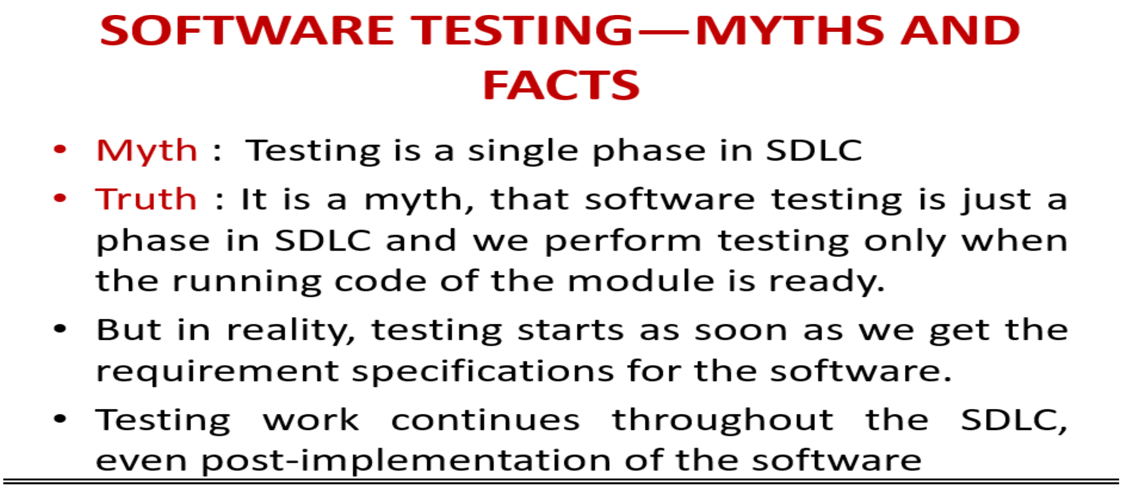
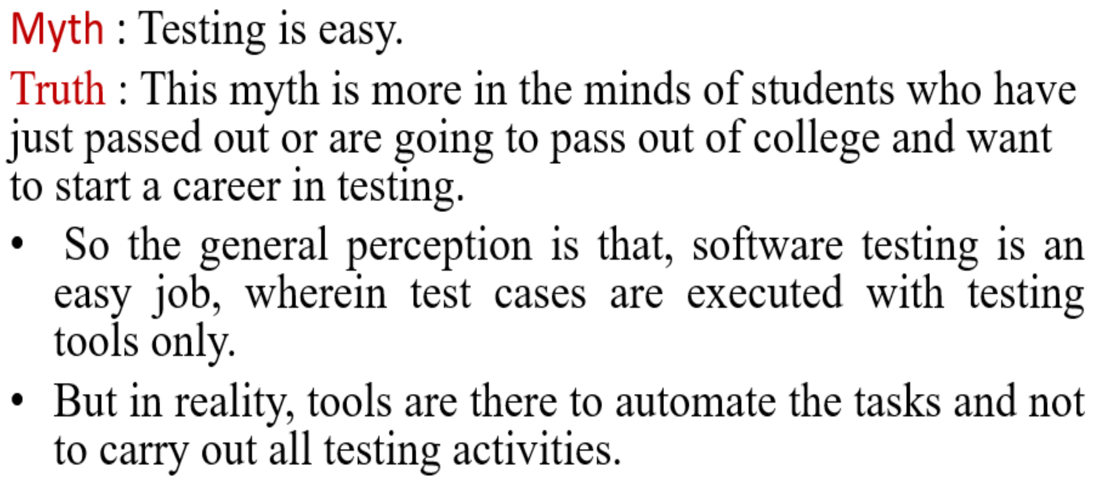
1) Software Testing - Myths and Facts



SOFTWARE TESTING—MYTHS AND FACTS

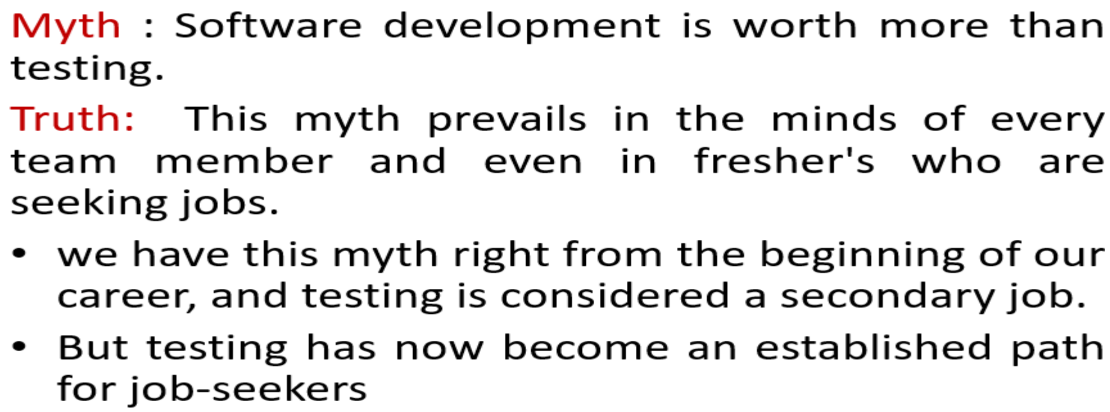
- Myth : Testing is a single phase in SDLC
- Truth : It is a myth, that software testing is just a phase in SDLC and we perform testing only when the running code of the module is ready.
- But in reality, testing starts as soon as we get the requirement specifications for the software.
- Testing work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



Myth : Testing is easy.

Truth : This myth is more in the minds of students who have just passed out or are going to pass out of college and want to start a career in testing.

- So the general perception is that, software testing is an easy job, wherein test cases are executed with testing tools only.
- But in reality, tools are there to automate the tasks and not to carry out all testing activities.



Myth : Software development is worth more than testing.

Truth: This myth prevails in the minds of every team member and even in fresher's who are seeking jobs.

- we have this myth right from the beginning of our career, and testing is considered a secondary job.
- But testing has now become an established path for job-seekers

Myth : Complete testing is possible

Truth : This myth also exists at various levels of the development team.

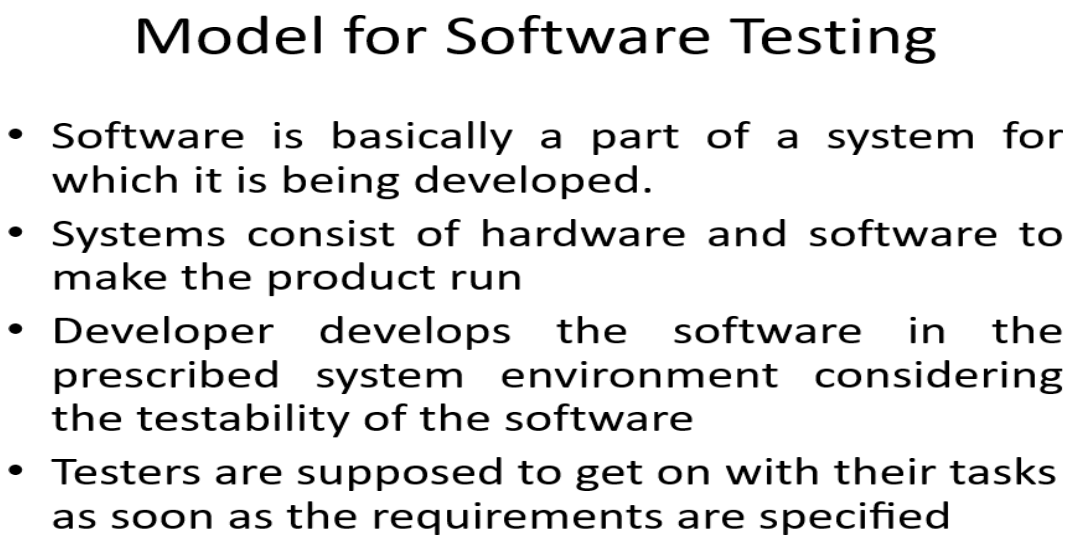
- Complete testing at the surface level assumes that if we are giving all the inputs to the software, then it must be tested for all of them
- But in reality, it is not possible to provide all the possible inputs to test the software, as the input domain of even a small program is too large to test.
- there are many things which cannot be tested completely, as it may take years to do so

- **Myth :** Testing starts after program development. **Truth :** Most of the team members, who are not aware of testing as a process, still feel that testing cannot commence before coding.
- But this is not true, the work of a tester begins as soon as we get the specifications.
- The tester performs testing at the end of every phase of SDLC in the form of verification and plans for the validation testing .

- **Myth** : The purpose of testing is to check the functionality of the software.
- **Truth :** Today, all the testing activities are driven by quality goals.
- Ultimately, the goal of testing is also to ensure quality of the software.
- But quality does not imply checking only the functionalities of all the modules.

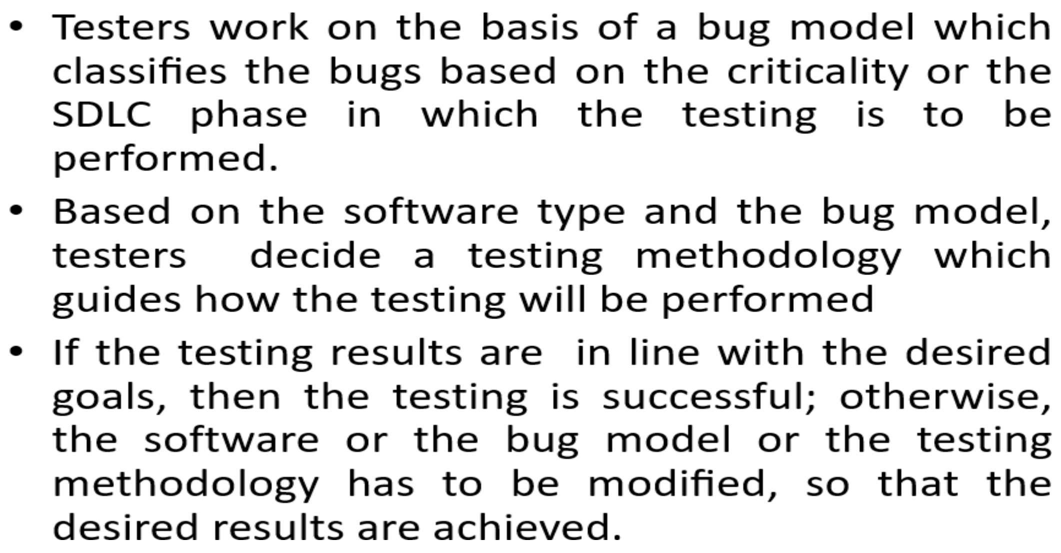
2) Model for Software Testing

Ans :



Model for Software Testing

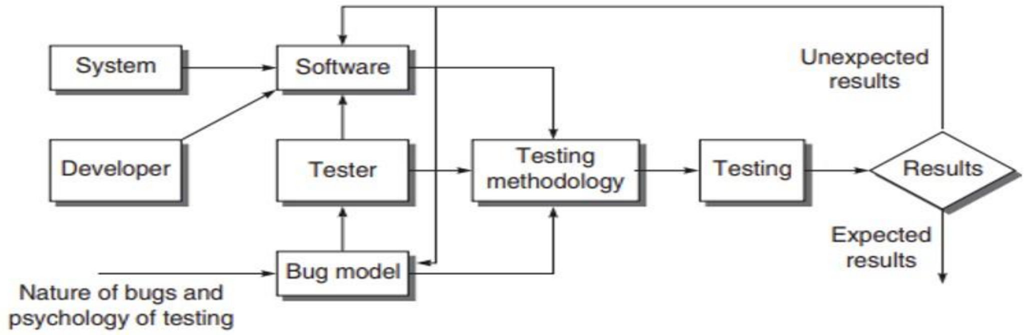
- Software is basically a part of a system for which it is being developed.
- Systems consist of hardware and software to make the product run
- Developer develops the software in the prescribed system environment considering the testability of the software
- Testers are supposed to get on with their tasks as soon as the requirements are specified



- Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed.
- Based on the software type and the bug model, testers decide a testing methodology which guides how the testing will be performed
- If the testing results are in line with the desired goals, then the testing is successful; otherwise, the software or the bug model or the testing methodology has to be modified, so that the desired results are achieved.

SOFTWARE TESTING MODEL

- Click to add text



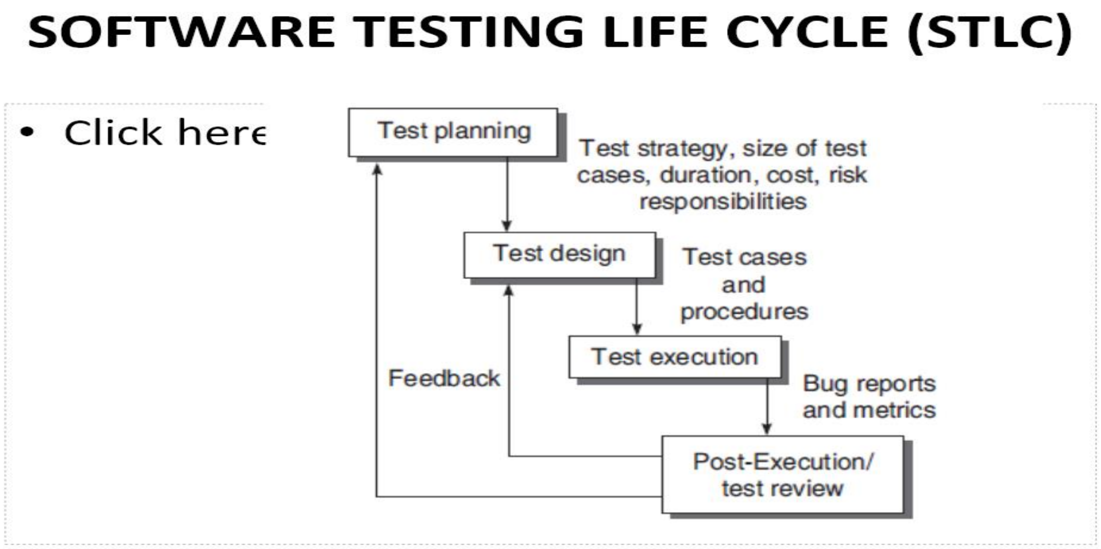
Software and Software Model

- Software is built after analysing the system in the environment
- It is a complex entity which deals with environment, logic, programmer psychology, etc.
- Complex software makes it very difficult to test
- Developers should design and code the software such that it is testable at every point.
- The software to be tested may be modeled such that it is testable, avoiding unnecessary complexities

Bug Model

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

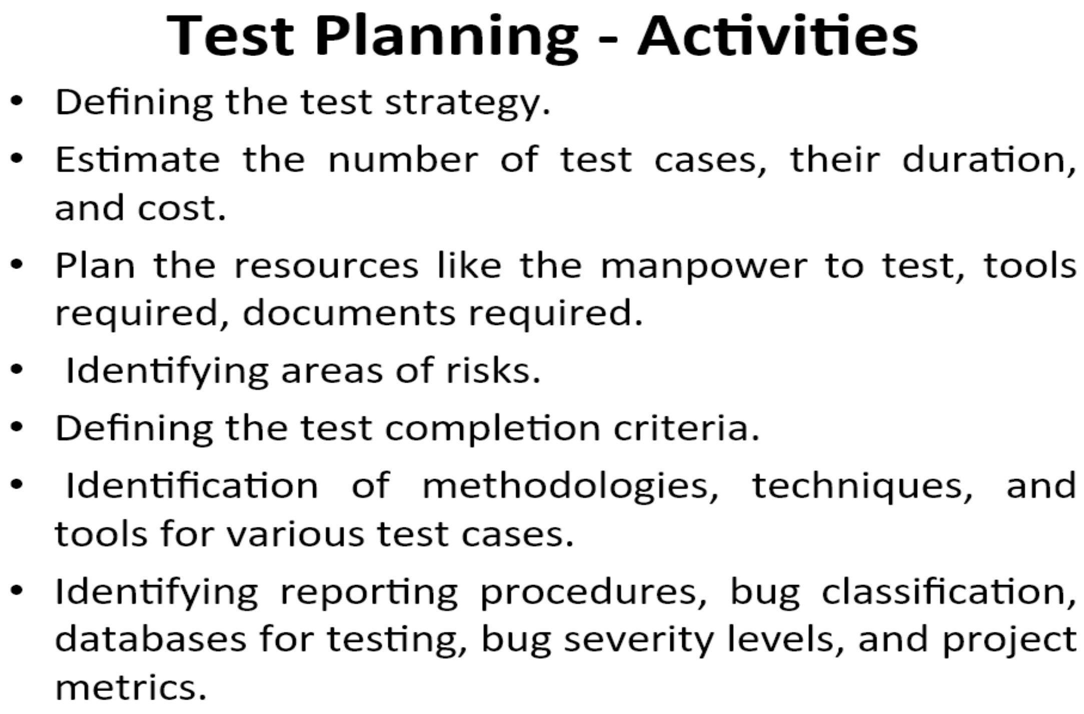
3)software testing life cycle ?
Ans :



SOFTWARE TESTING LIFE CYCLE (STLC)

- Click here

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



Test Planning - Activities

- Defining the test strategy.
- Estimate the number of test cases, their duration, and cost.
- Plan the resources like the manpower to test, tools required, documents required.
- Identifying areas of risks.
- Defining the test completion criteria.
- Identification of methodologies, techniques, and tools for various test cases.
- Identifying reporting procedures, bug classification, databases for testing, bug severity levels, and project metrics.

Test Design

- It is a well-planned process and important

Activities:

Determining the test objectives and their prioritization

Preparing list of items to be tested

Mapping items to test cases **-** A matrix can be created

This matrix will help in:

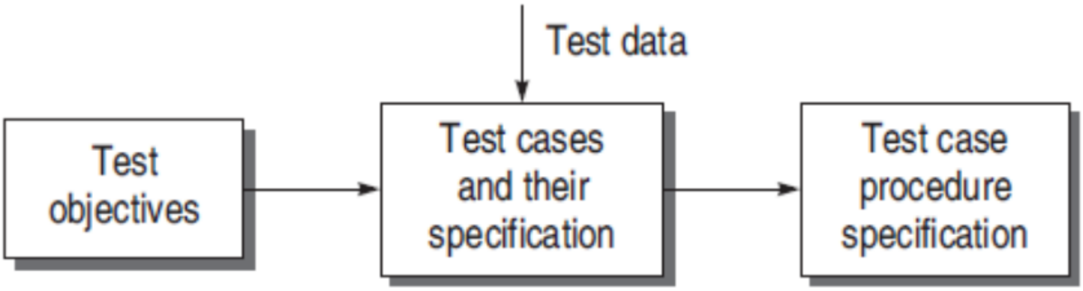
(a) Identifying the major test scenarios.

(b) Identifying and reducing the redundant test cases.

(c) Identifying the absence of a test case for a particular objective and as a result, creating them.

- Designing the test cases demands a prior analysis of the program at functional or structural level

- ***Creating test cases and test data***
- ***Setting up the test environment and supporting tools***
- ***Creating test procedure specification***

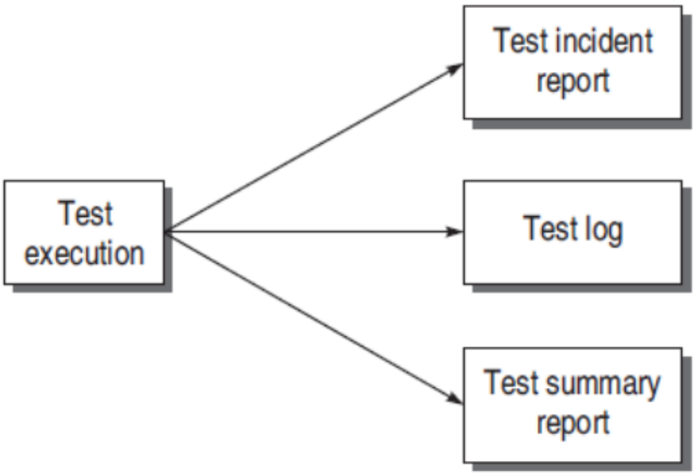

Test data
Test objectives
Test cases and their specification
Test case procedure specification

Test Execution

- Test cases are executed including verification and validation
- Opt for automation or manual execution
- Test results are documented in the test incident reports, test logs, testing status, and test summary reports

Documents in Test Execution

- Click here to add text



Post-Execution/Test Review

- Analyze bug-related issues and get feedback so that maximum number of bugs can be removed

Activities performed by developer

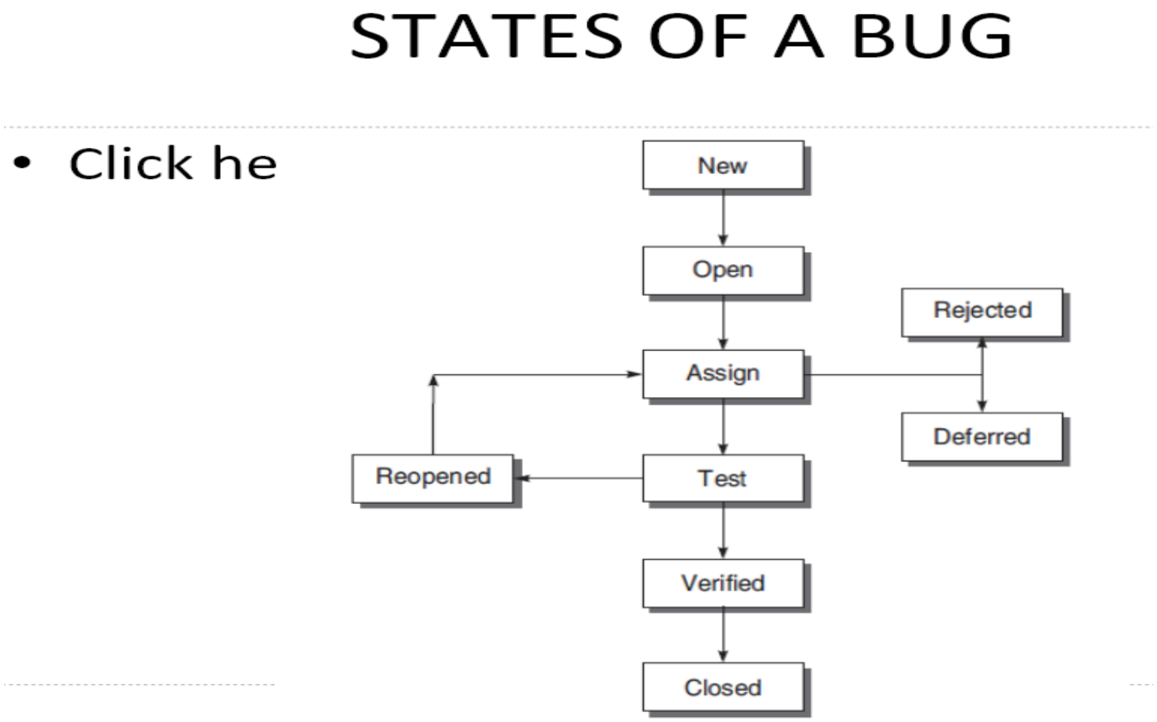
- ***Understanding the bug*** The developer analyses the bug reported and builds an understanding of its whereabouts

- ***Reproducing the bug*** Next, he confirms the bug by reproducing the bug and the failure that exists. This is necessary to cross-check failures. However, some bugs are not reproducible which increases the problems of developers
- ***Analyzing the nature and cause of the bug*** **-** A developer starts debugging its symptoms and tracks back to the actual location of the error in the design

==

5)explain different states of bug ?

ANS :



STATES OF A BUG

- Click he

STATES OF A BUG

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

- ***Deferred*** The developer who has been assigned to fi x the bug will check its validity and priority. If the priority of the reported bug is not high or there is not sufficient time to test it or the bug does not have any adverse effect on the
- software, then the bug is changed to deferred state which implies the bug is expected to be fixed in next releases.
- ***Rejected*** It may be possible that the developer rejects the bug after checking its validity, as it is not a genuine one.

- ***Test*** After fixing the valid bug, the developer sends it back to the testing team for next round of checking. Before releasing to the testing team, the developer changes the bug's state to 'TEST'. It specifies that the bug has been fixed by the development team but not tested and is released to the testing team.
- ***Verified/fixed*** The tester tests the software and verifies whether the reported
- bug is fixed or not. After verifying, the developer approves that the bug is fixed and changes the status to 'VERIFIED'

- ***Reopened*** If the bug is still there even after fixing it, the tester changes its status to ‘REOPENED’. The bug traverses the life cycle once again.
- In another case, a bug which has been closed earlier may be reopened if it appears again. In this case, the status will be REOPENED instead of OPEN.

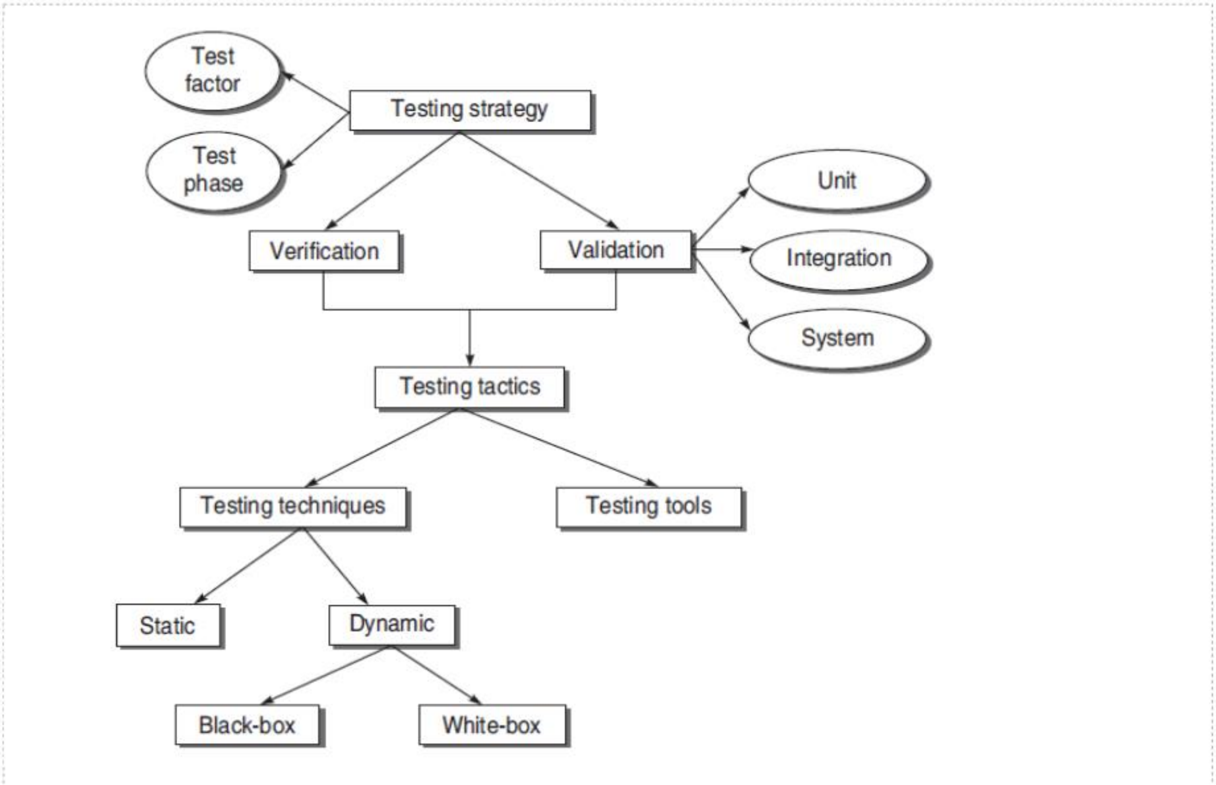
- ***Closed*** Once the tester and other team members are confirmed that the bug is completely eliminated, they change its status to ‘CLOSED

---

6)explain software testing methodology with neat diagram ?

Ans :

SOFTWARE TESTING METHODOLOGY



Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved, as shown in Fig. 2.11. All the terms related to software testing methodology and a complete testing strategy is discussed in this section

SOFTWARE TESTING STRATEGY

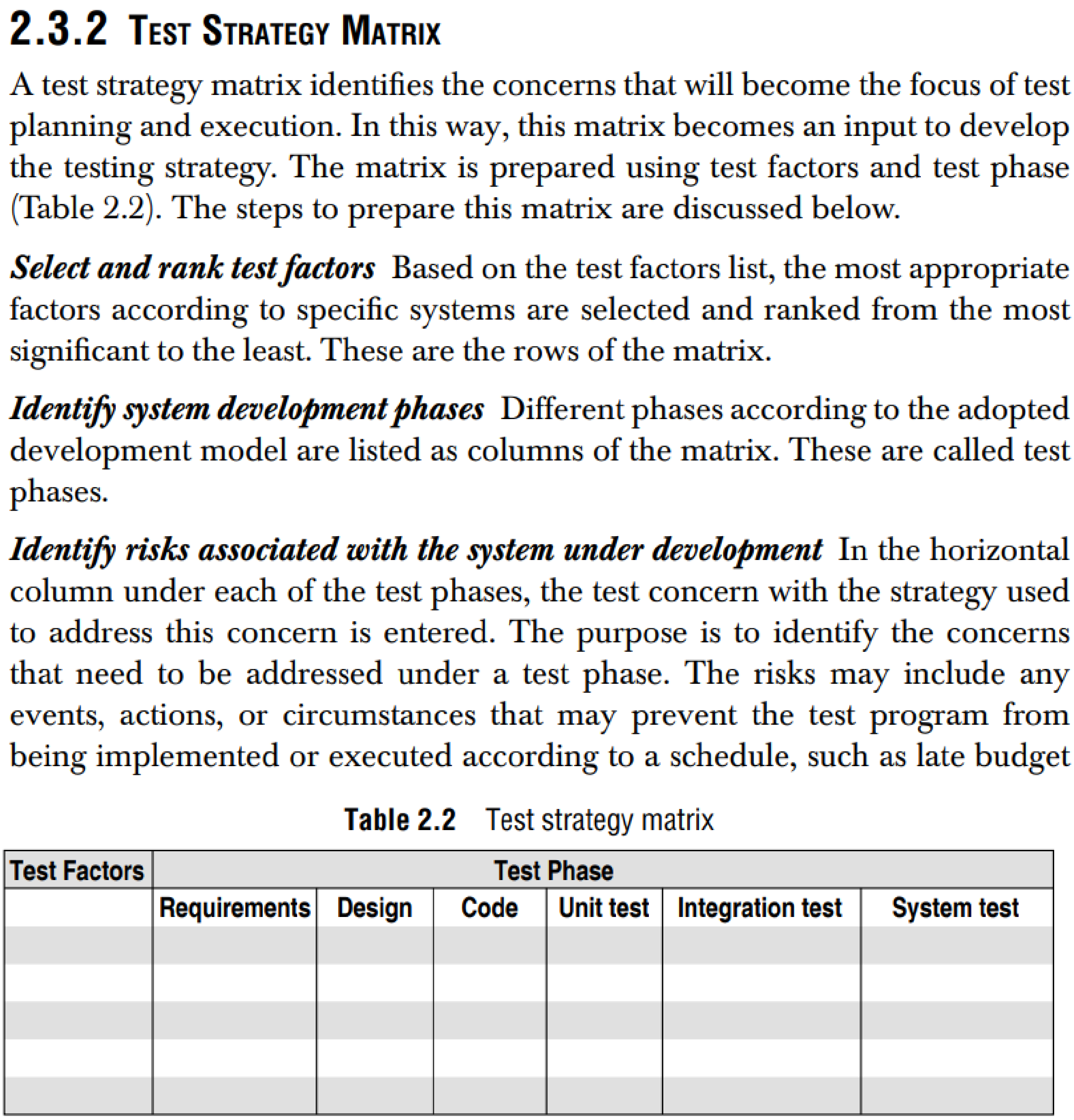
Testing strategy is the planning of the whole testing process into a well-planned series of steps. In other words, strategy provides a roadmap that includes very specific activities that must be performed by the test team in order to achieve a specific goal.

Test Factors

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

Test Phase

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



2.3.2 Test Strategy Matrix

A test strategy matrix identifies the concerns that will become the focus of test planning and execution. In this way, this matrix becomes an input to develop the testing strategy. The matrix is prepared using test factors and test phase (Table 2.2). The steps to prepare this matrix are discussed below.

Select and rank test factors Based on the test factors list, the most appropriate factors according to specific systems are selected and ranked from the most significant to the least. These are the rows of the matrix.

Identify system development phases Different phases according to the adopted development model are listed as columns of the matrix. These are called test phases.

Identify risks associated with the system under development In the horizontal column under each of the test phases, the test concern with the strategy used to address this concern is entered. The purpose is to identify the concerns that need to be addressed under a test phase. The risks may include any events, actions, or circumstances that may prevent the test program from being implemented or executed according to a schedule, such as late budget

Table 2.2 Test strategy matrix

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

7) Discuss V-testing and Expanded V-testing model with diagram ?

Ans)

V-Testing Life Cycle Model

In V-testing concept [4], as the development team attempts to implement the software, the testing team concurrently starts checking the software. When the project starts, both the system development and the system test process begin. The team that is developing the system begins the system development process and the team that is conducting the system test begins planning the system test process, as shown in Fig. 2.12. Both teams start at the same point using the same information. If there are risks, the tester develops a process to minimize or eliminate them.

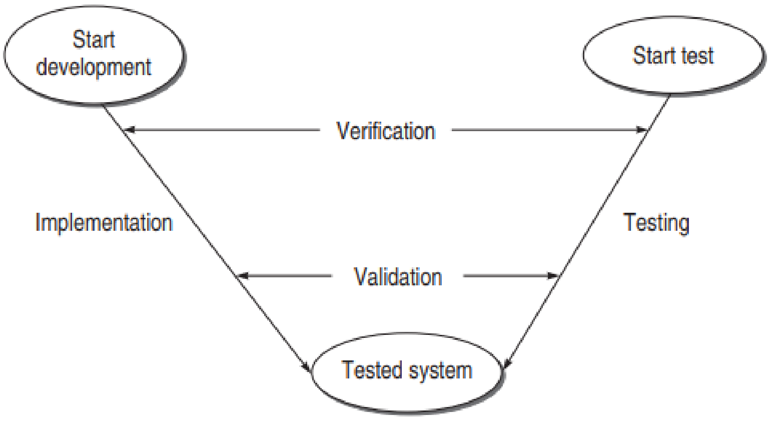


Figure 2.12 V-testing model

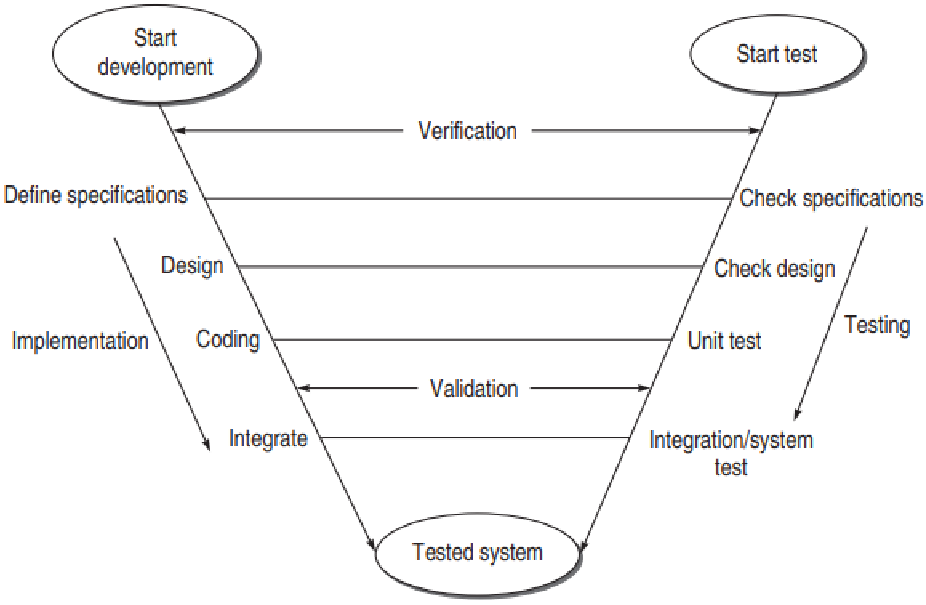
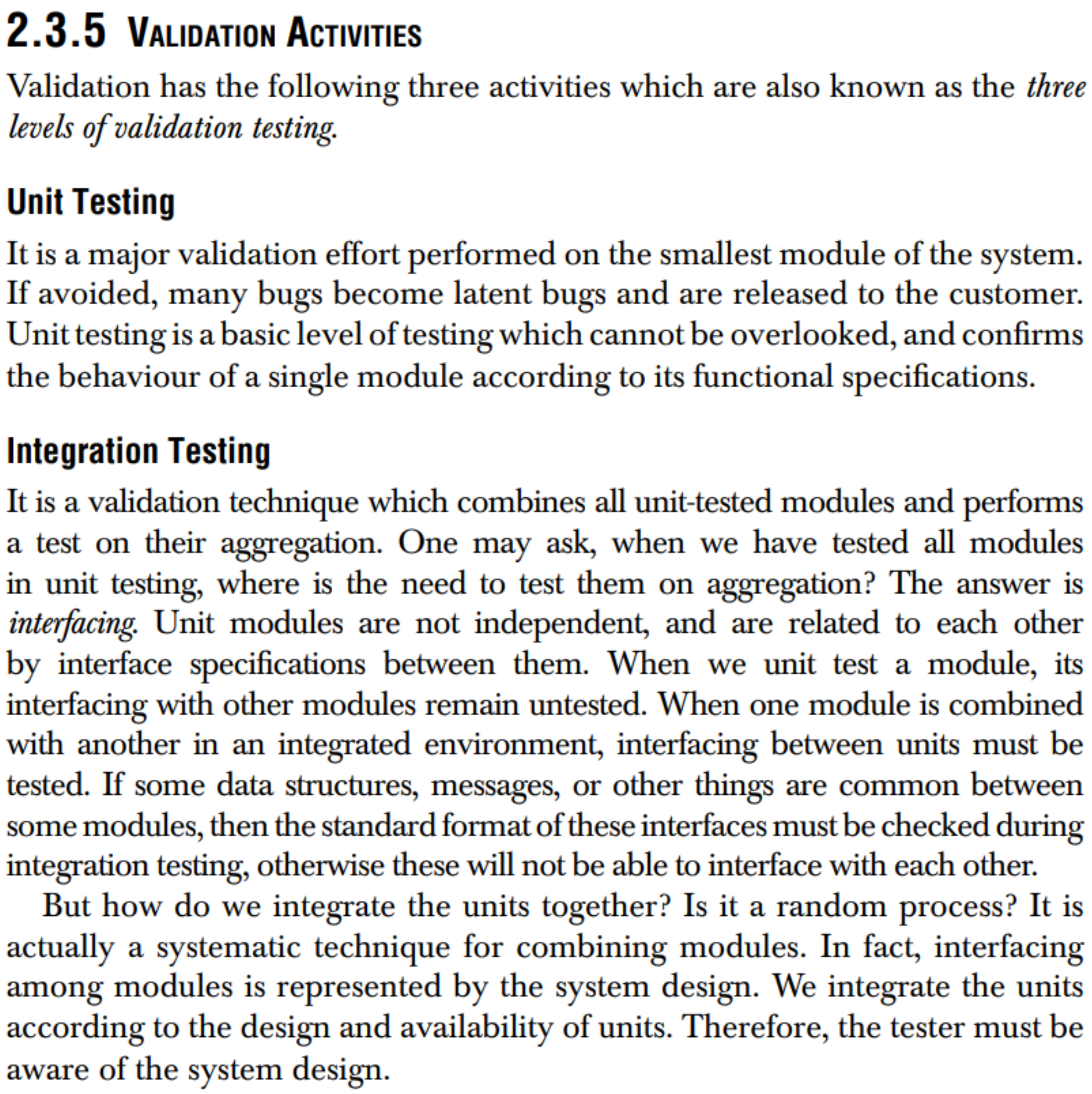


Figure 2.13 Expanded V-testing model



2.3.5 Validation Activities

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

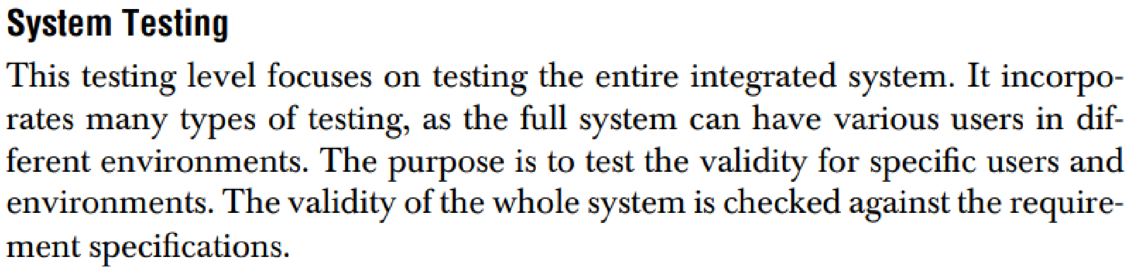
Unit Testing

It is a major validation effort performed on the smallest module of the system. If avoided, many bugs become latent bugs and are released to the customer. Unit testing is a basic level of testing which cannot be overlooked, and confirms the behaviour of a single module according to its functional specifications.

Integration Testing

It is a validation technique which combines all unit-tested modules and performs a test on their aggregation. One may ask, when we have tested all modules in unit testing, where is the need to test them on aggregation? The answer is *interfacing*. Unit modules are not independent, and are related to each other by interface specifications between them. When we unit test a module, its interfacing with other modules remain untested. When one module is combined with another in an integrated environment, interfacing between units must be tested. If some data structures, messages, or other things are common between some modules, then the standard format of these interfaces must be checked during integration testing, otherwise these will not be able to interface with each other.

But how do we integrate the units together? Is it a random process? It is actually a systematic technique for combining modules. In fact, interfacing among modules is represented by the system design. We integrate the units according to the design and availability of units. Therefore, the tester must be aware of the system design.



System Testing

This testing level focuses on testing the entire integrated system. It incorporates many types of testing, as the full system can have various users in different environments. The purpose is to test the validity for specific users and environments. The validity of the whole system is checked against the requirement specifications.

9. Black box testing techniques - BVA, Equivalence partitioning, State table based testing, decision table based testing ?

BVA :

4.1 Boundary Value Analysis (BVA)

An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. BVA technique addresses this issue. With the experience of testing team, it has been observed that test cases designed with boundary input values have a high chance to find errors. It means that most of the failures crop up due to boundary values.

BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain. For example, if *A* is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254). Similarly, *B* is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101), as shown in Fig. 4.2.

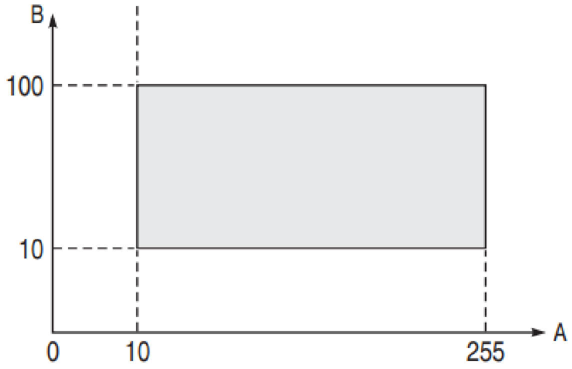


Figure 4.2 Boundary value analysis

4.1.1 Boundary Value Checking (BVC)

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

The variable at its extreme value can be selected at:

(a) Minimum value (Min)

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

(c) Maximum value (Max)

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

Let us take the example of two variables, *A* and *B*. If we consider all the above combinations with nominal values, then following test cases (see Fig. 4.3) can be designed:

1. $A_{\text{nom}}, B_{\text{min}}$
2. $A_{\text{nom}}, B_{\text{min+}}$
3. $A_{\text{nom}}, B_{\text{max}}$
4. $A_{\text{nom}}, B_{\text{max–}}$
5. $A_{\text{min}}, B_{\text{nom}}$
6. $A_{\text{min+}}, B_{\text{nom}}$
7. $A_{\text{max}}, B_{\text{nom}}$
8. $A_{\text{max–}}, B_{\text{nom}}$
9. $A_{\text{nom}}, B_{\text{nom}}$

4.1.2 Robustness Testing Method

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

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

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

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

10. $A_{\text{max+}}, B_{\text{nom}}$
11. $A_{\text{min–}}, B_{\text{nom}}$
12. $A_{\text{nom}}, B_{\text{max+}}$
13. $A_{\text{nom}}, B_{\text{min–}}$

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

4.1.3 Worst-Case Testing Method

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

Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

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

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

BVA is applicable when the module to be tested is a function of several independent variables. This method becomes important for physical quantities where boundary condition checking is crucial. For example, systems having requirements of minimum and maximum temperature, pressure or speed, etc. However, it is not useful for Boolean variables.

Sums :

Example 4.1,4.2,4.3,4.4

4.2 Equivalence Class Testing

We know that the input domain for testing is too large to test every input. So we can divide or partition the input domain based on a common feature or

a class of data. Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called *equivalence classes* are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed. It means only one test case in the equivalence class will be sufficient to find errors. This test case will have a representative value of a class which is equivalent to a test case containing any other value in the same class. If one test case in an equivalence class detects a bug, all other test cases in that class have the same probability of finding bugs. Therefore, instead of taking every value in one domain, only one test case is chosen from one class. In this way, testing covers the whole input domain, thereby reduces the total number of test cases. In fact, it is an attempt to get a good *hit rate* to find maximum errors with the smallest number of test cases.

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

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

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

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

1. Identify equivalence classes
2. Design test cases

4.2.1 Identification of Equivalent Classes

How do we partition the whole input domain? Different equivalence classes are formed by grouping inputs for which the behaviour pattern of the module is similar. The rationale of forming equivalence classes like this is the assumption that if the specifications require exactly the same behaviour for each element in a class of values, then the program is likely to be constructed such that it either succeeds or fails for each value in that class. For example, the specifications of a module that determines the absolute value for integers specify different behaviour patterns for positive and negative integers. In this case, we will form two classes: one consisting of positive integers and another consisting of negative integers [14].

Two types of classes can always be identified as discussed below:

Valid equivalence classes These classes consider valid inputs to the program.

Invalid equivalence classes One must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program, as shown in Fig. 4.4.

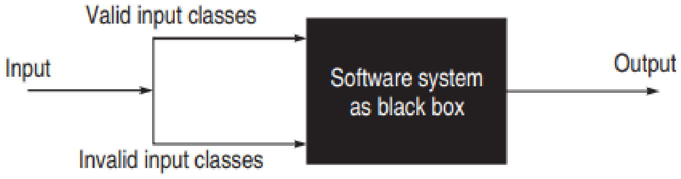


Figure 4.4 Equivalence classes

4.2.2 Identifying the Test Cases

A few guidelines are given below to identify test cases through generated equivalence classes:

- Assign a unique identification number to each equivalence class.
- Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.

- Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases. The reason that invalid cases are covered by individual test cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states ‘Enter type of toys (Automatic, Mechanical, Soft toy) and amount (1–10000)’, the test case [ABC 0] expresses two error (invalid inputs) conditions (invalid toy type and invalid amount) will not demonstrate the invalid amount test case, hence the program may produce an output ‘ABC is unknown toy type’ and not bother to examine the remainder of the input.

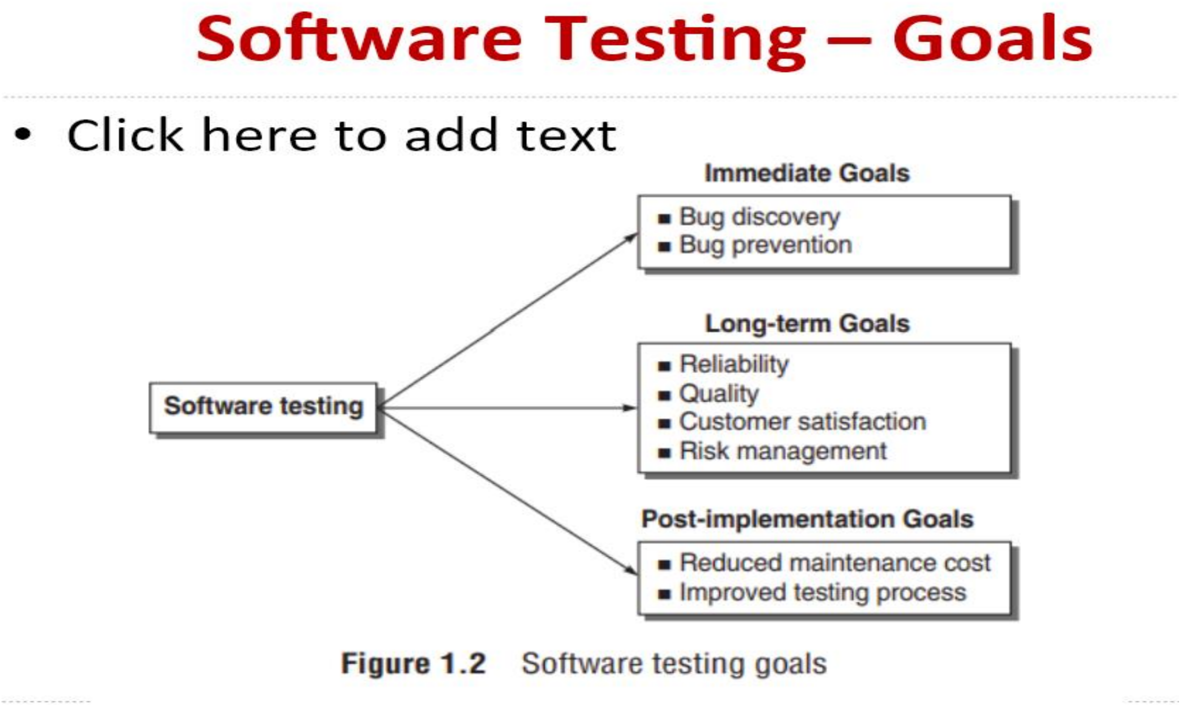
Sums :

Example 4.5,4.6,4.7

1 MARK :

1) What are the goals of software testing?

Ans :



Software Testing – Goals

- Click here to add text

Immediate Goals

- Bug discovery
- Bug prevention

Long-term Goals

- Reliability
- Quality
- Customer satisfaction
- Risk management

Post-implementation Goals

- Reduced maintenance cost
- Improved testing process

Figure 1.2 Software testing goals

2) . Define software testing ?

ans :

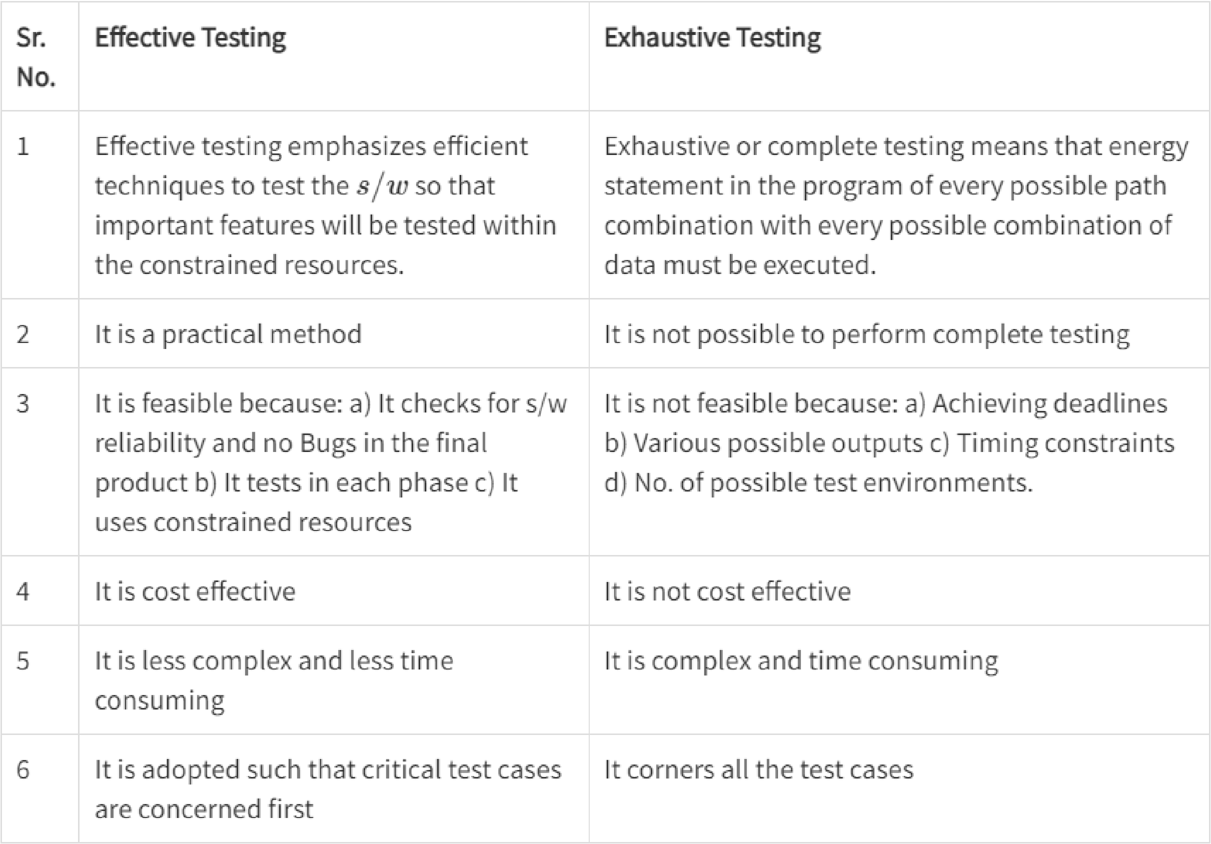
Testing is the process of executing a program with the intent of finding errors.

A successful test is one that uncovers an as-yet-undiscovered error.

Testing can show the presence of bugs but never their absence.

3) Differentiate between effective testing and exhaustive software testing ?

Ans :



Sr. No.	Effective Testing	Exhaustive Testing
1	Effective testing emphasizes efficient techniques to test the s/w so that important features will be tested within the constrained resources.	Exhaustive or complete testing means that energy statement in the program of every possible path combination with every possible combination of data must be executed.
2	It is a practical method	It is not possible to perform complete testing
3	It is feasible because: a) It checks for s/w reliability and no Bugs in the final product b) It tests in each phase c) It uses constrained resources	It is not feasible because: a) Achieving deadlines b) Various possible outputs c) Timing constraints d) No. of possible test environments.
4	It is cost effective	It is not cost effective
5	It is less complex and less time consuming	It is complex and time consuming
6	It is adopted such that critical test cases are concerned first	It corners all the test cases

4)Define Bug, Failure, Error

Ans :

Failure:

inability of a system or component to perform a required function according to its specification when results or behavior of the system under test are different as compared to specified expectations, then failure exists.

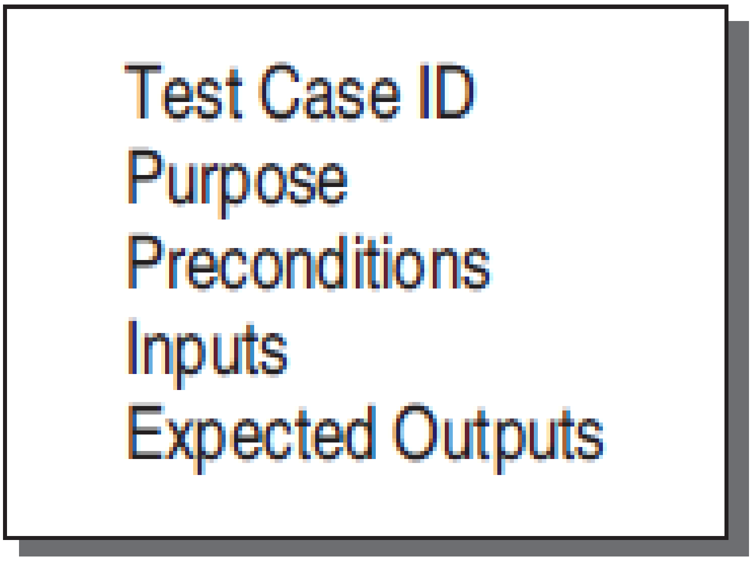
Error :

Error causes a bug and the bug in turn causes failures It might be a typographical error, a misleading of a specification, a misunderstanding of what a subroutine does.

Bug :

A software bug is an error, flaw or fault in the design, development, or operation of computer software that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.

5) Give the template of test case



Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs

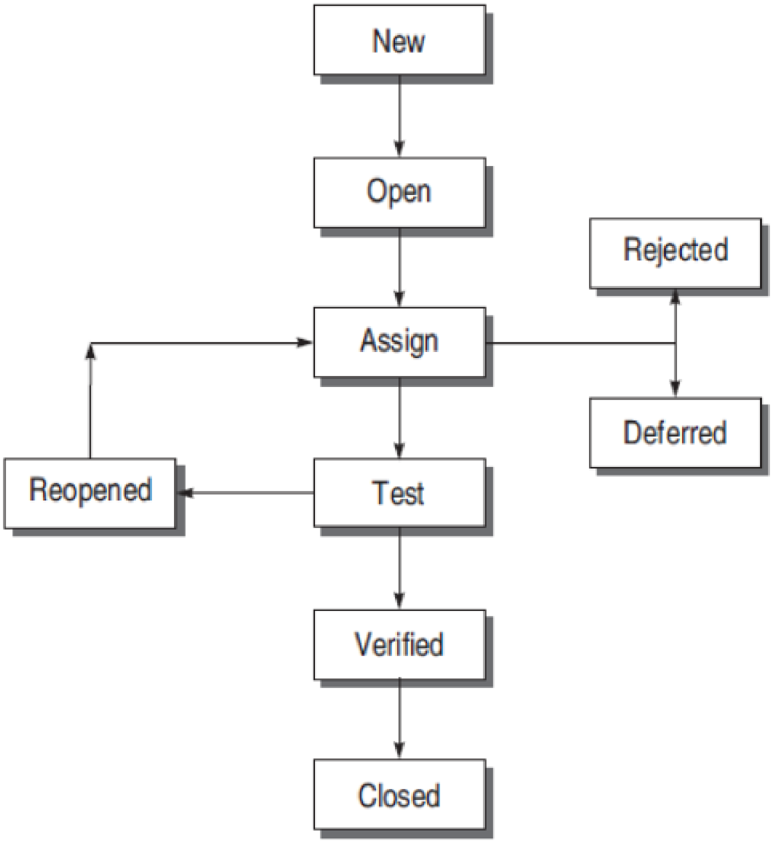
6) What is the significance of test strategy matrix?

Ans :

A test strategy matrix identifies the concerns that will become the focus of test planning and execution.

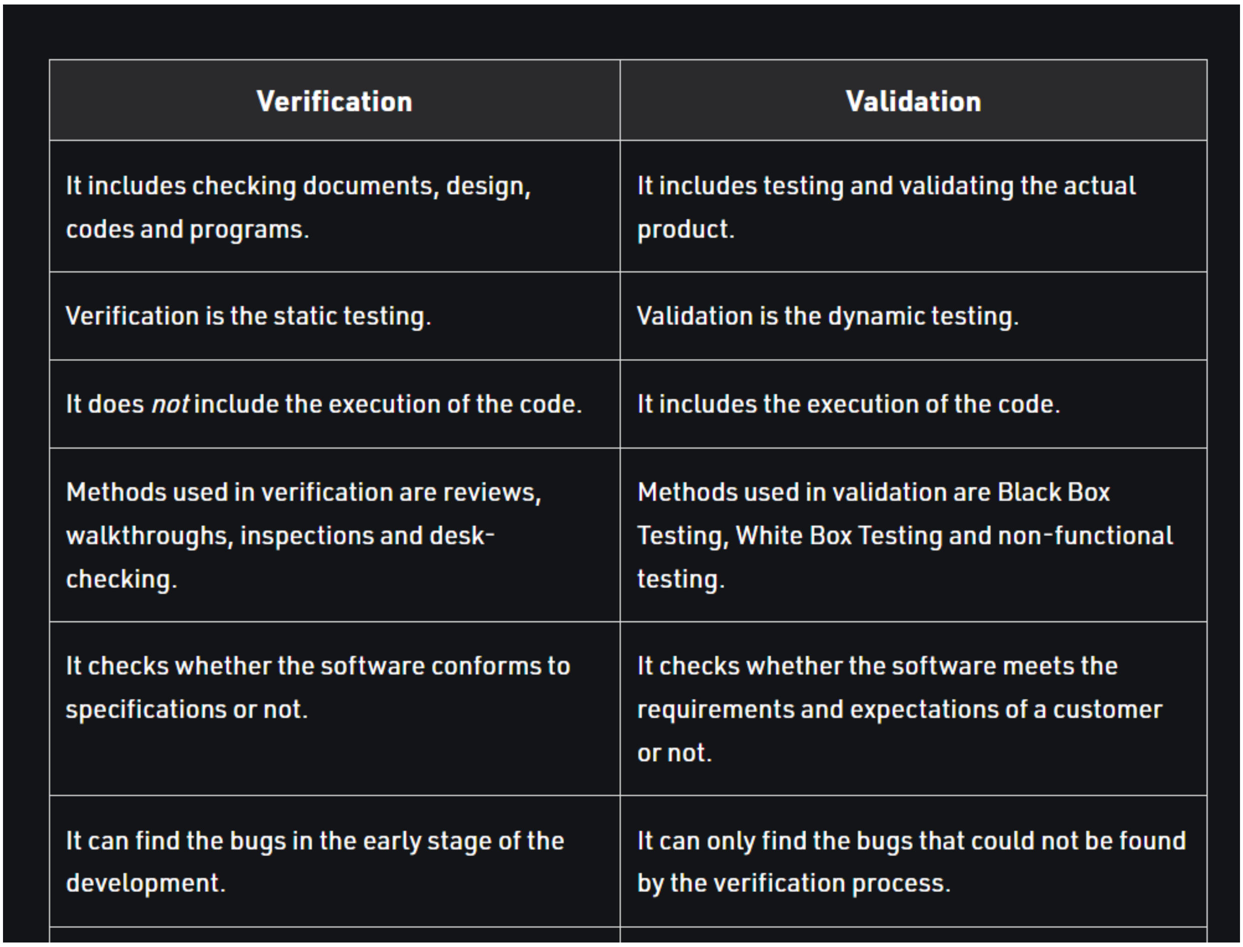
7) How many states does a bug have?

Ans :



In above 9 states are their

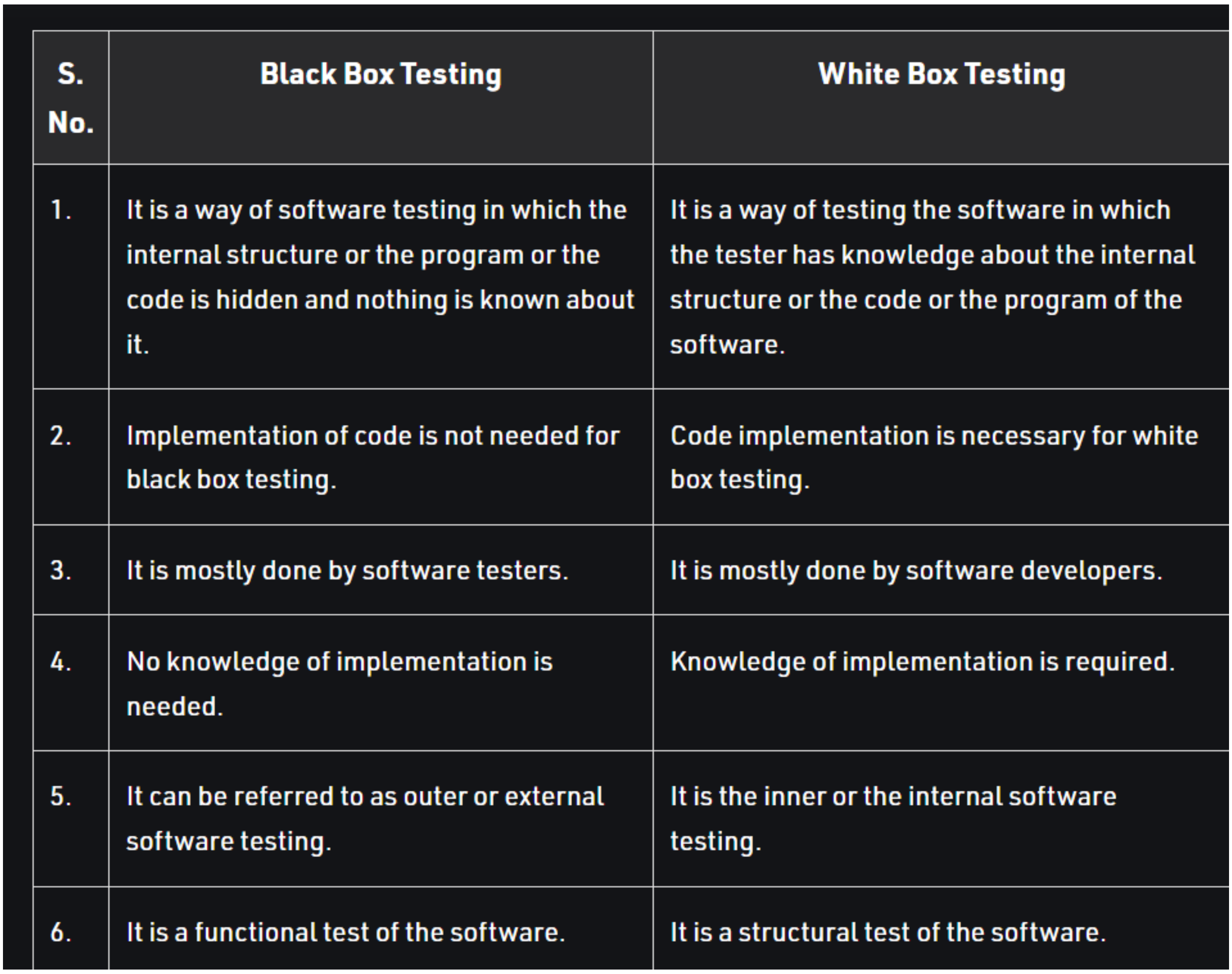
8) Differentiate between verification and validation.



Verification	Validation
It includes checking documents, design, codes and programs.	It includes testing and validating the actual product.
Verification is the static testing.	Validation is the dynamic testing.
It does *not* include the execution of the code.	It includes the execution of the code.
Methods used in verification are reviews, walkthroughs, inspections and desk-checking.	Methods used in validation are Black Box Testing, White Box Testing and non-functional testing.
It checks whether the software conforms to specifications or not.	It checks whether the software meets the requirements and expectations of a customer or not.
It can find the bugs in the early stage of the development.	It can only find the bugs that could not be found by the verification process.

9) what is the difference between black box and white box testing?

Ans :



S. No.	Black Box Testing	White Box Testing
1.	It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
2.	Implementation of code is not needed for black box testing.	Code implementation is necessary for white box testing.
3.	It is mostly done by software testers.	It is mostly done by software developers.
4.	No knowledge of implementation is needed.	Knowledge of implementation is required.
5.	It can be referred to as outer or external software testing.	It is the inner or the internal software testing.
6.	It is a functional test of the software.	It is a structural test of the software.

10) What is traceability? what is the difference between Backward traceability and Forward traceability?

Ans :

Traceability in software testing is the ability to trace tests forward and backward through the development lifecycle. Test cases are traced forward to test runs. And test runs are traced forward to issues that need to be fixed (or are traced forward to a passed test case).

Mapping take place from from requirements to end products is **Forward Traceability**
Mapping take place from end product back to requirements is **Backward Traceability**

Using both the Forward and Backward Traceability is called **Bidirectional Traceability**. When the requirements are managed well to testcases and testcases to defects and vice versa. This helps nothing is missed in testing process....Bidirectional traceability needs to be implemented both forward and backward i.e., from requirements to end.

Forward traceability maps requirements to test cases.

Backward traceability maps test cases to requirements

11) what are the steps to be performed for equivalence partitioning testing?

Ans :
1. Identify equivalence classes
2. Design test cases

12) What is a state graph?

Ans :

State Graphs provide framework for a model testing, where a State Graph is executed or simulated with event sequences as test cases, before starting the actual implementation phase. State Graphs specify system specification and support for testing the system implementation against the system specification.

13) What are the components of state table?

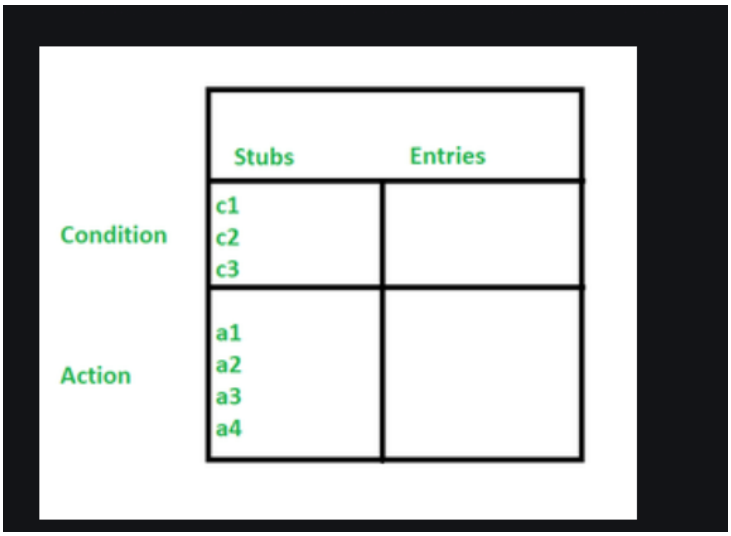
Ans :

In a State Table, all the valid states are listed on the left side of the table, and the events that cause them on the top.

Each cell represents the state system will move to when the corresponding event occurs.

14) Give the structure of decision table.

And :



	Stubs	Entries
Condition	c1 c2 c3	
Action	a1 a2 a3 a4	