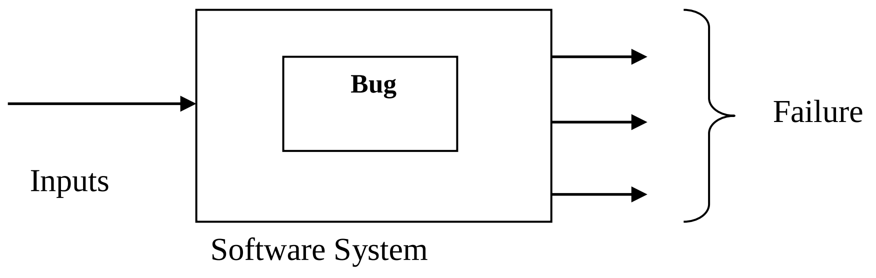


Fig 1.7: Testing Terminology

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 $\rightarrow$ Bug $\rightarrow$ Failure(s) } **Fig 1.8: 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 1.9: 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:

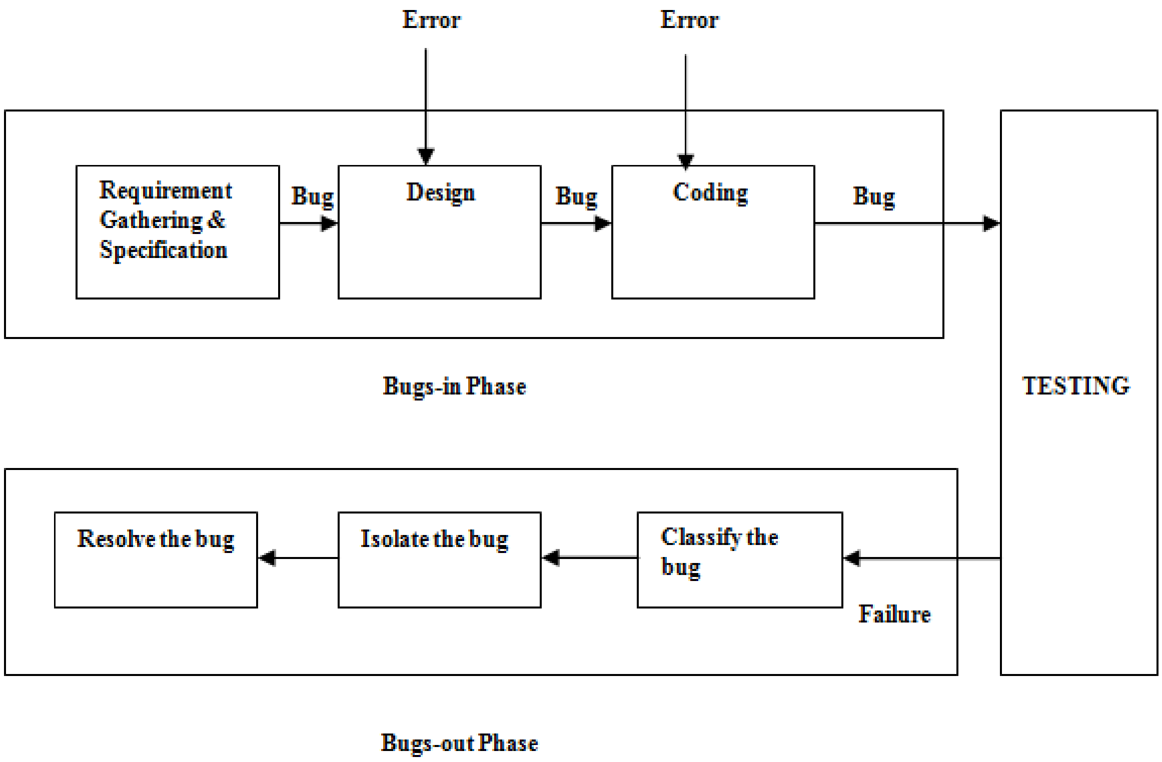


Fig 1.10: Life cycle of a Bug

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'.

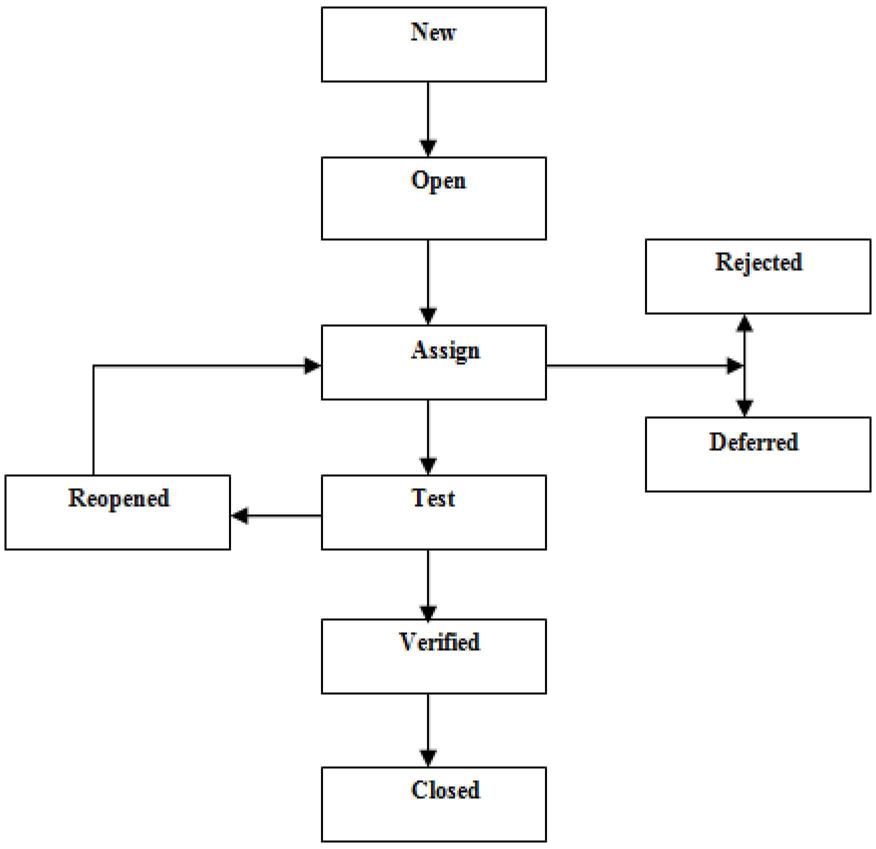


Fig 1.11: States of a Bug

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.

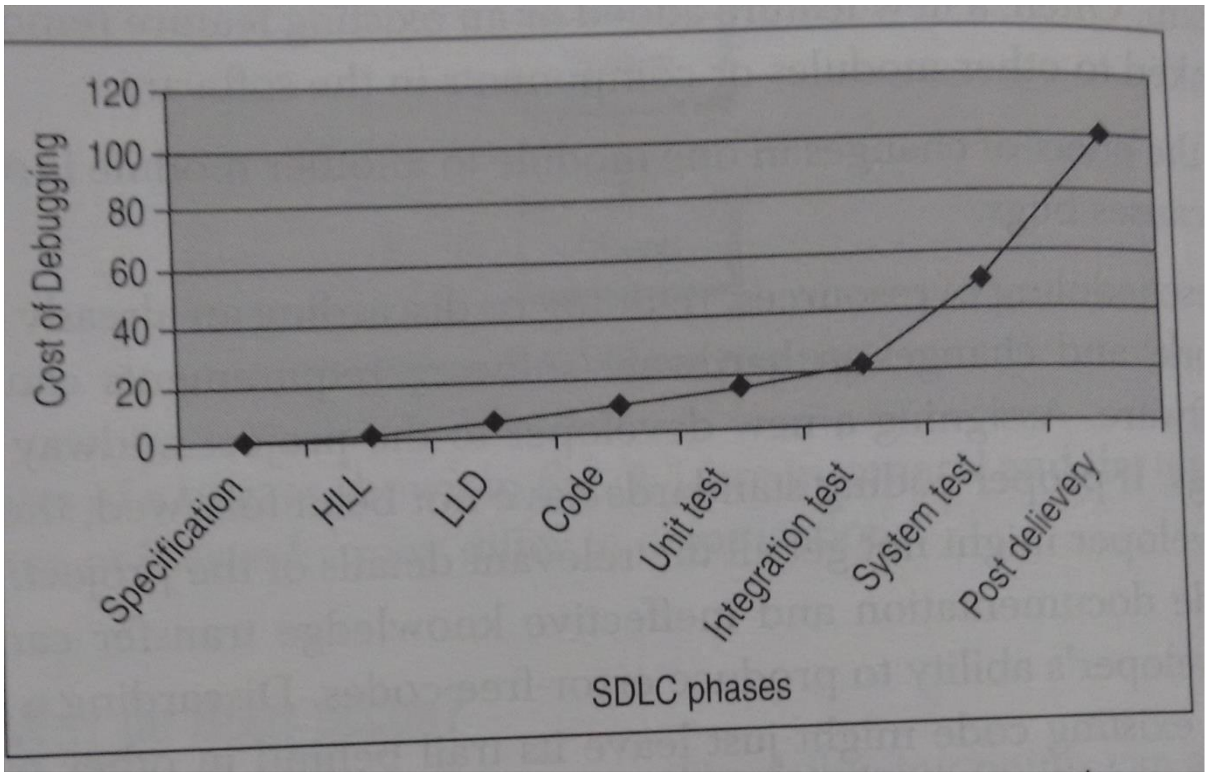


Fig 1.12: Cost of debugging increases if bug propagates

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

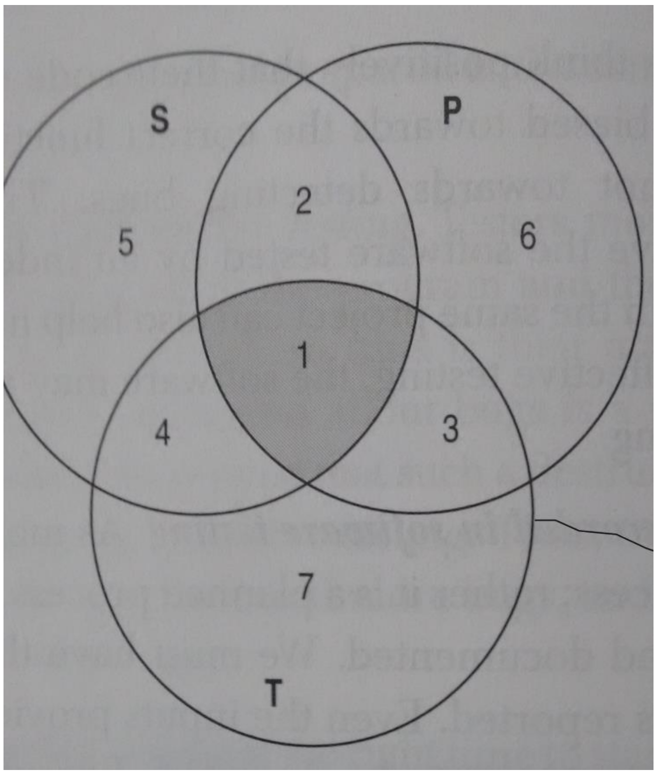


Fig 1.13: Venn diagram for S, P, T

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.

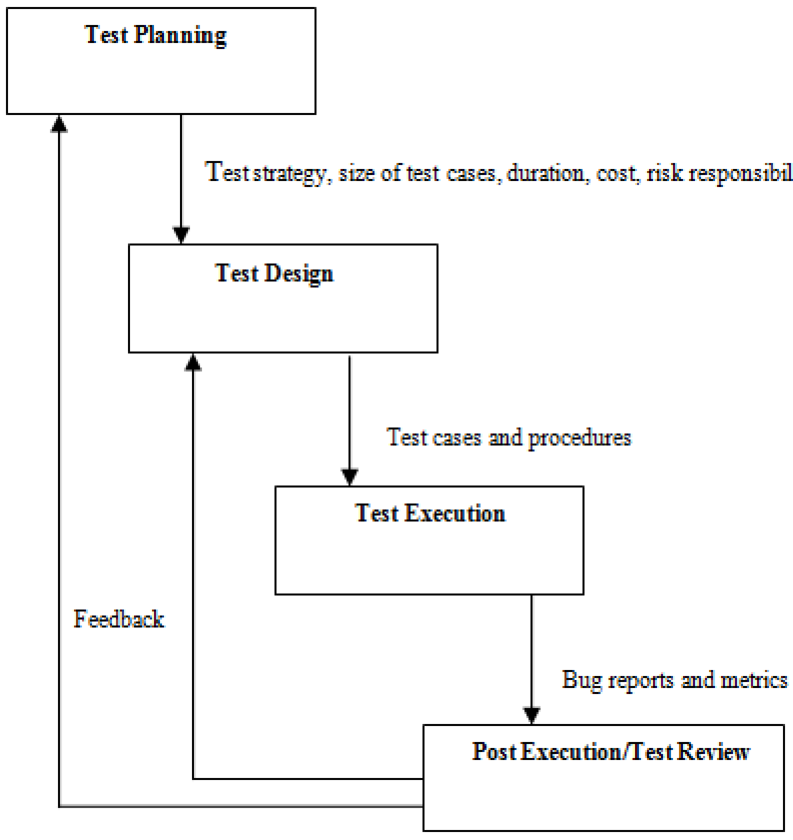

Test Planning
Test strategy, size of test cases, duration, cost, risk responsibilities
Test Design
Test cases and procedures
Test Execution
Feedback
Bug reports and metrics
Post Execution/Test Review

Fig 1.14: 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.

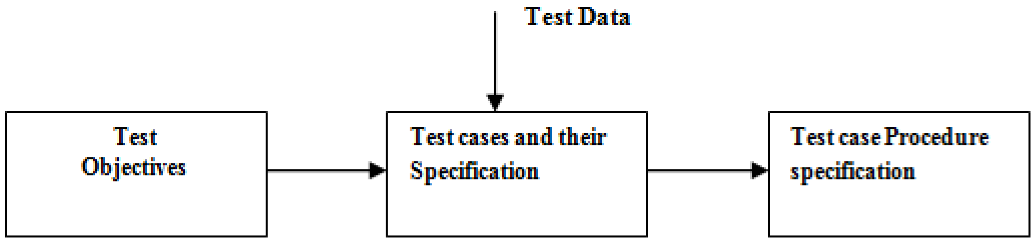

Test Data
Test Objectives
Test cases and their Specification
Test case Procedure specification

Fig 1.15: 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) 1.16: 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 1.18.
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. 1.17: Test Strategy Matrix

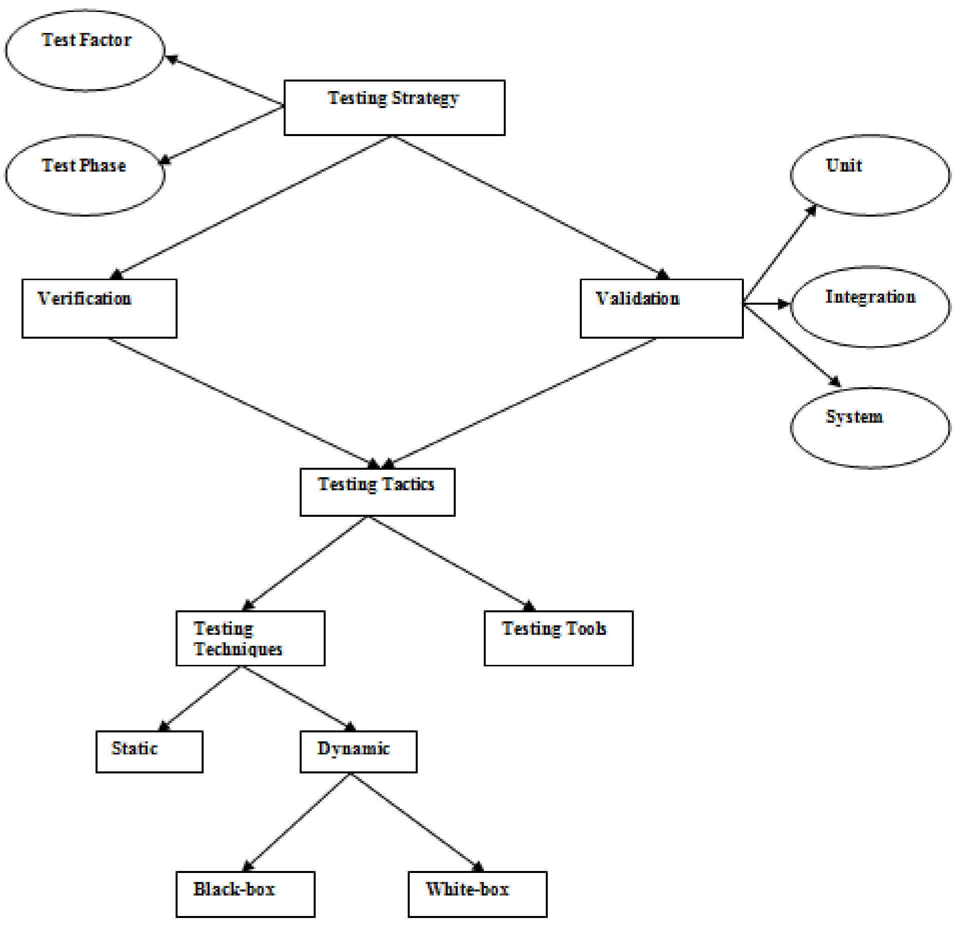


Fig 1.18: Testing Methodology

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 1.19: Example Test Strategy Matrix

41. **Creating a Test Strategy**: As shown in Fig 1.19, 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 1.19.
    - 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.

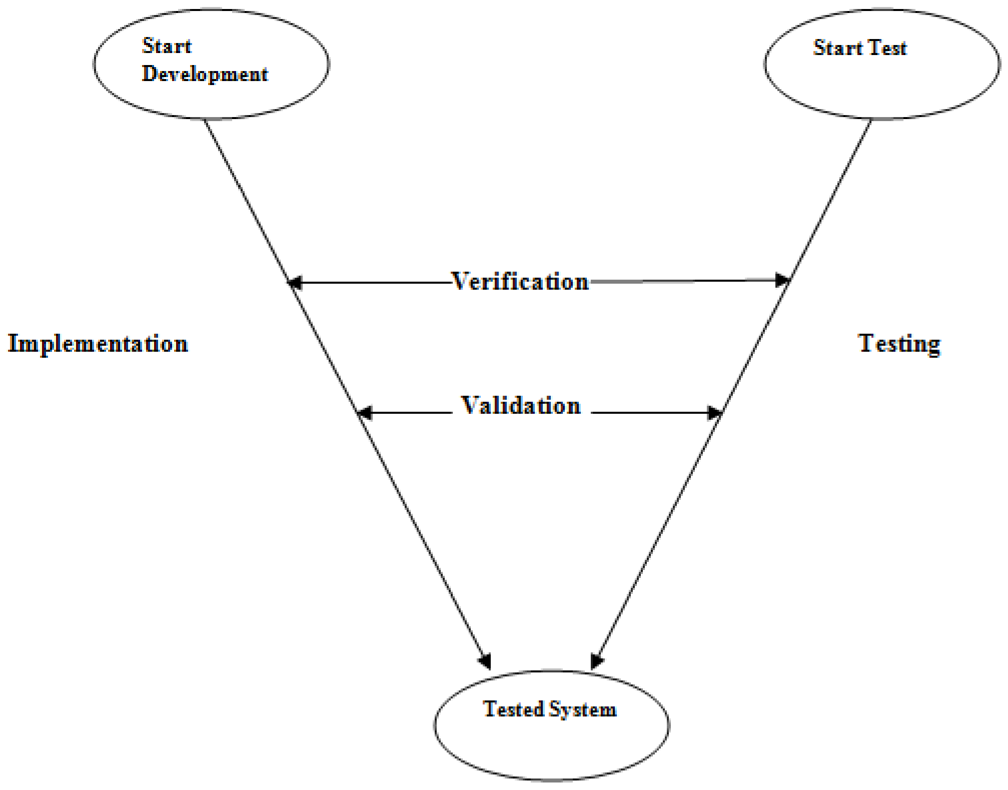


Fig 1.20: V-testing Model

It should be noted that Fig 1.20 can be expanded and elaborated further.

(b) Expanded V-Testing Model:

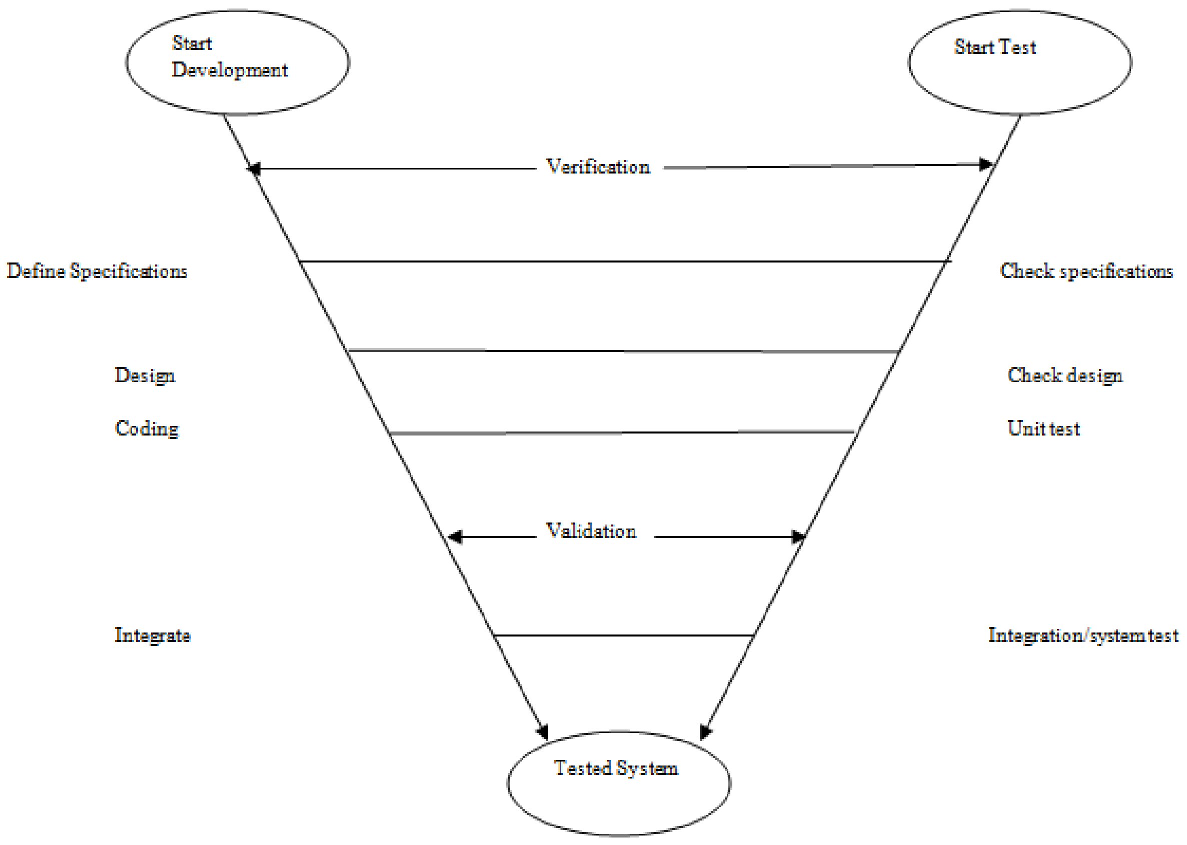


Fig 1.21: Expanded V-testing Model

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.

****

Software Testing Methodologies (R13)

UNIT – 1

Quiz

1. Discovery of errors in software is a _____ goal of Testing. [KEY: B]
   (a) Long-term
   (b) Short-term/Immediate
   (c) Post-Implementation
   (d) All

2. ST produces ______ [KEY: D]
   (a) Reliability
   (b) Quality
   (c) Customer Satisfaction
   (d) All

3. In software companies, ST starts at ___ phase of SDLC. [KEY: B]
   (a) Analysis
   (b) Requirements
   (c) Coding
   (d) Testing

4. ____ is the probability that undesirable events will occur in a system. [KEY: C]
   (a) Bug
   (b) Error
   (c) Risk
   (d) Fault

5. Time and cost are ____ factors. [KEY: D]
   (a) Customer Satisfaction
   (b) Reliability
   (c) Quality
   (d) Risk

6. Testing is a process of executing a program to show that ___. [KEY:]
   (a) It is Error-free
   (b) Errors are found
   (c) Customer satisfaction is good
   (d) All

7. The timing variation in two or more inputs, if reversed/changed after a long time may cause the program to crash. This is called ___. [KEY:C]

(a) Valid Inputs
(b) Invalid Inputs
(c) Race condition Inputs
(d) Edited Inputs

8. The domain of testing is ____. [KEY: D]
(a) Practical
(b) Finite
(c) General
(d) Infinite

9. Software testing is a branch of ____. [KEY:C]
(a) STLC
(b) Quality
(c) SQA
(d) SDLC

10. ____ is the condition that in actual causes a system to produce a failure. [KEY:D]
(a) Defect
(b) Bug
(c) Fault
(d) All

11. A well-documented procedure to test the functionality of a feature in the system is known as ____. [KEY:D]
(a) Test Plan
(b) Test Cycle
(c) Test Life Cycle
(d) Test Case

12. When the test leader approves that the bug is genuine, its state becomes ___. [KEY:B]
(a) New
(b) Open
(c) Test
(d) Assign

13. Bugs that do not affect the functionality of the software are called ___ bugs. [KEY:B]
(a) Normal
(b) Minor
(c) Medium
(d) General

14. If the user is not satisfied while using the software, then there are ___ bugs. [KEY:C]
    (a) Module
    (b) Interface
    (c) User Interface
    (d) Coding

15. The testing process divided into a well-defined sequence of steps is called ___. [KEY:A]
    (a) Software testing life cycle
    (b) Software bugs life cycle
    (c) Test plan
    (d) Test case

16. Dynamic testing with code consideration is known as ___. [KEY:B]
    (a) Gray box testing
    (b) White box testing
    (c) Black box testing
    (d) Code review

17. ___ identifies the concerns that will become the focus of test planning and execution. [KEY: D]

    (a) Matrix
    (b) Strategy matrix
    (c) Graph matrix
    (d) Test strategy matrix

18. The lowest level of testing is called___ testing. [KEY:A]
    (a) Unit
    (b) Integration
    (c) Line
    (d) Module

UNIT – 2

Course (Unit-2) Objectives:

- Get the student acquainted with the verification and validation activities
- Application of verification at different levels of software development life cycle
- Understand the process of validation of code and testing methodologies
- Use the BVA process to make the student realize the importance of border-level values
- Make the student to understand and apply the other processes of state table based testing, decision table based testing, cause-effect graph testing and error guessing

1. **Verification and Validation**:

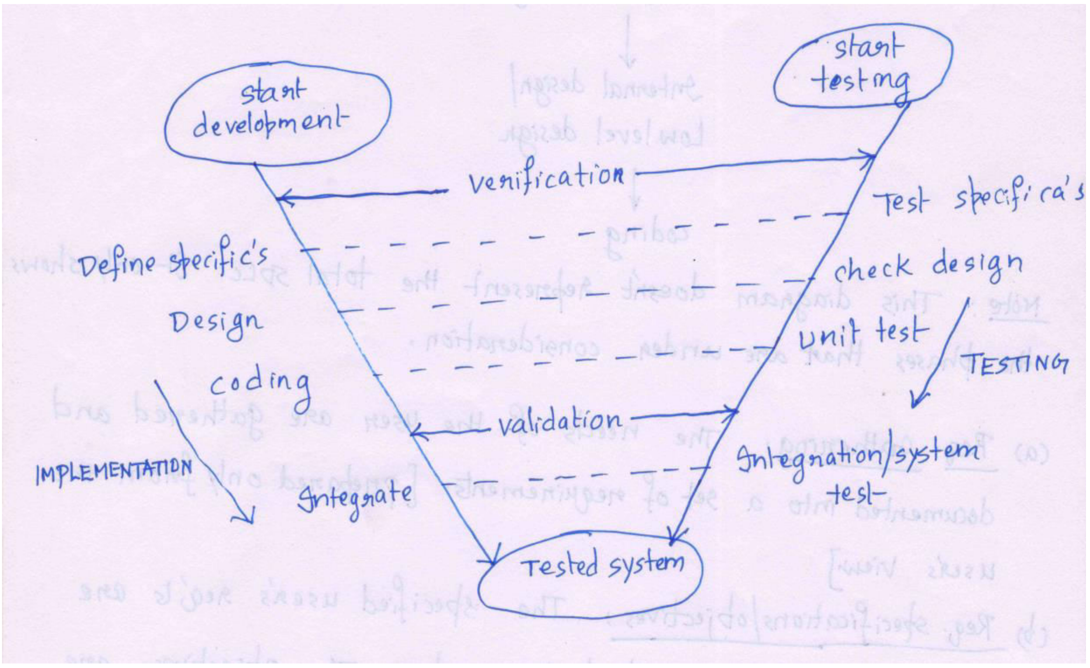


Fig 2.1: V-Testing Model

A V-diagram provides the following insights about ST:

(a) Testing can be implemented in the same flow as for SDLC.
(b) Testing can be broadly planned as verification and validation.
(c) Testing must be performed at every stage of SDLC.
(d) V-diagram supports the concept of early testing.
(e) V-diagram also supports parallel activities of developers and testers.
(f) More concentration on V-V process results in producing effective software.
(g) Testers should be involved in the development process.

2. **Verification and Validation (V & V) Activities**: The SDLC phases are given in the diagram below:

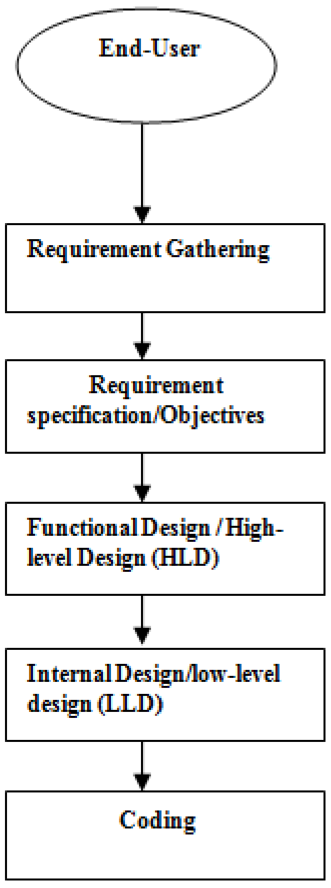

End-User
Requirement Gathering
Requirement specification/Objectives
Functional Design / High-level Design (HLD)
Internal Design/low-level design (LLD)
Coding

Fig 2.2 SDLC Phases

3. **Requirements Gathering**: The needs of the user are gathered and documented into a set of requirements. Note that these are prepared from the user's viewpoint.

   Requirement Specification or Objectives: The specified users' requirements are translated into the developer's terminology and the SRS are specified.

   Functional Design or HLD: Functional design is the process of translating user requirements into a set of external interfaces. It contains architecture diagrams, functionalities of the system, list of modules, functionality of each module and interface relationships and database tables. [Macro-level]

   Internal Design or LLD: A HLD document can't be used for coding by the developers. So, the analysts prepare a micro-level document called internal design or LLD that describes every module in a detailed manner.

   Coding: Based on LLD, coding is done by the developers.

4. **Verification**:

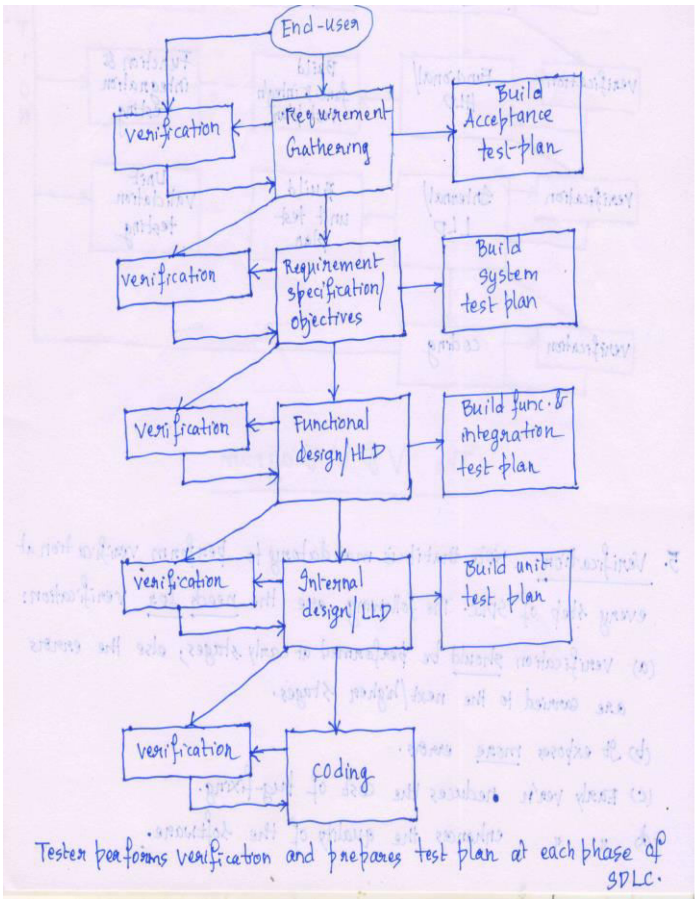


Fig 2.3 Tester prepares verification & test plan during each SDLC Phase

5. When coding is completed for a unit or system, and parallel verification activities have been performed, the system can be validated. The validation activities are executed with the help of test plans. This brings up the complete V&V activities diagram. [Fig 2.4]

6. **Verification**: Under the V&V process, it is mandatory that verification is performed at every step of SDLC. The following are the needs for verification:

   (a) Verification should be performed at early stages (of SDLC); else the errors are carried on to the next phases.
   (b) Verification exposes more errors.
   (c) Early verification decreases the cost of fixing bugs.
   (d) Early verification enhances the quality of the software.

7. **Goals of Verification**:
   - 'Everything' must be verified.
   - Results of verification may not be binary – not just accept/reject. A phase might be accepted with minor errors which can be corrected later.

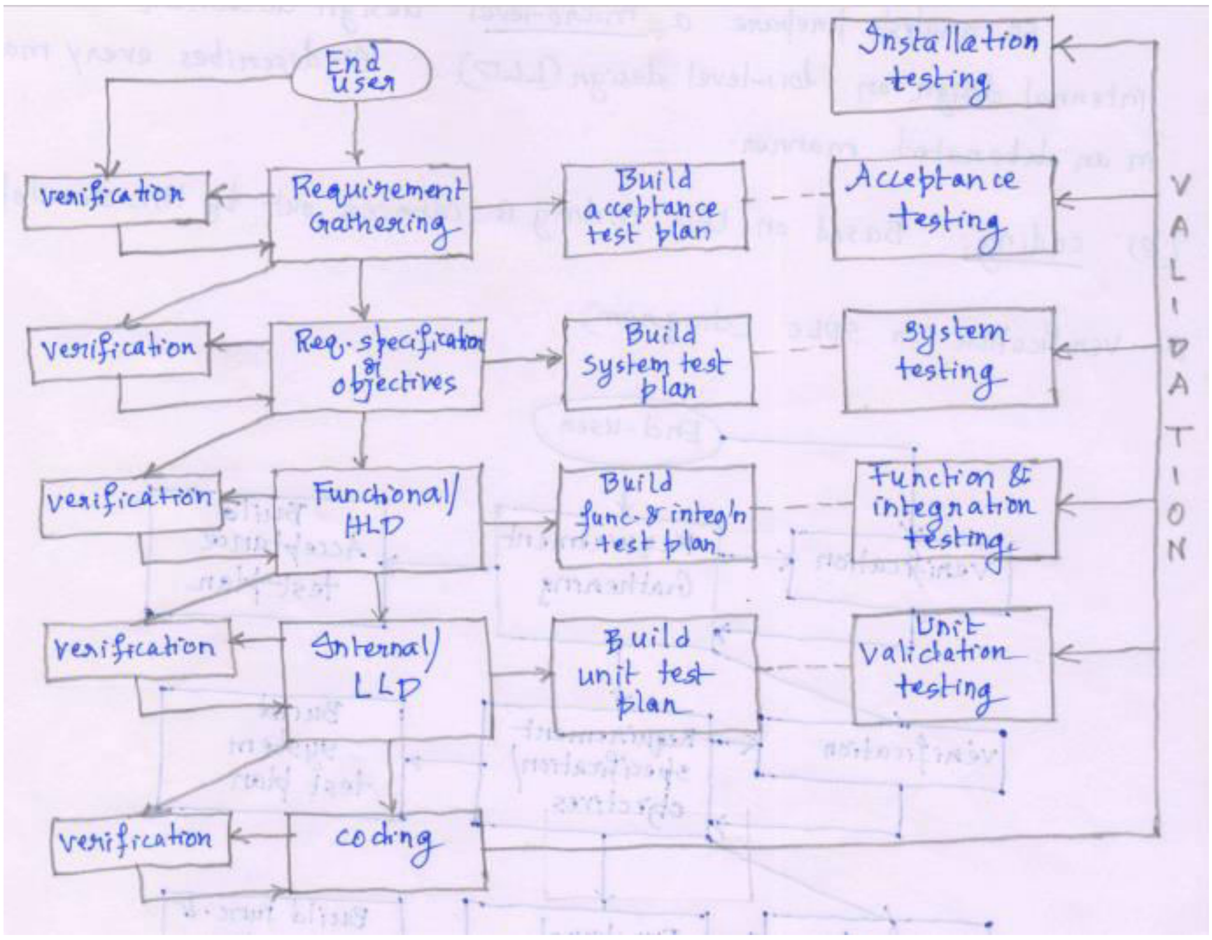


Fig 2.4: V&V Diagram

- Implicit (indirect) Qualities must also be verified: Explicit (stated clearly) qualities are provided in the SRS. Still, the requirements that have not been mentioned over there should also be tested.

8. **Verification Activities**:
   - Verification of requirements and objectives
   - Verification of HLD
   - Verification of LLD
   - Verification of coding (unit verification)

9. **Verification of Requirements**: In this case, all the requirements gathered from the user's viewpoint are verified.

   An 'acceptance criterion' is prepared and it defines the goals requirements of the proposed system. This criterion is important in the real time systems where performance is the critical issue.

Acceptance criteria must be defined by the designers of the system and utilized by the tester in parallel.

(a) Tester reviews the acceptance criteria for clarity, testability, and completeness.
(b) Tester prepares the 'acceptance test plan', which is to be utilized during acceptance testing.

10. **Verification of Objectives**: After gathering requirements, specific objectives are prepared considering every specification. These are prepared in the SRS document. In this activity also, two parallel activities are performed by the tester:
    (a) The tester verifies all the objectives mentioned in the SRS. The purpose is to ensure that the user's needs are understood properly before the next step is taken.
    (b) The tester also prepares the system test plan, which is based on the SRS. It is used in system testing.

11. **Verification Process of Req. and objectives**: R & O verification has a high potential of detecting bugs. An SRS can be verified iff a procedure exists that can be used for checking if the software meets its requirements. Points to be verified:
    (a) **Correctness**: This should be verified by referring to documentation or standards and compare the specified requirement with them. If problems persist, the tester should interact with the user. Tester should also check the correctness in the sense of realistic requirement (if it is applicable or not).
    (b) **Unambiguous** (clear-cut): A requirement should not provide more meanings or interpretations (redundancy). Each requirement should have only one interpretation and each characteristic should be described by a single unique term.
    (c) **Consistent**: No specification should contradict with others. Ex: measurements, legal terms, terminology (function or module or process) etc.
    (d) **Completeness**: Verify that all significant requirements like functionality, performance, design constraints, attributes etc. are complete. Check whether the responses of every input have been defined or not; check whether the figures and tables have been labelled and referenced completely.
    (e) **Updating**: Check whether previous requirements have been updated or new ones have been added. Change the specifications accordingly and check their feasibility.
    (f) **Traceability**: Verify if ach requirement can be traced back to its origin facilitating its documentation. Types here are backward traceability (check the origin of each requirement) and forward traceability (every requirement can be identified and documented uniquely in other documents).

12. **Verification of HLD**: All the requirements mentioned in the SRS are addressed here. Note that the architecture and design is documented in another document called 'Software Design Document' (SDD). In this phase, the tester is responsible for two parallel activities.
    (a) Since the system is decomposed into sub-systems/components, the tester should verify the functionality of these components. Low-level details are not tested or considered here (BBT); the tester should concentrate on how the interfaces will contact with the

outside world. The tester verifies that all the components and their interfaces satisfy each of the requirements.

(b) The tester also prepares a 'function test plan', which is based on the SRS. This plan is used in functional testing (discussed later).

13. The high-level design (HLD) verification is done as follows:
    (a) Data Design: Creates a model of data or information represented at a high level abstraction (user's view of data). In the program components, the design of data structures and the associated algorithms are required to create high-quality applications. The points verified here are:
    - Check whether the sizes of DS have been estimated properly.
    - Check the consistency of data formats with the requirements.
    - Check the chance of any overflow of DS.
    - Check the relationships among data objects.
    - Check the consistency of DBs and DWs with the specified requirements.

    (b) Architectural Design: It focuses on the representation of the structure of software components, their properties and interaction. The points to be verified:
    - Functional requirements
    - Exceptions taken care of or not.
    - Transaction mapping
    - Functionality of each module
    - Inter-dependence and interface between modules.

    (c) Interface Design: It creates a communication medium between interfaces of different software modules/interfaces between s/w system and inputs. The points to be verified:
    - Design of interfaces between the modules
    - Interfaces between humans and computers, their consistency
    - Response time
    - Help offered by the system
    - Error messages/advices/clarifications etc.
    - For typed commands, verify the mapping between menu options and their commands.

14. **Verification of low-level design (LLD)**:
    (a) Verify each module of the LLD such that logic and details are consistent.
    (b) Prepare the unit test plan (used in unit testing).
    (c) Verify the software design and description (SDD) of each module.

15. **Code Verification**: Note that LLD is converted into source code using some programming language. The points to be verified are:
    (a) Check that every design specification in HLD and LLD has been coded using traceability matrix.
    (b) Examine the code against a language's specification(s). [Ex: C/Java]
    (c) Points to be verified:
    - Check for any misused arithmetic precedence

- Mixed mode operations (Integer added to a real number etc.)
- Incorrect initialization
- Incorrect symbolic representation of an expression
- Representation of the data types
- Improper loop termination
- Failure to exit
- Static (Without running the program) and dynamic testing (while the program is running)

16. **Unit Verification**: This method is to test the code as modules brought out by the developers. It is also known as unit verification testing. Points to be tested:
    (a) Interfaces (check if the information is properly flowing in and out of the concerned program)
    (b) The local data structure utilization
    (c) Boundary conditions
    (d) All independent paths (if they had been used at least once or not)
    (e) All error handling paths

17. **Validation**: It is a set of activities that ensures the software under construction satisfies the user requirements. Validation testing is performed after the coding is completed. The need(s) for validation are given below:
    (a) To determine if all the user's requirements have been satisfied by the product.
    (b) To determine whether the product's actual behaviour satisfies the desired behaviour.
    (c) To uncover the bugs after the coding phase.
    (d) To enhance the quality of the software (through updates/patches)
18. Validation Activities:
    (a) **Validation Test Plan**: It starts as soon as the first output of SDLC (SRS) is prepared. In every phase, tester performs verification and validation in parallel. For this, a tester must understand the current SDLC phase, must study the relevant documents and bring out the related test plans with a sequence of test cases. The types of test plans are discussed below:
        - Acceptance Test Plan: Prepared in the requirements phase depending on user acceptance criteria. It is used in acceptance testing.
        - System Test Plan: Prepared to verify the objectives given in the SRS. Test cases are designed considering how the system will behave in different conditions.
        - Function Test Plan: This is prepared in the HLD phase. Test cases are designed in such a way that all the functions and interfaces are tested for their functionality. Used in functional testing.
        - Integration Test Plan: This plan is prepared to validate the integration of all the modules such that their independencies are checked. It also checks out whether the integration output satisfies all the rules of design process. Used in integration testing.

- Unit Test Plan: This is prepared in the LLD phase and consists of a test plan of every module in the system separately. Functionality of every unit is to be tested. Used in unit testing.

(b) Validation Test Execution: The activities here are:

- Unit Validation Testing: It is the process of testing individual components at the lowest level of a system. A unit/module must be validated before integrating it with other modules. Unit validation is the first activity after coding of the module is completed. The motivation for unit testing instead of system testing:
    1. Since the developer concentrates on the smaller 'parts' of the system, it is natural that the unit is primarily tested.
    2. If the whole s/w is tested at the same time, it is difficult to trace the bug. Consequently, debugging is easier in unit testing.
    3. If the module/unit concept is not followed, a team of developers/testers might end up with same/large tasks; parallel works can't be carried out and integration fails.
- Integration Testing: It is the process of combining and testing multiple modules together. The intention here is to discover bugs that have surfaced after the integration of modules. Note that each 'module' should be unit tested first.
- Function Testing: Each of the functionalities of the specified system is tested accordingly. Basically, function testing is to explore the bugs related to differences between actual system behaviour and its functional specifications. The objective here is to *measure the quality* of the functional components of the system.
  Ex: Is a function in a system able to call another?
  Can a function return the expected values?
  Are the interfaces between the programs/functions working correctly?
  We have to check if the function performing correctly, responding with correct return values, exchanges data correctly and started in order. (f1 calls f2. So f1 should start first.)
- System Testing: It is a series of different tests on the total s/w system to check its correctness.
- Acceptance Testing: When the system is ready, it can be tested against the acceptance criteria provided by the customer. The final system is compared against the needs of the user.
- Installation Testing: This checks out if the system can be installed correctly and made to work in normal. This should be done since while installing, some type of files may be converted to other type. Ex: .rar gives extracted files, .exe may be executed to bring out some other files.

19. Views of Testing Techniques: Verification & validation, static and dynamic testing. In static testing, the code is executed while in dynamic testing code is executed. Further, dynamic testing is divided into BBT and WBT.
20. NOTE: Regression testing is carried out after making some changes to the code according to testing report so as to make sure that new errors have not been induced.

21. Black box Testing Techniques: This technique considers only the functional requirements of the s/w or module. It is one of the major techniques in dynamic testing to design effective test cases. The structure or logic of the software is not considered.

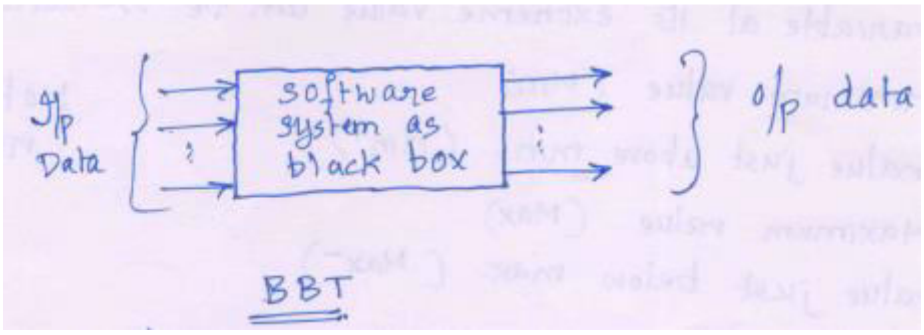


Fig. 2.5: Black box testing

22. BBT attempts to find errors in the following categories:
    - To test the modules independently
    - To test the functional validity of the software (identify the missing functions) Ex: Incorrect or missing functions
    - To look for interface errors
    - To test the system behaviour and performance
    - To test the maximum load or stress on the system
    - To test the acceptance limits of the software (user should be satisfied with its working)
23. Boundary Value Analysis (BVA): It is a BBT technique that uncovers bugs at the boundaries of input values. The maximum or minimum values taken by the inputs are considered.
    Ex: A is an integer between 10 and 100. The boundary check can be (9,10,11) for minimum values and (101,100,99) for the maximum values.
    Let B be an integer between 10 and 50. Then we consider (9,10,11) and (51,50,49).
    This can be represented as:

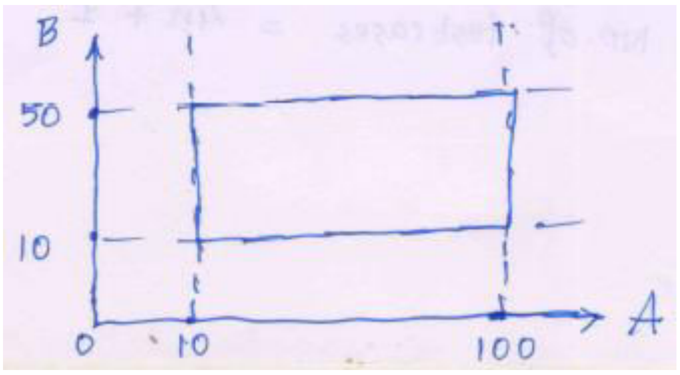


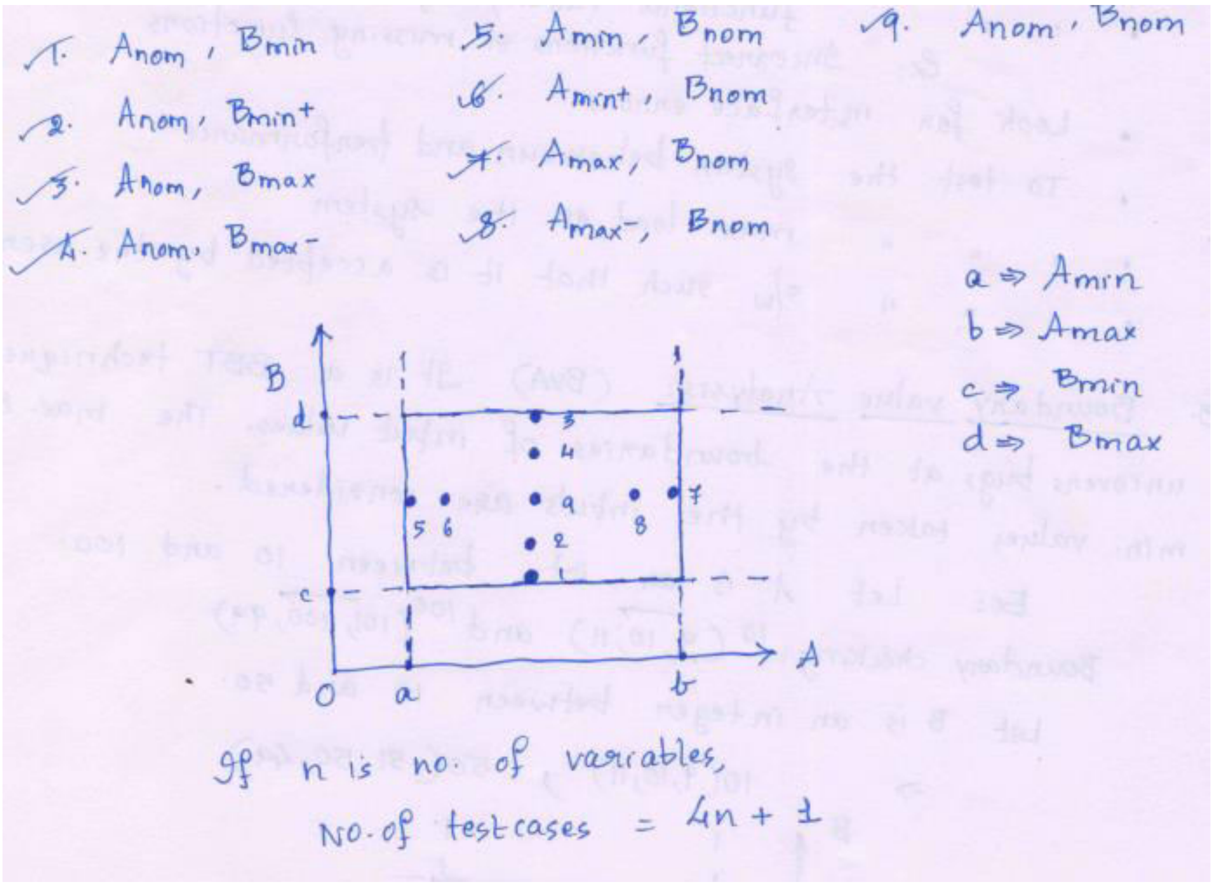
Fig 2.6: BVA

24. Types of BVA:
    (a) Boundary Value Checking: Here the test cases are designed by holding one variable at its extreme value and other variables at their normal values in the input domain.
    The variable at its extreme value can be selected at:
    - Minimum value (min)

- Value just above min (min$^+$)
- Maximum value (max)
- Value just below max (max$^-$)

  Ex: Consider two variables A and B. (nom => nominal values)

  Test cases:



1. Anom , Bmin
2. Anom , Bmin+
3. Anom , Bmax
4. Anom , Bmax-
5. Amin , Bnom
6. Amin+ , Bnom
7. Amax , Bnom
8. Amax- , Bnom
9. Anom , Bnom

a ⇒ Amin
b ⇒ Amax
c ⇒ Bmin
d ⇒ Bmax

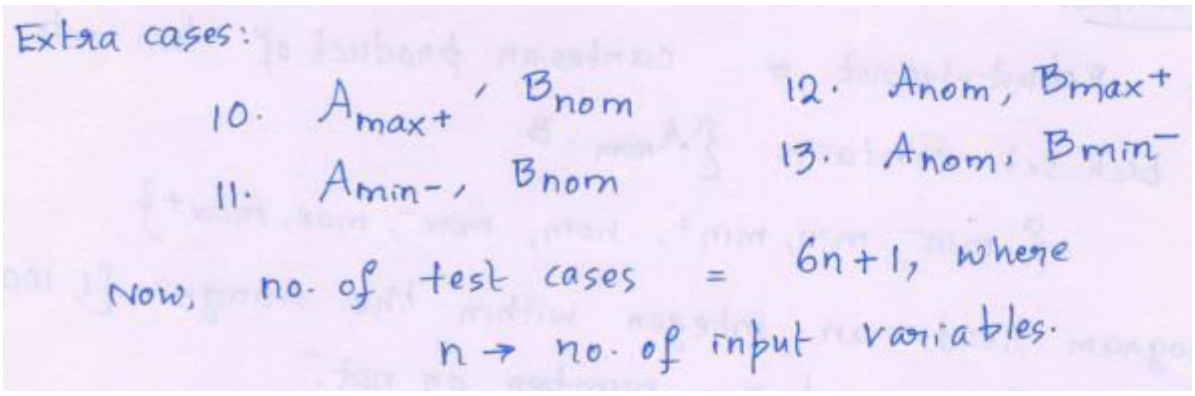
If n is no. of variables,
No. of test cases = 4n + 1

Fig 2.7: BVC

(b) Robust Testing Method: In this, BVC is extended such that boundary values are considered as:

- A value just greater than the maximum value (max$^+$)
- A value just less than the minimum value (min$^-$)

Ex: Considering the previous example again, test cases are 1 to 9. Extra cases here are:



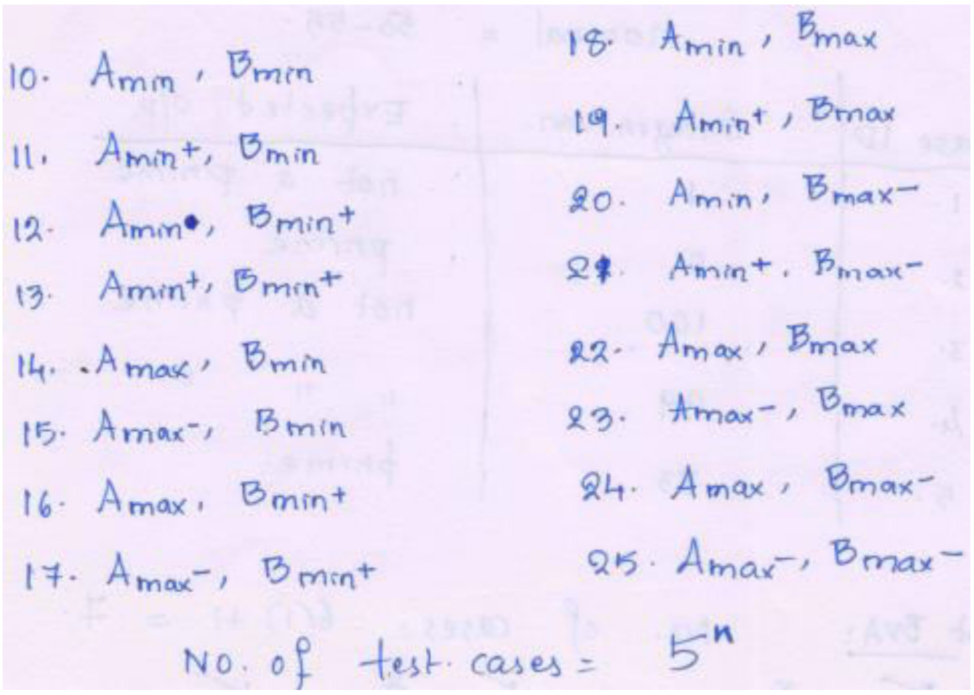
Extra cases:

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

Now, no. of test cases = $6n+1$, where
$n \rightarrow$ no. of input variables.

Fig 2.8: Robust Method

(c) Worst-Case testing Method: BVC is again extended by assuming more than one variable on the boundary.
Ex: Consider the 9 test cases of BVC. It is extended as follows:



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

No. of test cases = 5^n

Fig 2.9: Worst case Testing method

(d) BVA is applicable when the concerned module is a function of several independent variables. It is to be considered when boundary ‘condition’ checking is given priority.
Ex: Systems checking max/min temperature, pressure, speed etc.
NOTE: BVA is not useful for Boolean variables.

(e) Robust-worst => Cartesian product of two or more sets, where each set contains:
{min-, min, min+, nom, max-, max, max+)

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

BVC

No. of variables (n) = 1.

 Total no. of test cases = 4n+1=5.

Min = 1
Min+ = 2
Max= 100
Max- = 99
Nom = 40-45

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

Test Case ID	**Integer Variable**	**Expected Output**
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	43	Prime number

Robust

Total no. of test cases = 6n+1=7.

Min = 1
Min- = 0
Min+ = 2
Max=100
Max-=99
Max+=101
Nom=40-45

Test cases:

Test Case ID	**Integer Variable**	**Expected Output**
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	43	Prime number

Worst

Since there is one variable, total number of test cases = 5^n = 5.

 Number of test cases will be same as BVC.

Robust-Worst

Total no. of test cases = 7^n = 7; no. of test cases will be same as robust.

26. Equivalence Class Testing: A typical input domain of variable(s) and their combinations is too large to test every input. To overcome this problem we should divide the input domain based on a common feature/class of data. Equivalence partitioning is a method for deriving test cases in which equivalence classes are identified at positions where :
    (a) Each member causes SAME kind of processing
    (b) Each member causes SAME kind of OUTPUT to occur

    Instead of testing every input value, one case from each set of equivalence classes is sufficient for finding errors.

    Ex: We are interested in values among 50-100. Instead of testing each and every value, it is acceptable to consider a single value.

    This way of testing uses minimum no. of test cases to cover the whole input domain. It also gives maximum probability of finding errors (hit rate) with minimum test cases.

27. Goals of equivalence partitioning:
    (a) Completeness: Without utilizing all the test cases possible, we try to obtain the best result.
    (b) Non-redundancy: Reduce the inputs that are generated from the same class.
28. Steps in Equivalence Partitioning:
    (a) Identify Equivalent Classes (b) Design Test cases

29. Identification of Equivalent Classes: Equivalent Classes are formed by grouping inputs for which results/behaviour pattern is similar.

    Types of Classes:
    (a) Valid Equivalent Classes: Valid inputs to the program are considered.
    (b) Invalid Equivalent Classes: Invalid inputs that generate errors are considered here.

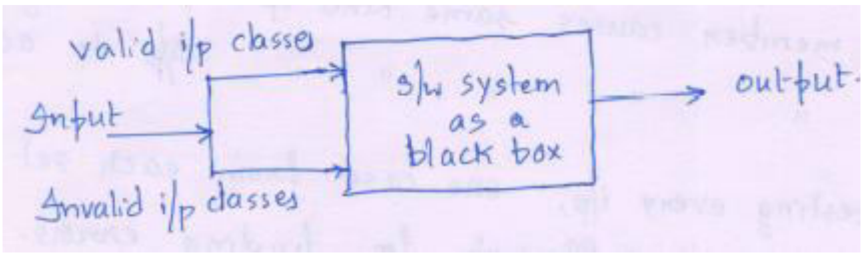


Fig. 2.10: Equivalence Classes

30. Guidelines for forming equivalence classes:
    1. If a range of inputs doesn't produce the same outputs, divide the range into two or more equivalent classes.
    2. If each valid input is treated differently by the program, then one valid equivalent class = one valid input.
    3. BVA can be used in identifying classes.
       Ex: Let $0 \le a \le 100$. Valid equi. classes = 1 (in the range)
       Invalid classes = $a<0$ and $a>100$.

4. If an input variable can identify more than one category, make equi. classes for each category.
   Ex: Input is a character. Equi. class=alphabets/numbers/special characters
   Invalid classes=None of the above (white space)
5. If a 'must be' condition exists, identify two equi. classes. (valid/invalid)
   Ex: First character of username should be a letter.
6. We can also focus on outputs to form equi. classes.

31. Identifying the test cases:
   (a) Assign a unique number for each equivalent class.
   (b) Check if all equivalent classes have been covered by the test cases.
   (c) Check if all invalid equivalent classes have been covered.

32. Ex: Let A, B, C be three numbers. Range: [1, 50]. Find the largest number.
   Inputs:
   $I_1 = \{<A,B,C>: 1 \le A \le 50\}$; $I_6 = \{<A,B,C>: B<1\}$;
   $I_2 = \{<A,B,C>: 1 \le B \le 50\}$; $I_7 = \{<A,B,C>: B>50\}$;
   $I_3 = \{<A,B,C>: 1 \le C \le 50\}$; $I_8 = \{<A,B,C>: C<1\}$;
   $I_4 = \{<A,B,C>: A<1\}$; $I_9 = \{<A,B,C>: C>50\}$;
   $I_5 = \{<A,B,C>: A>50\}$;

Test Case ID	A	B	C	Expected Result	Classes covered by the Test case
1.	13	25	36	C is greatest	I_1, I_2, I_3
2.	0	13	45	Invalid	I_4
3.	51	34	10	Invalid	I_5
4.	20	0	18	Invalid	I_6
5.	31	53	40	Invalid	I_7
6.	41	46	0	Invalid	I_8
7.	21	32	51	Invalid	I_9

33. State Table-based Testing:
   Finite State Machine (FSM): It is a mathematical model of computation used to design computer programs and circuits. It is an abstract machine that can be in one of its finite number of states. The outcome depends on previous and current inputs. FSM is used to design functional testing (BBT).

34. State Transition Diagrams (State Graphs): A system or its components may have a no. of states depending on its inputs and time allocated.
   Ex: States of an OS:
   (a) New state (new task)
   (b) Ready (waiting in the queue)
   (c) Running (being executed)
   (d) Waiting (for an input/output/an event to occur)
   (e) Terminated (finished execution)

States => nodes; nodes are connected by links (transitions); this leads to a state transition diagram or state graph.

A state graph is a pictorial representation of FSM.
Transition => one state is changed to another.

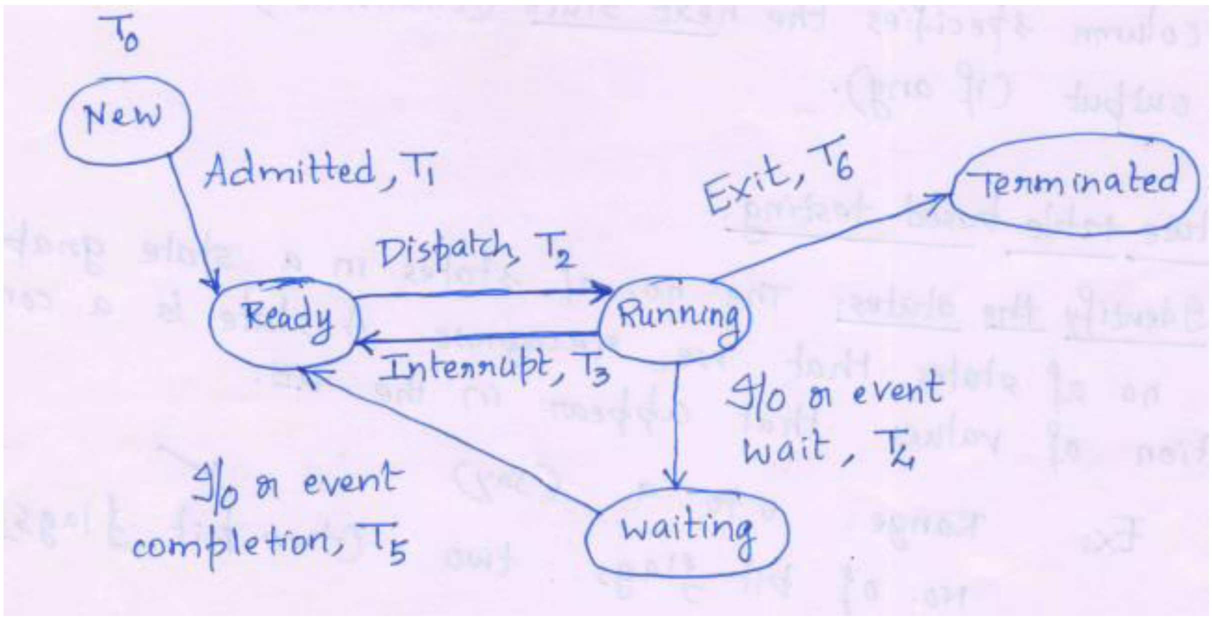
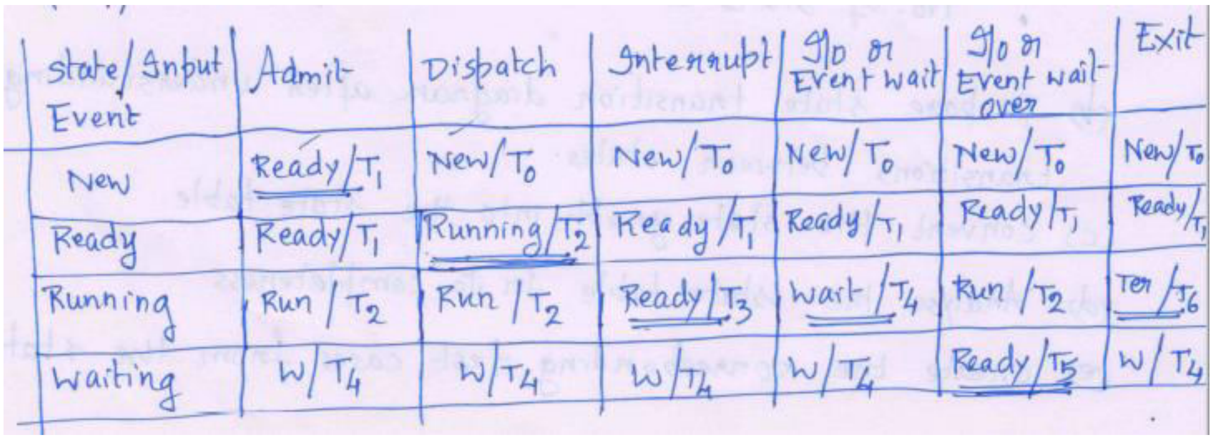


Fig. 2.11: State graph of an OS Concept

An arrow link provides:

(a) Transition events (dispatch, interrupt etc.)
(b) Resulting output from a state

35. State Table: State graphs are converted into tabular form to specify the states, inputs, transitions and outputs.
Ex:



state/Input Event	Admit	Dispatch	Interrupt	I/O or Event wait	I/O or Event wait over	Exit
New	Ready/T_1	New/T_0	New/T_0	New/T_0	New/T_0	New/T_0
Ready	Ready/T_1	Running/T_2	Ready/T_1	Ready/T_1	Ready/T_1	Ready/T_1
Running	Run/T_2	Run/T_2	Ready/T_3	wait/T_4	Run/T_2	Ter/T_6
Waiting	W/T_4	W/T_4	W/T_4	W/T_4	Ready/T_5	W/T_4

Fig. 2.12: State Table

Conventions used for state table:

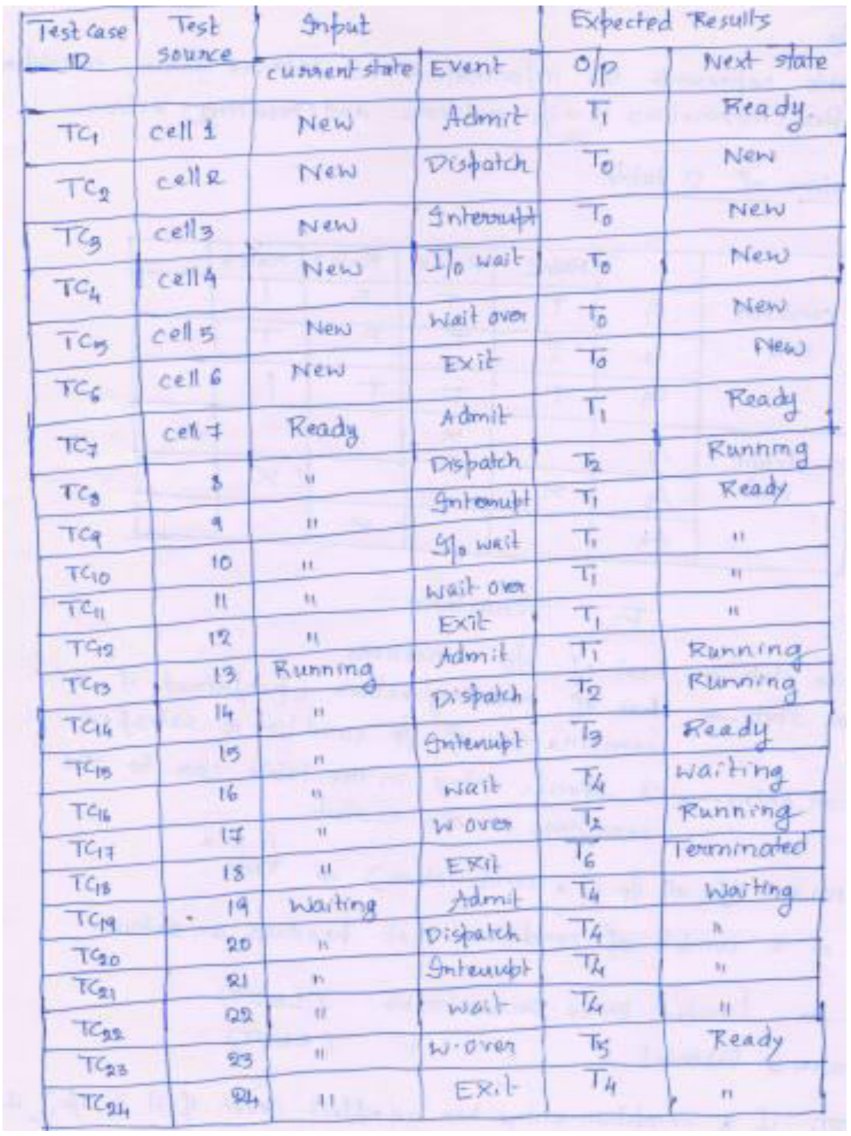
(1) Each row of the table corresponds to a state
(2) Each column corresponds to an input condition
(3) Each box/cell specifies the next state (transition) and the output (if any).

36. State Table-based Testing:

(a) Identify the states: The no. of states in a state graph is the no. of states that we recognise. A state is a combination of values that appear in the database.
Ex: Say, range = [0, 9].
No. of bit flags (not bits) = two
- (2*2)*10 = 40 states

The common bugs here are: Failing to show/recognise all the states
Find the no. of states as follows:
(1) Identify all the component factors of the state
(2) Identify all the allowable values for each factor
(3) No. of states = no. of allowable values * no. of factors

(b) Prepare state transition diagram after understanding the transitions between the states
(c) Convert the state graph into the state table
(d) Analyse the state table for its completeness
(e) Create the corresponding test cases from the state table
Ex: For the state table on the P15, the test cases are given below:



Test case ID	Test source	Input		Expected Results	
		current state	Event	O/P	Next state
TC_1	cell 1	New	Admit	T_1	Ready.
TC_2	cell 2	New	Dispatch	T_0	New
TC_3	cell 3	New	Interrupt	T_0	New
TC_4	cell 4	New	I/o wait	T_0	New
TC_5	cell 5	New	wait over	T_0	New
TC_6	cell 6	New	Exit	T_0	New
TC_7	cell 7	Ready	Admit	T_1	Ready
TC_8	8	"	Dispatch	T_2	Running
TC_9	9	"	Interrupt	T_1	Ready
TC_{10}	10	"	I/o wait	T_1	"
TC_{11}	11	"	wait over	T_1	"
TC_{12}	12	"	Exit	T_1	"
TC_{13}	13	Running	Admit	T_1	Running
TC_{14}	14	"	Dispatch	T_2	Running
TC_{15}	15	"	Interrupt	T_3	Ready
TC_{16}	16	"	wait	T_4	waiting
TC_{17}	17	"	W over	T_2	Running
TC_{18}	18	"	Exit	T_6	Terminated
TC_{19}	19	Waiting	Admit	T_4	Waiting
TC_{20}	20	"	Dispatch	T_4	"
TC_{21}	21	"	Interrupt	T_4	"
TC_{22}	22	"	wait	T_4	"
TC_{23}	23	"	w-over	T_5	Ready
TC_{24}	24	"	Exit	T_4	"

Fig. 21.3: Test cases for the state table

37. Decision Table Based Testing: BVA and equivalent class don't consider *combinations* of input conditions. Each input is considered separately.

    Decision table represents the information in a tabular form, considering complex combinations of input conditions and resulting actions.

    (a) Formation of a Decision Table:

Condition Stub		Rule 1	Rule 2	Rule 3	Rule 4	...
	C1	T	T	F	I	
	C2	F	T	F	T	
	C3	T	T	T	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

- Condition Stub: List of input conditions
- Action stub: List of resulting actions (performed if a combination of input is satisfied)
- Condition Entry: A specific entry in the table corresponding to the conditions in the condition stub
- Rule: True/False/Immaterial (for all inputs of a combination)
- If we use only T/F, then the table is known as limited entry decision table (LEDT)
- If we use several values, the table becomes extended entry decision table (EEDT)

Example: The conditions for admission (marks) into a university are given below:
Java ≥ 70; C++ ≥ 60; UML ≥ 60; Total ≥ 220 OR Total (of Java and C++) ≥ 150
If the aggregate is ≥ 240 => eligible for scholarship; else, normal admission.
So, the outputs are: (a) Not eligible for admission (b) Eligible for scholarship (c) Eligible for normal admission

Design the test cases using decision table testing.

	R1	R2	R3	R4	R5	R6	R7	R8	R9
C1: J ≥ 70	T	T	T	T	F	I	I	I	T
C2: C++ ≥ 60	T	T	T	T	I	F	I	I	T
C3: UML ≥ 60	T	T	T	T	I	I	F	I	T
C4: Total ≥ 220	T	F	T	T	I	I	I	F	T
C5: Total (J and C++) ≥ 150	F	T	F	T	I	I	I	F	T
C6: Aggregate ≥ 240	F	F	T	T	I	I	I	I	F
A1: Normal	X	X							X
A2: Scholarship			X	X					
A3: Not eligible					X	X	X	X	

Minimum no. of test cases = no. of rules [Here, it is 9]
Maximum no. of test cases = 2^n (n-> no. of conditions) [Here, it is $2^6 = 64$]

The test case details are given below:

Test case ID	Java	C++	UML	Aggregate	Expected output
1.	75	75	75	225	Normal
2.	75	75	70	220	Normal
3.	75	74	93	242	Scholarship
4.	76	77	89	242	Scholarship
5.	68	78	80	226	Not eligible
6.	78	45	78	201	Not eligible
7.	80	80	50	210	Not eligible
8.	70	72	70	212	Not eligible
9.	75	75	76	226	Normal

(b) Action Entry: It is an entry in the table for an action to be performed when one rule is satisfied. Usage of ‘X’ is action entry; else we leave the box blank.

- List all actions associated with a module/procedure
- List all conditions during execution of the procedure
- Associate sets of conditions with actions and remove impossible combinations
- Define the rules by indicating what action will result for a set of conditions.

38. Test Case Design using Decision Table:
    - Interpret condition stubs as inputs for the test case
    - Interpret action stubs as expected outputs for the test case
    - Rule becomes a test case
    - If k rules exist for n binary conditions, there are at least k test cases and 2^n maximum test cases.
39. Example: An app to validate a number according to the following data:
    C1: Number can start with an optional sign
    C2: Optional sign can be followed by any number of digits
    C3: Digits followed by a decimal point represented by a period
    C4: If a decimal exists, there should be two digits after decimal
    C5: Any number should be terminated by a blank space

    Use T/F/I.
    Answer: Maximum no. of rules = $2^{\text{conditions}} = 2^5 = 32$
    Outputs possible: (Actions) Valid and Invalid

	R1	R2	R3
C1	I	I	...
C2	I	I	...
C3	T	F	...
C4	T	I	...
C5	T	T	...
A1: Valid	X	X	...
A2: Invalid			

40. Cause-Effect Graph (CEG): CEG is another BBT technique for combinations of input conditions. CEG takes help of a decision table (DT) to design a test case.

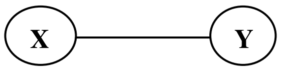
    CEG is the technique to represent the situations of combinations of input conditions. Then, we can convert the CEG into a DT for obtaining the test cases.

    CEG helps in selecting the 'required' combinations of input conditions in a systematic way.

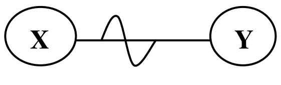
    NOTE: The combinations must not be too large in number.

41. Methodology of CEG:
    (a) Division of Specification(s): A specification is divided into possible no. of pieces and parts.
    (b) Identification of causes and effects:
    Cause => Distinct input condition. (reason)
    Effect => Distinct output condition (consequence)
    (c) Transformation of specification into CEG: A specification is transformed into a Boolean graph that links causes and effects.
    (d) Conversion into DT: CEG is converted into 'limited entry' DT (LEDT) by verifying state conditions in the graph.
    (e) Each column represents a test case; test cases are derived from them.

42. Notations for CEG:
    (a) Identity:

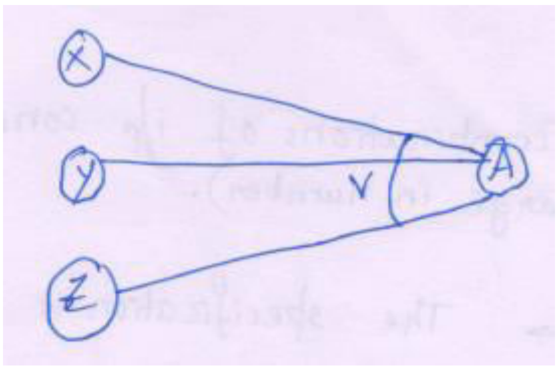


    If x=1, y=1 else y=0
    (b) NOT:

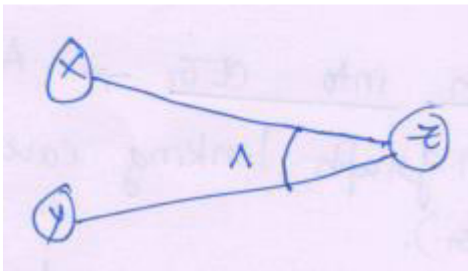


    If x=1, y=0 else y=1

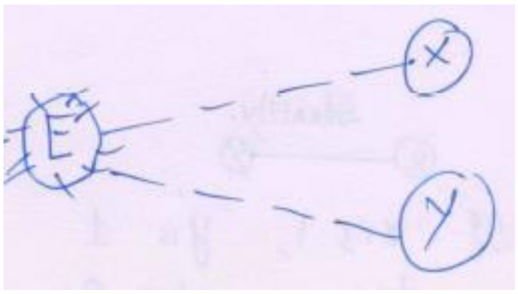
    (c) OR:



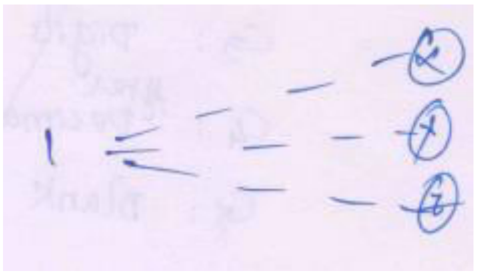
(d) AND:



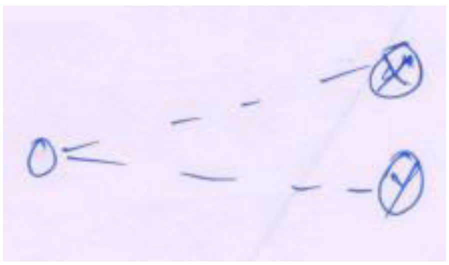
(e) Exclusive: Either x or y can be 1 but not both.



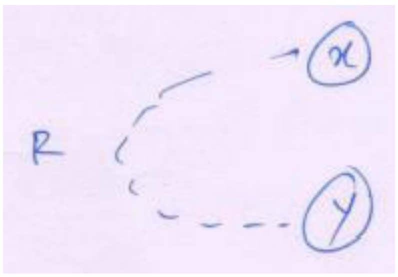
(f) Inclusive: At least one of x, y and z must always be 1.



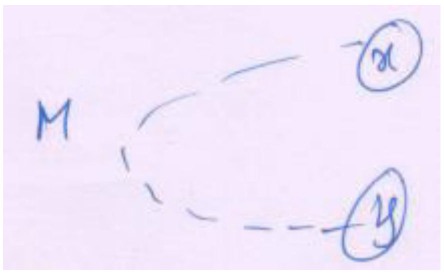
(g) One and only one: Only one of x and y must be 1.



(h) Requires: For x to be 1, y must be 1.



(i) Mask: If x=1, y is forced to be 0.



43. Error Guessing: This is performed when all other models have failed. It is also used in some special situations.

By this method, those errors/bugs can be guessed which do not fit into any of the previous methods. It is used by experienced people to find out the errors easily.

For this purpose, the history of bugs is useful to identify some errors that are repeated again and again.

Error guessing is an 'ad-hoc' approach based on experience, insight, knowledge of project and bug history.

Ex: What will happen when a=0 in the quadratic equation?

(a) If a=0 then the equation is not quadratic
(b) For calculation of roots, division is by zero

What will happen if all inputs are –ve?

What will happen if the input list is empty?

****

Software Testing Methodologies (R13)

UNIT – 2

Quiz

1. All _____ activities can be seen in the form of verification and validation. [KEY: B]
   (a) Development
   (b) Testing
   (c) Maintenance
   (d) Effectiveness

2. The document prepared on the specified objectives of a system to be developed is known as _____. [KEY: C]
   (a) Requirement Gathering
   (b) SQA
   (c) SRS
   (d) SDLC

3. If ___ is not performed at early stages, there can be a mismatch between the required and delivered product. [KEY:B]
   (a) Validation
   (b) Verification
   (c) Maintenance
   (d) Testing

4. Checking that each requirement references its source in previous documents is known as ___. [KEY:C]
   (a) Traceability
   (b) Previous Traceability
   (c) Backward Traceability
   (d) Reverse Traceability

5. The process of converting LLD specifications into a specific language is called ___. [KEY: D]
   (a) Mapping
   (b) Testing
   (c) SRS
   (d) Coding

6. The testing technique that doesn’t involve program execution is called ___. [KEY: A]
   (a) Static
   (b) Dynamic
   (c) Unique
   (d) Manual

7. The validation activities involve ___ and ___. [KEY: C]
   (a) Test case, Test execution
   (b) Test case, Test plan
   (c) Test plan, Test execution
   (d) Test execution, test report

8. The objective of function test is to measure the ___ of the ____ components of the system. [KEY: A]
   (a) Quality, functional
   (b) Speed, Functional
   (c) Performance, Structural
   (d) Quality, Structural

9. If some modifications are done for the program, testing the old test cases again is called ___ testing. [KEY: C]
   (a) Smoke testing
   (b) Performance testing
   (c) Regression testing
   (d) Dynamic testing

10. ____testing largely maps verification and dynamic testing to validation. [KEY: C]
    (a) Dynamic
    (b) Smoke
    (c) Static
    (d) Regression

11. BBT is also known as ___ testing. [KEY: D]
    (a) Structural
    (b) Regressive
    (c) Manual
    (d) Functional

12. BBT can look for ___ errors. [KEY: D]
    (a) Interface
    (b) System behaviour
    (c) Load & stress
    (d) All of the above

13. In ____ method, test cases are designed by holding one variable at its extreme value and other values at their nominal values. [KEY: B]
    (a) Boundary value Analysis
    (b) Boundary Value Checking
    (c) Robust Boundary value
    (d) Robust Worst BVA

14. By taking more than one variable on the boundary, we utilize the ____ method. [KEY: D]
    (a) BVA
    (b) BVC
    (c) Robust
    (d) Worst

15. The total no. of test cases in BVC is ____ where ‘n’ is the no. of variables. [KEY: B]
    (a) 6n+1
    (b) 4n+1
    (c) 3n+1
    (d) 5^n

16. In ____ testing, instead of producing every test case possible, only one test case for every partition can be utilized. [KEY: C]
    (a) Equal Class
    (b) Robust-Worst
    (c) Equivalence Class
    (d) Single module

17. A ___ is a behavioural model whose outcome depends upon previous and current inputs. [KEY: C]
    (a) State transition
    (b) State table
    (c) Finite state machine
    (d) State graph

18. ____ takes the help of ____ to design a test case. [KEY: A]
    (a) Cause-effect graph, decision table
    (b) Decision table, Cause-effect graph
    (c) FSM, Cause-effect graph
    (d) Cause-effect graph, Condition/action stubs

UNIT – 3

Course (Unit-3) Objectives:

- Make the student familiar with white box testing and different methods in it
- Get the student acquainted with Basis path testing and its methods
- Application of WBT by using graph matrices
- Understand the process of loop testing and data flow testing
- Make the student recognize and use mutation testing, both in theory and lab
- Make the student to understand and apply all methods of static testing

1. White box testing (WBT): Also known as glass box testing, this method studies the entire design, structure and code of the software. It is also known as structural or development testing.
2. Need of WBT: The reasons for the need (requirement) of WBT are given below:
   (a) WBT is used for testing a program at its initial stage. BBT is the second stage testing. Note that though BBT test cases can be designed earlier than WBT test cases, they can be executed only after WBT. Hence WBT is not an alternative but essential stage.
   (b) WBT is supportive to BBT; there exist some bugs that can be revealed only through WBT but not BBT.
   (c) We MUST execute WBT test cases to reveal bugs that have been transported from earlier phases of SDLC.
   (d) Some logical paths that are not used frequently can also be tested by WBT.
   (e) WBT can also find typographical errors which go unobserved in BBT.
3. Logic Coverage Criteria: WBT considers the program code and the test cases are designed based on the logic of the program. Every part of the logic is covered in WBT test cases.
4. Statement Coverage: It is assumed that if all the statements are executed at least once, every bug will be notified.

   Ex:
   ```
   scanf("%d", &x); scanf ("%d", &y);
     while(x!=y)
     {
      if (x>y)
         x=x-y;
      else
         y=y-x;
     }
     printf("x=%d",x);printf("y=%d",y);
   ```

   To cover all these statements, the test cases are:
   T1: x=y=n, where n is any number
   T2: x=n,y=m where n and m are different numbers

   T1 skips the 'while' loop. T2 executes the loop but every statement inside the loop is not executed. For this, two more test cases are added.

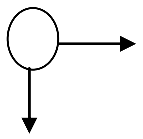
T3: x>y T4: x<y

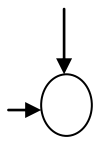
Though all statements are now covered, the logic of the program is not yet under testing. It can be concluded that statement coverage is necessary but not sufficient.

5. Decision or Branch Coverage: Branch coverage states that each decision considers all possible outcomes at least once. Considering the previous example, the test cases here must consider the outcomes of both 'while' and 'if' statements.
   T1: x=y; T2: x!=y; T3: x<y; T4: x>y
6. Condition Coverage: This states that each condition in a decision takes on all possible outcomes at least once.
   Ex: while((i≤5) && (j>COUNT)). Two conditions exist here. The test cases can be:
   T1: i≤5, j>count T2: i<5, j>count
7. Condition coverage in a decision doesn't mean that the decision also has been covered. If the decision "if(A&&B)" is being tested, the condition coverage would allow to write two test cases.
   T1: A is true, B is false; T2: A is false, B is true
   Note that these test cases have no effect on the 'then' clause of the 'if' statement. So this requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once and each point of entry is invoked (called) at least once.
8. Multiple Condition Coverage: It requires that sufficient test cases should be written so that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. The following types of test cases can exist:

	A	B
TC1	T	T
TC2	F	F
TC3	T	F
TC4	F	T

9. Basis Path Testing: This technique is based on the control structure of the program. It is the method of selecting the paths that provide a basis set of execution paths through the program. BPT is used to identify and utilize those paths that cover maximum logic – not all the paths.
10. Guidelines for effectiveness of path testing:
    (1) Path testing (PT) is based on control structure of the program and a flow graph is prepared to depict the same.
    (2) PT requires complete knowledge of the program structure.
    (3) PT is mainly used by the developer.
    (4) The effectiveness of PT is reduced with the increase in the size of the software.
    (5) Choose enough paths such that maximum logic of the program is covered.
11. Control Flow Graph (CFG): It is a graphical representation of the control structure of a program.
    V-> Vertices; E-> Edges/links; N-> nodes

Decision node: 

Junction Node: 

Region -> Area bounded by edges and nodes

When counting the regions, the area outside the graph is also considered as a region.

12. CFG Notations:

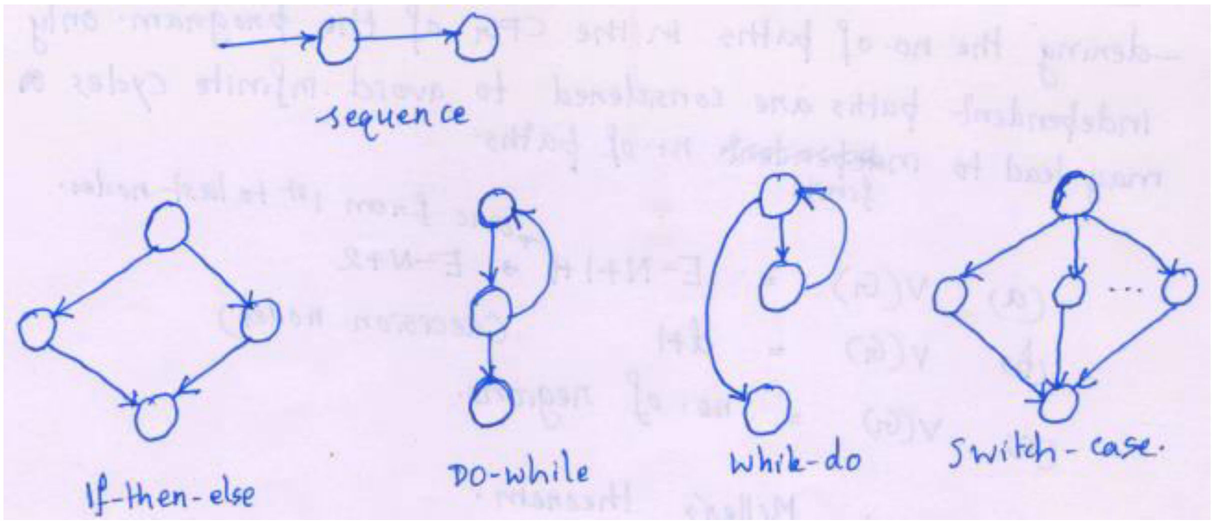


Fig. 3.1: CFG Notations

13. Path Testing Terminology:
    - Path: It is a sequence of instructions/statements starts at an entry, junction or decision and at another junction, decision or exit.
    - Segment: A path consists of segments where the smallest segment is a single link.
    - Path Segment: It is succession of consecutive links that belong to the same path.
    - Length of a path: It is measured by the number of links (or sometimes by no. of nodes) in it; not by the no. of statements executed.
    - Independent Path: It is any path through the graph that introduces at least one new set of processing statements or new conditions. It must move along at least one edge that has not been traversed before the path was defined.

14. Cyclomatic Complexity (CC): Complexity is measured by considering the no. of paths in the CFG of the program. Only independent paths are considered to avoid infinite cycles.
    (a) V(G) = E-N+2
    (b) V(G) = R
    (c) V(G) = P+1 (predicates) [Miller's Theorem]

15. Applications of Path Testing:
    (a) Through Testing/More Coverage: It gives us the no. of test cases that are considered as important. CC also provides more coverage of paths.
    (b) Unit Testing: Path testing is mainly used by in structural testing of a module. In unit testing, it is more useful since each outcome of a decision is considered.

(c) Integration Testing: Since one module may call one or more other modules, interface errors are likely to surface here. Path testing analyzes all the possible paths on the interfaces and explores all the errors.

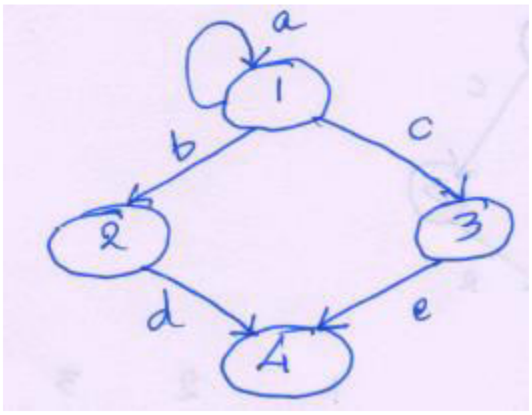
(d) Maintenance Testing: PT is also necessary for the modified version(s) of the software and to re-test the modules or interfaces.

(e) Testing effort is proportional to the complexity of the software. PT takes care of this, and obtains the no. of paths to be tested by CC.

(f) Basis Path testing concentrates more on error-prone software. The CC (or the no. of paths) signifies that the testing effort is only on the error-prone part of the software, thus minimizing the testing effort.

16. Graph Matrices: Graph matrix is a square matrix whose rows and columns are equal to the no. of nodes in the flow graph. Each row and column identifies a particular node and the matrix entries are the connections between the nodes. Each cell is a link between the two concerned nodes.
NOTE: If there is a link between node ‘a’ and node ‘b’, it doesn’t mean that the opposite is also true.

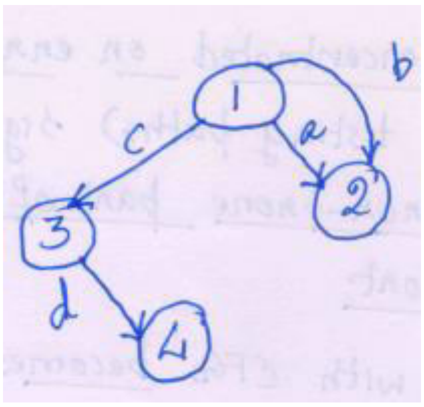
17. Ex:



Graph Matrix:

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

18. Ex:



Graph Matrix:

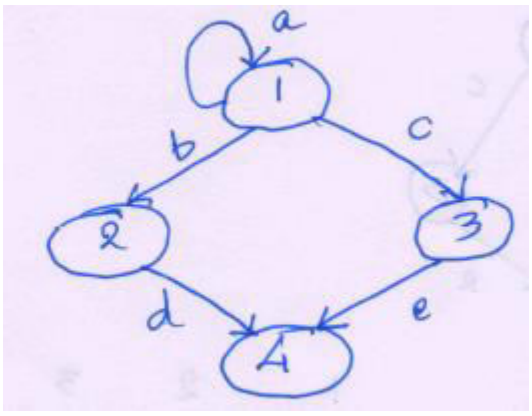
	1	2	3	4
1		a+b	c	
2				
3				d
4				

19. Connection Matrix: The matrix representation used above is only a tabular representation of the graph with no other extra information.

    If link weights are added to each cell entry, the graph matrix can be used as a powerful tool in testing. The links between two nodes are assigned link weight that becomes an entry in the matrix cells.

    *Link weight provides direct information about control flow. If a link exists, link weight is 1; else 0 or empty cell.

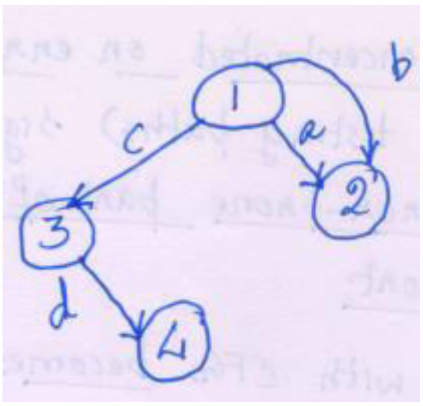
    Ex:



Connection Matrix:

	1	2	3	4
1	1	1	1	
2				1
3				1
4				

Note that even if more than one link is present between two nodes we show it only as 1.



Connection Matrix:

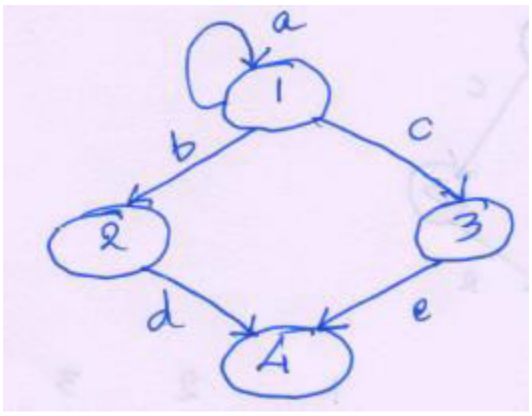
	1	2	3	4
1		1	1	
2				
3				1
4				

2-1=1

1-1=0

Total = 1+1=2 (Cyclomatic number)

20. Method:
    (1) For each row, count the no. of 1s and write it in front of the row.
    (2) Subtract 1 from that number. Ignore the blank rows.
    (3) Add the final count of all rows.
    (4) Add 1 to the final count to obtain the final sum.
    (5) The number obtained is known as Cyclomatic number of the graph.
21. Ex:



	1	2	3	4
1	1	1	1	
2				1
3				1
4				

3-1=2
1-1=0
1-1=0

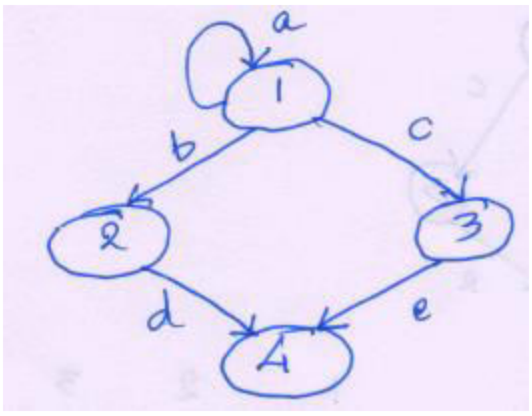
Total = 2+1 = 3

22. Use of graph matrix for finding set of all paths: Producing a set of paths is important to trace out the k-link paths.
    Ex: How many 2-link paths exist from node 1 to node 2?

    Aim: Using matrices to obtain the set of all paths between all nodes.

    The power operation on a matrix expresses the relation between each pair of nodes via intermediate nodes under the assumption that the relation is transitive. (a->b, b->c => a->c).

23. Ex:



	1	2	3	4
1	a	b	c	
2				d
3				e
4				

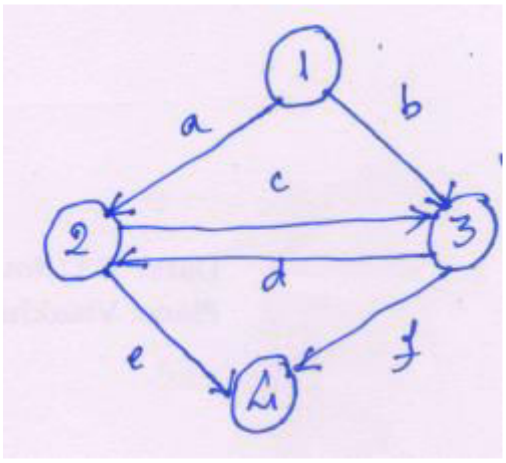
Find 2-link paths for each node.

For finding 2-link paths, we should square the matrix.

$$\begin{array}{cccc} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{array} \quad X \quad \begin{array}{cccc} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{array} \quad = \quad \begin{array}{cccc} a^2 & ab & ac & bd+ce \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}$$

The resulting matrix shows all the 2-link paths from one node to another.
Ex: From node 1 to node 2, there exists a link ‘ab’

24. Ex2:



Graph Matrix:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & a & b & 0 \\ 2 & 0 & 0 & c & e \\ 3 & 0 & d & 0 & f \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

2-link Paths:

2-link paths:

$$\begin{bmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3-link Paths:

3-link paths:

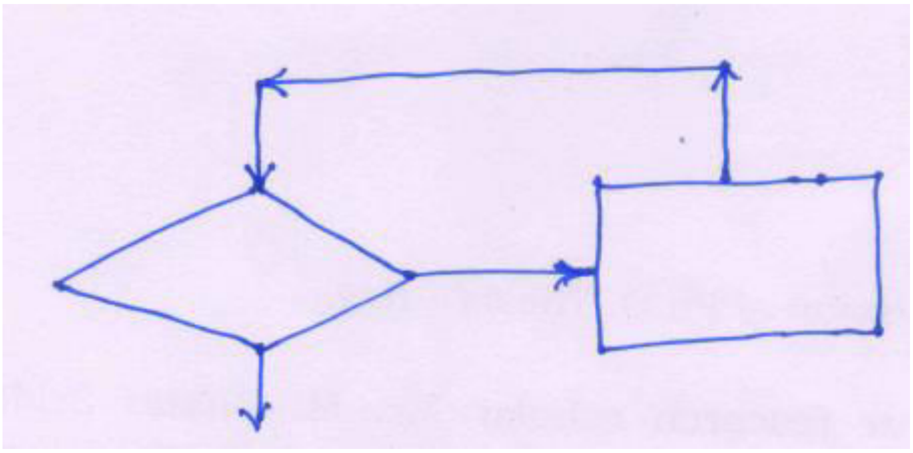
$$\begin{bmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & acd & bdc & bde+acf \\ 0 & 0 & cdc & cde \\ 0 & dcd & 0 & dcf \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

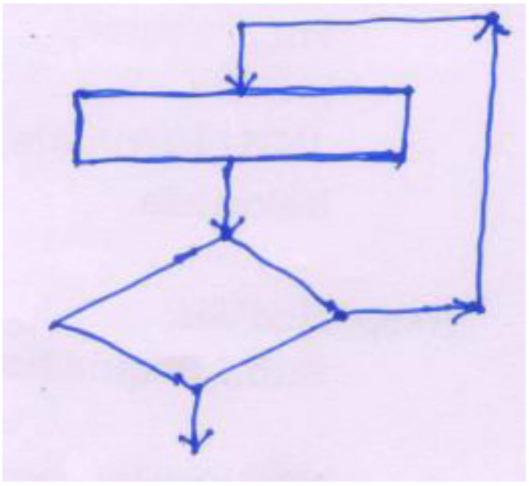
25. □ For n number of nodes, we can get the set of all paths of (n-1) links length with the use of matrix operations.

    These operations can be programmed and used as a software testing tool.

26. Loop Testing: It is an extension to branch coverage. If loops are not tested properly, bugs may be undetected. Loop testing can be performed effectively while performing development testing on a module.

    (a) Simple loops: Single loop exists in the control flow.



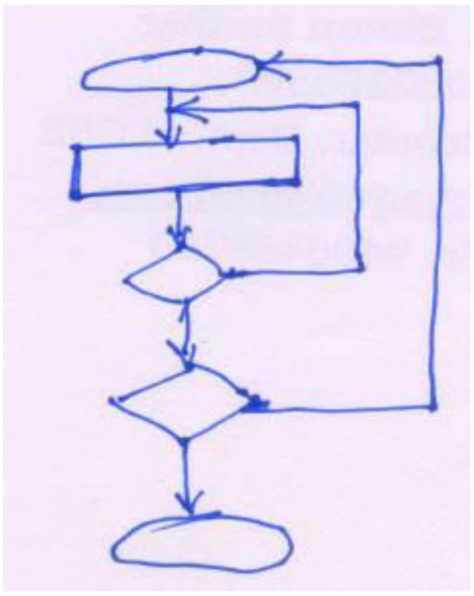


Test cases to be considered here:

- Check whether you can bypass the loop or not.

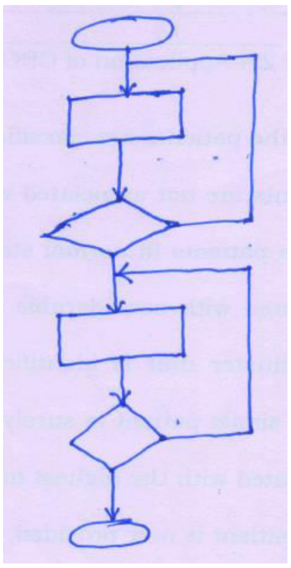
- Check whether the loop control variable is negative or not.
- Write one TC that executes the statements inside the loop.
- Write TCs for a typical no. of iterations through the loop.
- Write TCs for checking the boundary values of the maximum and minimum no. of iterations. (min-1, min, min+1 .....)

(b) Nested Loops: When two or more loops are embedded, it becomes a nested loops.



In this case, we should start with the innermost loop while holding the outermost loops to their minimum values.

(c) Concatenated Loops: Two loops are concatenated if it is possible one after exiting the other, while still on a path from entry to exit.



(d) Unstructured Loops: These loops cannot be tested directly and must be converted into nested/concatenated loops or the program must be resigned.

27. Data Flow Testing: It is a WBT technique that can be used to detect improper use of data values due to coding errors.
    Ex: Not initialized before usage or not utilized at all after declaration/initialization.

DFT looks out for inappropriate data definition, its usage in predicates, computations and terminations. Mostly used for detect out of scope data usage (double for int).

For this purpose, A CFG is used.

28. States of a Data Object:
    (a) Defined (d): A data object (DO) is called defined (d) when it is initialized i.e., on the LHS of an assignment statement.
    Other examples: File has been opened, an object has been allocated, data pushed into a stack, a record written etc.
    (b) Killed (k): When an object has been reinitialized, scope of a loop variable is finished, file closed etc.
29. Usage (u): DO is on the RHS of an assignment statement, or used as a loop control variable, as a pointer etc. Notes that there are two types of usage: 'c' for computation and 'p' for predicate.
30. DF Anomalies: These are usage patterns which may lead to incorrect code execution.

Anomaly	**Acronym**	**Effect of Anomaly**
'du'	Define-use	Normal case. Allowed.
*'dk'	Define-kill	Potential bug. Data killed without use.
'ud'	Use-define	Data is used and redefined. Normal.
'uk'	Use-kill	Allowed.
*'ku'	Kill-use	Serious bug.
'kd'	Kill-define	Allowed.
*'dd'	Define-define	Redefining without any usage. Potential bug.
'uu'	Use-use	Normal.
*'kk'	Kill-kill	Harmless bug but not allowed.

NOTE: Potential Bug => having a capacity to become a bug in the future.

31. Single Character Anomalies:
    -x: All prior actions are of no interest to x.
    x-: All post actions are of no interest to x.

Anomaly	**Explanation**
-d	Normal.
*-u	Used without definition. Potential Bug.
*-k	Data (might be) killed before definition. Bug.
k-	OK
*d-	[define last]. Potential bug.
u-	OK

32. Terminology in DFT: Let
    P --> program
    G(p) --> graph
    V--> variables
    Assumption is that G(p) has single entry and exit nodes.
    (1) Definition Node: Defining a variable means assigning a value to a variable for the first time in the program.

Ex: Input statements, assignment statements, loop control statements, procedure calls etc.

(2) Usage Node: It means the variable has been used in some statement of the program. Node ‘n’ that belongs to G(p) is a usage node of variable ‘v’ if the value of v is used at the statement corresponding to node n.
Ex: Output statements, assignment statements (RHS) etc. [c, p]

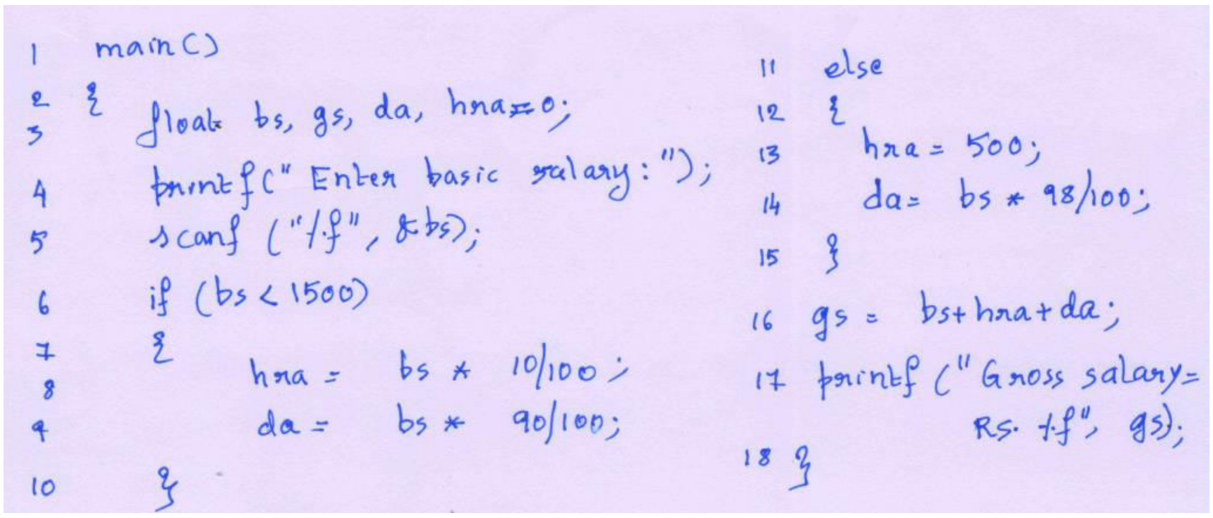
(3) Loop-free Path Segment: It is a path segment for which every node is visited once at most.

(4) Simple Path Segment: One node is visited twice. It may be loop free or if a loop does exist, only one node is involved.

(5) ‘du’ path: Path between definition node and usage node of a variable.

(6) Definition-clear Path (dc path): Path between definition node and usage node of a variable such that no other node in the path can redefine the variable v.

33. Static Data Flow Testing: Source code is analysed without execution of the program.
Ex: If basic < 1500, HRA = 10% of basic & DA = 90% of basic. If basic >= 1500, HRA = 500 & DA = 98% of basic. Calculate the gross salary.



```
1   main()
2   {
3       float bs, gs, da, hra=0;
4       printf(" Enter basic salary:");
5       scanf("%f", &bs);
6       if (bs < 1500)
7       {
8           hra = bs * 10/100;
9           da = bs * 90/100;
10      }
11  else
12  {
13      hra = 500;
14      da = bs * 98/100;
15  }
16  gs = bs+hra+da;
17  printf("Gross salary=
            Rs. %f", gs);
18  }
```

Find out d-u-k patterns for all variables in the source code.
For ‘bs’:

Pattern	Line No.	Explanation
-d	3	Normal
du	3-5	Normal
uu	4-6, 6-7, 7-12, 12-14	Normal
uk	14-16	Normal
k-	16	Normal

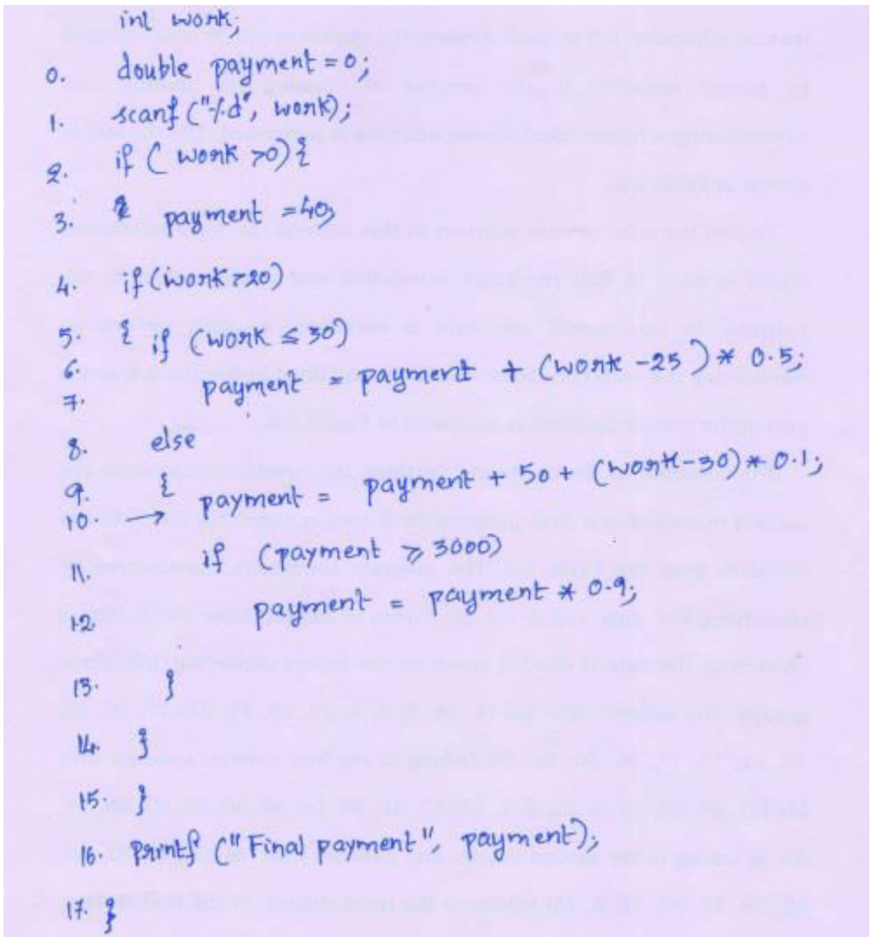
For ‘da’:

Pattern	Line No.	Explanation
-d	7	Normal
du	7-4	Normal
uk	14-16	Normal
k-	16	Normal

For ‘hra’:

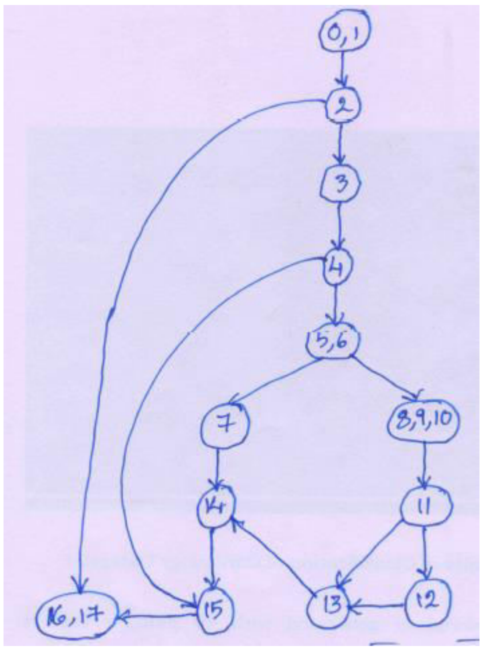
Pattern	Line No.	Explanation
-d	1	Normal
dd	1-6 or 1-11	Harmless bug.
du	6-14 or 11-14	Normal
uk	14-16	Normal
k-	16	

34. Disadvantages of Static Analysis:
    (a) It is not possible to determine the state of a data variable by only static analysis. This applies when the variable is an array or pointer and so on.
    (b) If the index is generated dynamically, static analysis is of no use.
    (c) The static methodology might detect an anomaly that is used rarely but not all of them.
35. Dynamic DFT: This is done to detect bugs in data usage during code execution. The TCs here try to detect every d and u of a data variable. The strategies are given below:
    (a) All du paths (ADUP): Every du-path of every variable should be used under some test. This can be quoted as the strongest strategy that uses maximum number of paths for testing. It is also the superset of all other strategies.
    (b) All uses (AU): For every usage of a variable, there is a path from the definition to the use.
    (c) APU+C (All predicates uses + some computational uses): For every variable at least one dc path should exist to all of its predicate uses. If predicate uses are less, we can add some c uses.
    (d) ACU+P (All computational uses + some predicate uses): One dc path to every computational uses and some predicate uses also.
    (e) APU
    (f) ACU
    (g) AD: All definitions are covered.
36. Example: A C program for calculating work and concerned payment is given below:

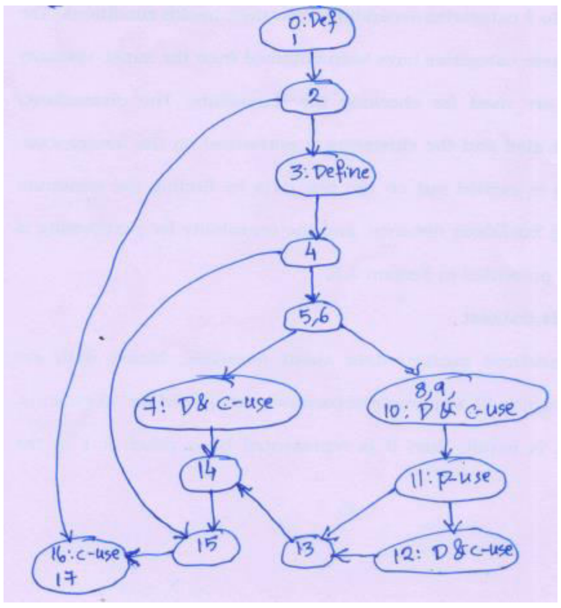


```
    int work;
0.  double payment = 0;
1.  scanf ("%d", work);
2.  if ( work >0) {
3.  {   payment =40;
4.  if (work >20)
5.  {  if (work ≤ 30)
6.
7.          payment = payment + (work -25) * 0.5;
8.      else
9.      {  payment =  payment + 50 + (work-30) * 0.1;
10. →
           if  (payment ≥ 3000)
11.
           payment = payment * 0.9;
12.
13.     }
14. }
15. }
16. printf ("Final payment", payment);
17. }
```

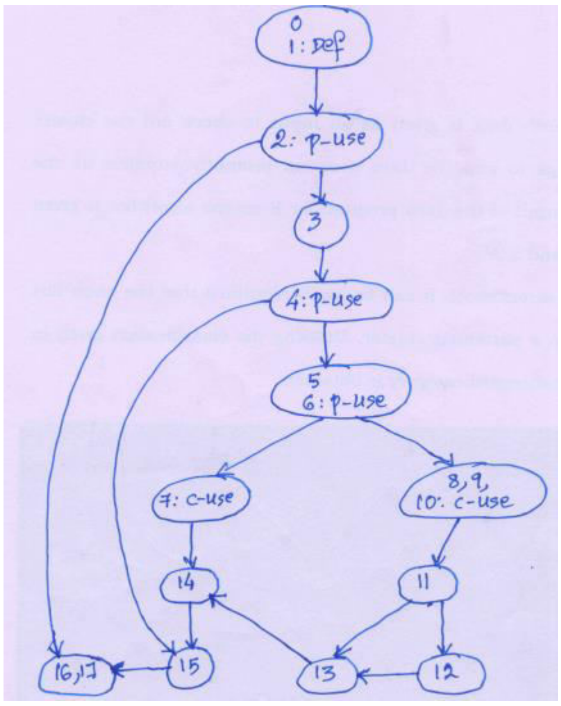
Decision to Decision Path Graph (DD Path Graph) for this program is given below:



DFG for 'payment' variable:



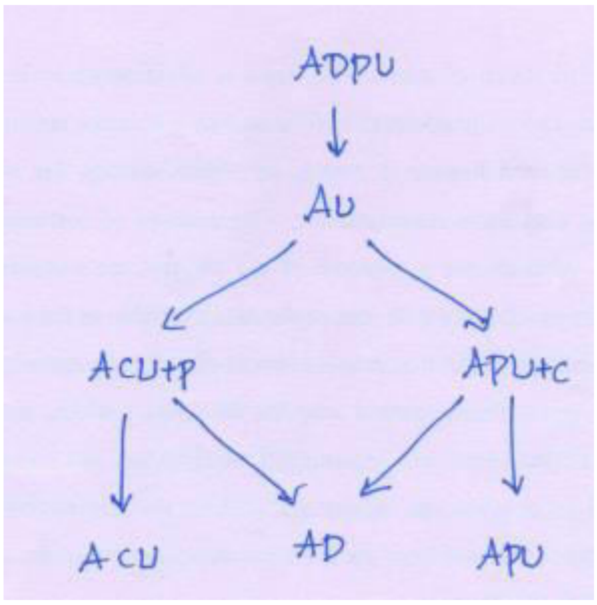
DFG for 'work' variable:



List:

Variable	Defined at	Used at
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

37. Order of DFT Strategies: Note that the strongest strategy is at the root and weakest at the last level.



38. Mutation Testing: It is the process of mutating (changing) some segment of code (placing some error) and testing this mutated code with some test data.

    If the test data can detect some errors, it is acceptable; else the quality of test data must be improved so that they can detect at least some mutants.

    Mutation helps the user to create high quality test data.

    Faults are introduced/induced into the program in different ways – each program becomes a new version. The goal here is to make the program collapse in some way.

    A mutant is *killed* if a test case causes it to fail. This is done since the fault has been noticed and rectified – now there is no use of the mutant.

    The main objective here is to select efficient test data with high error-detection power.

39. Primary Mutants:
    Ex: ......
    ```
    if (a>b)
      x+=y;
    else
      x=y;
    printf("%d",x);
    .........
    ```

    Mutants that can be considered here are:
    M1: x=x-y;
    M2: x=x/y;
    M3: x=x+1;

M4: printf("%d",y);

It is to be understood from the above example that primary mutants are single modifications of the initial program (using some operators). Mutation operators are dependent on programming languages that are to be worked upon.

Results:

Test Data	X	Y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2 (x & y ≠ 0)	4	3	7	1.4 (M2)
TD3 (y≠1)	3	3	6	4 (M3)
TD4 (y≠0))	4	2	6	2 (M4)

40. Secondary Mutants:

Ex:
```
if (a<b)
    c=a;
```
Mutants can be:

```
M1: if (a<=b-1) c=a;
M2: if (a+1<=b) c=a;
M3: if (a==b) c=a+1;
```

These are known as secondary mutants since multiple levels of mutation are applied on the initial program.

Ex1: Original Program (P)
```
r=1;
for(i=2;i<=3;++i)
{
        if(a[i] > a[r])
                r=i;
}
```
The mutants for P are:
```
M1: r=1;
     for(i=1;i<=3;++i)
    {
        if(a[i] > a[r])
                r=i;
     }
```

```
M2:
r=1; for(i=2;i<=3;++i) {
if (i>a[r])
  r=i;  }
```

M3:
```
r=1; for(i=2;i<=3;++i) {
if(a[i] >= a[r])
r=i;  }
```

M4:
```
r=1; for(i=2;i<=3;++i) {
if(a[r] > a[r])
r=i;  }
```

Test Data Selection:

	A[1]	A[2]	A[3]
TD1	1	2	3
TD2	1	2	1
TD3	3	1	2

Apply the above data to mutants M1, M2, M3 and M4.

	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2,M4
TD3	1	1	1	1	1	NONE

M1 and M3 are not killed.
Hence test data is incomplete and new test data is needed.

41. Mutation Testing Process:
    - Construct the mutants (induce faults).
    - Add TCs to the mutation system and check the outputs.
    - If the output is incorrect, that means a fault has been found, program must be modified and the process must be restarted.
    - If the output is correct, execute that TC with every mutant.
    - If the output of a mutant differs from the original program on the same test case (TC), kill the mutant.
    - After each TC is executed, the remaining mutant(s) fall into two categories:
      (a) One: Mutant is functionally equivalent to the original program.
      (b) Two: The mutant can be killed, but a new set of TCs must also be created and tested for the mutant before killing it.
    - Mutant score for a set of test data = The % of non-equivalent mutants killed by that data.

NOTE: If mutation score = 100%, then the test data is called mutation adequate. (satisfactory)

42. Drawbacks of Dynamic Testing:
    (a) Dynamic testing uncovers bugs at later stages of SDLC; so it becomes costly to debug.
    (b) Dynamic testing is expensive and time consuming (create, run, validate and maintain TCs).
    (c) Efficiency decreases as the size of software increases.
    (d) Dynamic testing provides information about bugs but debugging is not easy since the exact cause and place of failure are difficult to locate.
    (e) Dynamic testing can't detect all potential bugs. Ex: dd and kk are potential bugs (i.e., warnings in the present but may become bugs in the future).

43. Static testing techniques are now to be applied (in deep) in WBT to obtain higher quality of software. Static testing is also called human/non-computer testing.

    Static testing can be applied for most of the verification activities, at every stage of SDLC. It checks the software and confirms that all standards are met. Ex: Static testing can be applied for requirement design, test plans, source code, user manuals, maintenance etc.

    *Nearly 60% of the errors can be reported by static testing:

    Advantages:
    (a) Quality of product increases since bugs can be found out and fixed at early stages.
    (b) Overall cost when compared to dynamic testing is low.
    (c) By the usage of static testing, dynamic testing also improves.
    (d) Productivity increases.

44. Types of static testing:
    (a) Software Inspections
    (b) Walkthroughs
    (c) Technical reviews
45. Inspections:
    - Introduced by IBM in 1970s.
    - Detects and removes errors after each phase of SDLC and improves the quality of the software.
    - It is a manual examination of the software.
    - It can be applied to any product at any phase of SDLC.
    - Doesn't need executable code or test cases. Bugs here are found on less executed paths.
    - Inspection is machine independent and can be used even before the software is ready (on algorithms).
    - The documents that can be inspected are: SRS, SDD (s/w design and development), code and test plan.
    - Inspection involves inspection steps, role of participants, item being inspected.

- Entry and exit criteria are used to determine whether an item is ready to be inspected.

46. Inspection Team Members:
    - Author: Programmer who has written the code and who has to fix the bugs.
    - Inspector: Finds the errors. (Tester)
    - Moderator: Manages this whole (inspection) process.
    - Recorder: Files all the details of an inspection meeting.
47. Inspection Process: Also called Fagan's original Process.

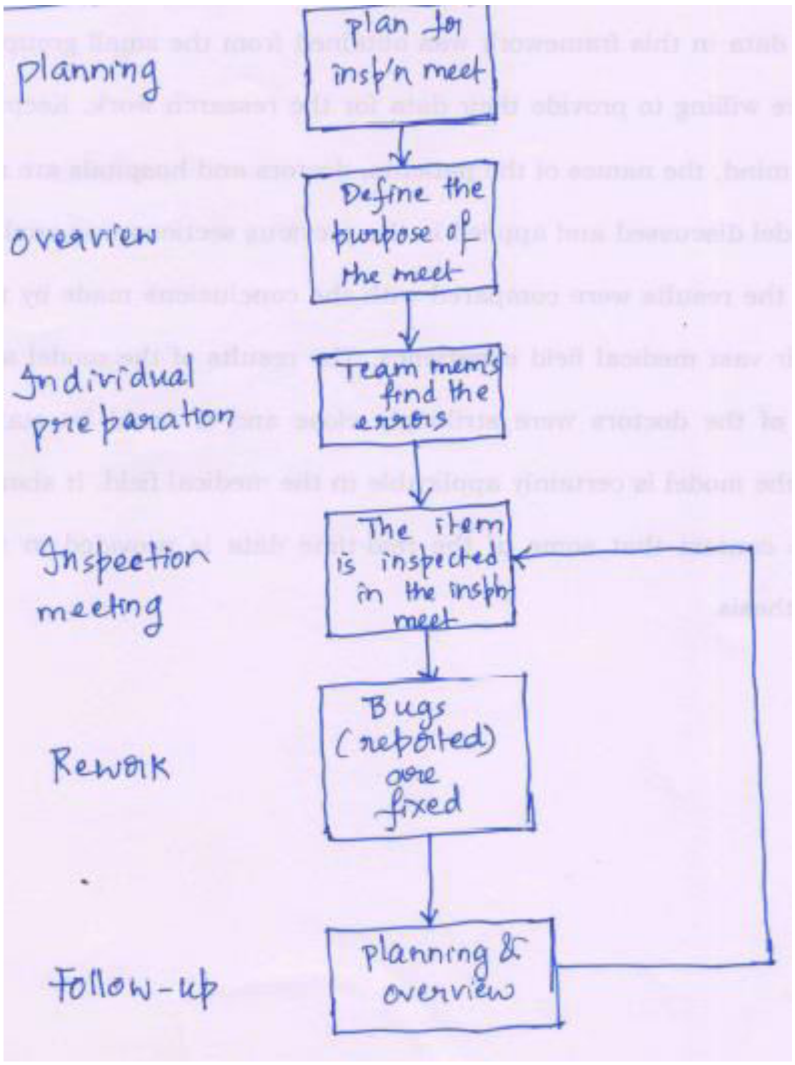

planning
plan for insp'n meet
overview
Define the purpose of the meet
Individual preparation
Team mem's find the errors
Inspection meeting
The item is inspected in the insp'n meet
Rework
Bugs (reported) are fixed
Follow-up
planning & overview

Fig 3.7: Inspection Process

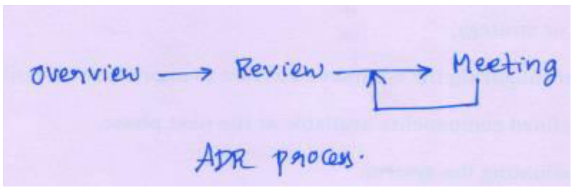
48. Planning (Inspection): The product to be inspected is identified, a moderator is assigned, and the goal is stated. The moderator checks if the product is ready for inspection, selects the team, gives the tasks, schedules time, and distributes the inspection material.
49. Overview: Here the inspection team is provided with the needed information for inspection.
50. Individual Preparation: A team member now utilizes the information provided in overview and prepares for the inspection process.
    Potential Errors are traced out, filed and reported to the moderator.
    Checklists are used during this stage for guidance on bug identification. Logs are prepared.
    Moderator checks the efficiency of the log and submits a final report to the author.

51. Inspection Meeting: It is the stage of actual inspection where the author checks each issue raised. All the members arrive at a final conclusion deciding the issues that have to be fixed, how and by whom. Summary of the meeting is produced by the moderator.
52. Rework: The author fixed the bugs and reports back to the moderator.
53. Follow-up: Moderator checks if all the reported bugs have been fixed. If not, another inspection meet is called.
54. Advantages of Inspection Process:
    - Bug Reduction: the no. of bugs per 1000 lines can be reduced by two-thirds.
    - Bug Prevention: Experiences of previous inspections can be used for future bug prevention.
    - Productivity: Cost of inspection is less since no execution is needed, thereby increasing the productivity. (General: 23%)
    - Real-time feedback to developers
    - Reduction in resource usage: Since bugs are identified and fixed close to their origin we get less cost and more time.
    - Quality Improvement: Better logic, more testing, etc.
    - Checking coupling and cohesion: Note that coupling is the level of independence between two software modules and cohesion is the degree to which elements of module belong to each other. Hence high cohesion => loose coupling.
    - Learning through inspection: This improves the programming capability of the team members.
    - Process Improvement: Learning from the results
      Ex: Finding most error-prone modules and identifying the error types.
55. Effectiveness of Inspection Process: Results of an analysis show that 52% of the errors can be found by inspection. In the remaining 48%, only 17% are detected by WBT.

    Rate of Inspection => How much evaluation of an item has been done by the system
    Rate is very fast => Coverage is high, but less no. of errors are detected
    Rate is too slow => Coverage is low, more no. of errors are found. But cost increases.

    □ Error Detection Efficiency = Error found by an inspection
    -- * 100
    Total no. of errors before inspection

56. Cost of Inspection Process: Let 4 members be part of an inspection team. 100 lines of code are considered. Then, cost of inspection = one person per a day of effort. Note that testing costs depend upon the no. of faults in the software.
57. Variants (diff. types) of Inspection Process:
    (a) Active Design Reviews (ADR): These are used for inspecting the design phase of SDLC. It covers all the sections of design based on a set of small reviews instead of a single large set. (single review)

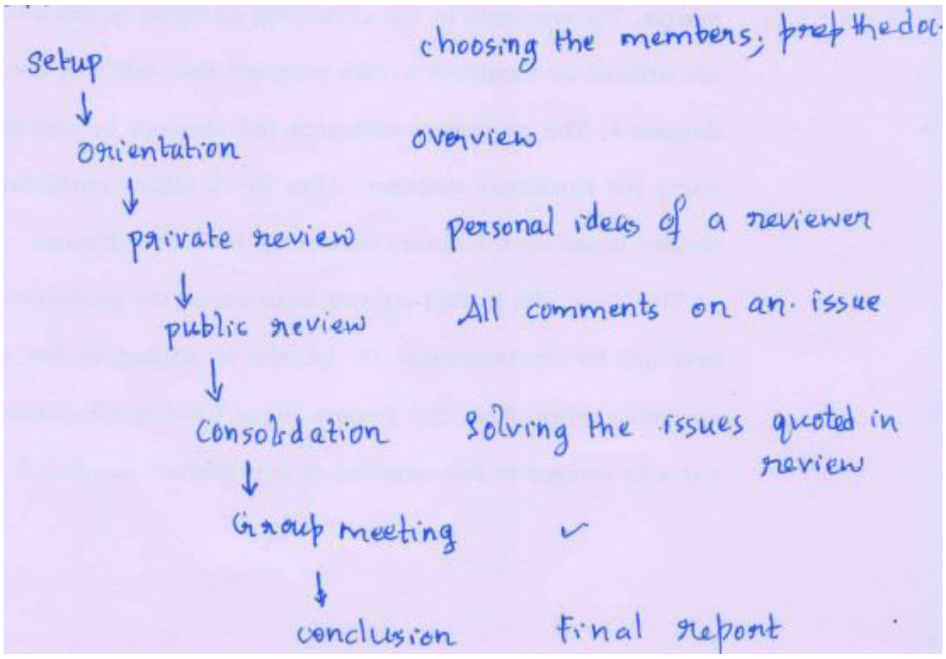


Overview: Explains the structure of the concerned module

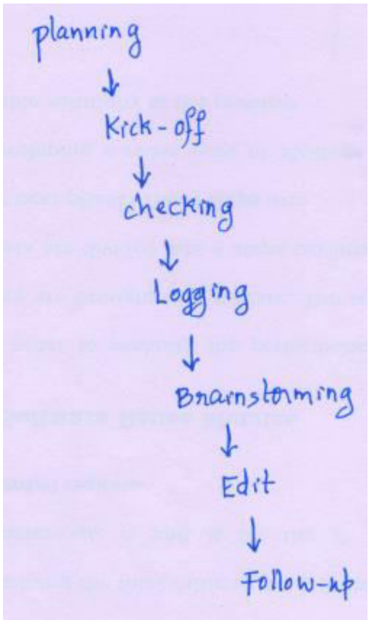
Review: Reviewers are given questionnaire and time frame to answer, raise any doubts etc.

Meeting: Designers read the responses and resolve the queries, problems etc. This meeting(s) goes on till all the issues are resolved.

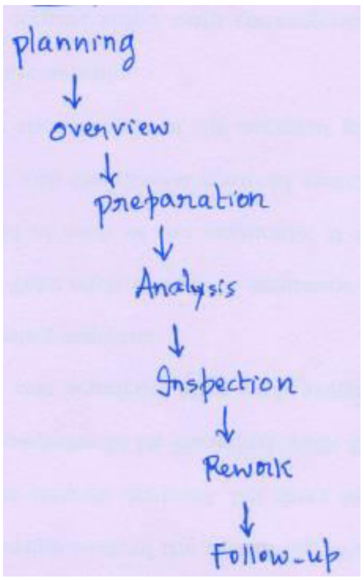
(b) Formal technical Asynchronous Review Method (FTArm):



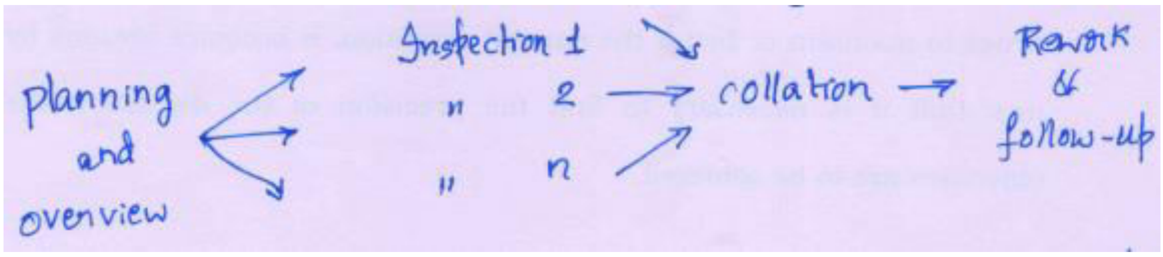
(c) Gilb Inspection: (Gilb & Graham)



(d) Humphrey's Inspection:



(e) N-fold Inspection: The effectiveness of the inspection process can be increased by replicating it.



(f) Reading Techniques: These are steps guiding an inspector to understand the product.

1. Adhoc Method
2. Checklists
3. Scenario based testing
   - Perspective based
   - Usage based
   - Abstract driven
   - Task driven
   - Function point based scenarios

58. Structured Walkthroughs: Less former and less rigorous technique. No rules and no methods exist. Members are: coordinator, presenter/developer, recorder, tester, maintenance oracle, standards bearer, user agent.
59. Tech Reviews: This is intended to evaluate the software in the light of development standards, guidelines, and specifications. Same as walkthrough but includes management.

****

Software Testing Methodologies (R13)

UNIT – 3

Quiz

1. The design, structure and code of the software are studied in ___ testing. [KEY: B]
   (a) Black box
   (b) Glass box
   (c) Mutation
   (d) Regression

2. BBT test cases can be designed ___ WBT test cases, but can be executed only ____them. [KEY: B]
   (a) After, before
   (b) Before, after
   (c) Before, with
   (d) With, after

3. ___coverage is a necessary but not sufficient criterion for logic coverage. [KEY: C]
   (a) Loop
   (b) Branch
   (c) Statement
   (d) Condition

4. ____is based on the control structure of the program. [KEY: C]
   (a) Path testing
   (b) Condition testing
   (c) Basis path testing
   (d) Independent path testing

5. A node with more than one arrow leaving it is called ____node. [KEY: B]
   (a) Junction
   (b) Condition
   (c) Region
   (d) Edge

6. A path through the graph that introduces at least one new edge or node that has not been traversed before is known as____ path. [KEY: D]
   (a) Basis
   (b) New
   (c) Non-traversed
   (d) Independent

7. ______ considers only independent paths. [KEY: B]
   (a) Path testing
   (b) Cyclomatic complexity
   (c) Basis path testing
   (d) None

8. ____provides the number of tests to be executed on the software based on the complexity of the software. [KEY:B]
   (a) Cyclomatic complexity
   (b) Cyclomatic number
   (c) No. of independent paths
   (d) No. of path segments

9. A ____ is a square matrix whose rows and columns equal the no. of nodes in the flow graph. [KEY: D]
   (a) Decision Matrix
   (b) Confusion Matrix
   (c) Connection Matrix
   (d) Graph Matrix

10. A matrix defined with link weights is called ___ matrix. [KEY: C]
    (a) Decision
    (b) Confusion
    (c) Connection
    (d) Graph

11. A path segment for which every node is visited at most once is called ___ segment. [KEY: A]
    (a) Loop-free
    (b) Single path
    (c) Multiple path
    (d) 'du' path

12. Faulty programs are called the ___ of the original program. [KEY: B]
    (a) Defaults
    (b) Mutants
    (c) Slices
    (d) Dices

13. A programmer, in inspection, is also known as ___. [KEY: D]
    (a) Developer
    (b) User
    (c) Moderator
    (d) Producer

14. ____ evaluate the system relative to specification and standards. [KEY: A]
    (a) Technical reviews
    (b) Walkthroughs
    (c) Inspections
    (d) Gilb inspections

15. A reviewer in a walkthrough concentrates on ____. [KEY: C]
    (a) Error rate
    (b) Syntax errors
    (c) Defects
    (d) Typographical errors