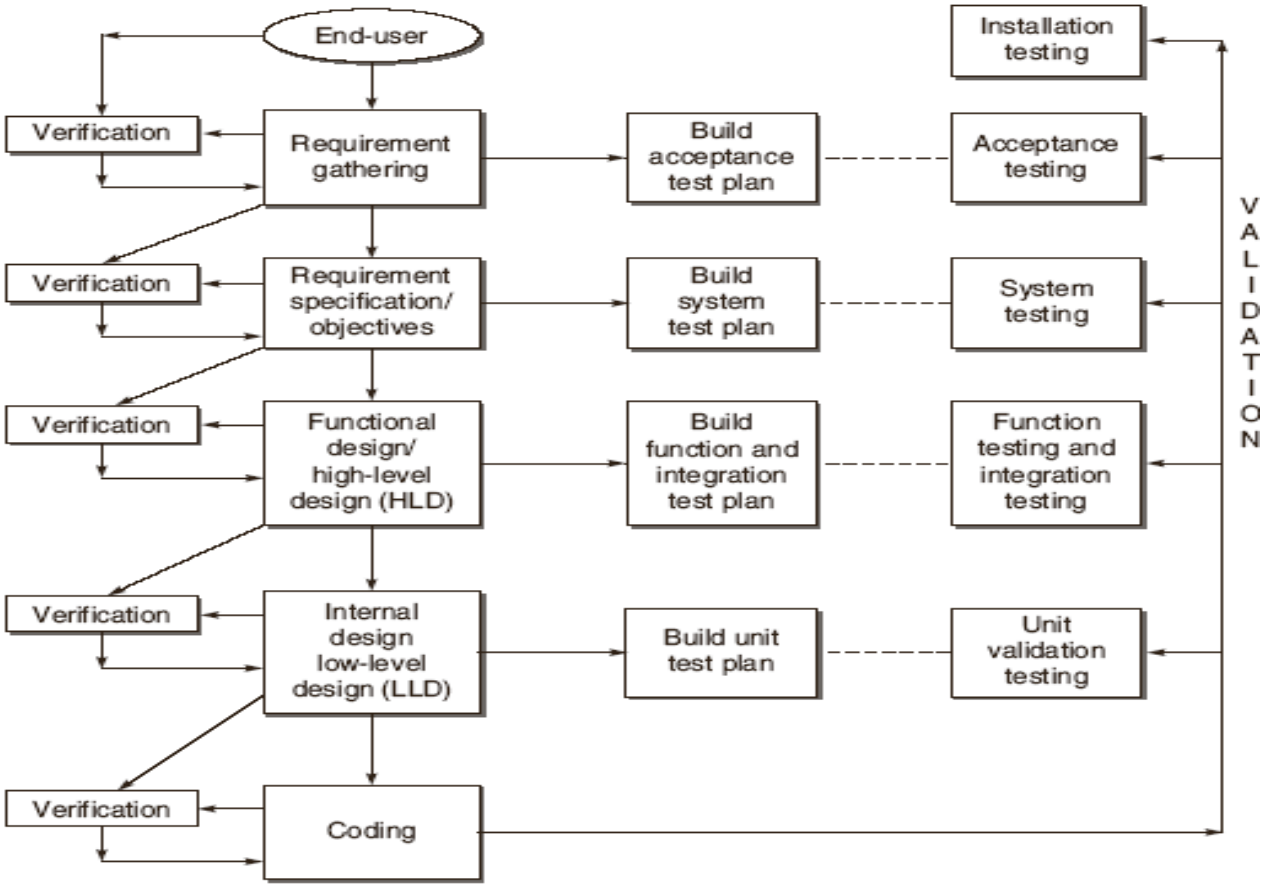
STANNS COLLEGE OF ENGINEERING &TECHNOLOGY
SOFTWARE TESTING METHODOLOGIES
Unit – II

Verification and Validation: Verification & Validation Activities, Verification, Verification of Requirements, High level and low level designs, How to verify code, Validation

Dynamic Testing I: Black Box testing techniques: Boundary Value Analysis, Equivalence class Testing, State Table based testing, Decision table based testing, Cause-Effect Graphing based testing, Error guessing

Verification and Validation

Verification & Validation Activities (V&V):



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

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

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

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

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

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

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

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

--Am I building the product right?

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

--It is a Low level activity

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

Advantages of Software Verification :

--Verification helps in lowering down the count of the defect in the later stages of development. --Verifying the product at the starting phase of the development will help in understanding the product in a better way. --
It reduces the chances of failures in the software application or product. --It helps in building the product as per the customer specifications and needs.

The goals of verification are:

--Everything must be verified.

--Results of the verification may not be binary.

--Even implicit qualities must be verified.

Verification of Requirements:

In this type of verification, all the requirements gathered from the users viewpoint sre verified. For this purpose, an acceptance criterion is preprared. An acceptance criterion defines the goals and requirments of the proposed system and acceptance limits for each of the goals and requirements.
The testers works in parallel by performing the following two tasks:
1. The tester reviews the acceptance criteria in terms of its completeness, clarity and testability.
2. The tester prepares the Acceptance Test Plan which is refered at the time of Acceptance Testing.

Verification of Objectives: After gathering of objectives specific objectives are prepared consedering every specification. These objectives are prpepared in a document called SRS. In this activity the tester performs two parallel activities:
1. The tester verifies all the objectives mentioned in SRS.
2. The tester also prepares the System Test Plan which is based on SRS.

How to verify Requirements and Objectives:

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

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

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

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

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

Consistency: Consistency refers to internal consistency. If an SRS does not agree with some higher-

level document, such as a system requirements specification, then it is a violation of correctness. An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict. The three types of likely conflicts in an SRS are as follows:

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

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

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

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

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

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

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

Verification of High Level Design:

Like the verification of requirements, the tester is responsible for two parallel activities in this place as well:
1. The tester verifies the high level design. Since the system has been decomposed in a number of sub-system or componens, the tester should verify the functionality of these components and their interfaces. Every requirement in SRS should map the design.
2.The tester also prepares a Function Test Plan which is based on the SRS. This plan will be referenced at the time of Fucntion Testing.

How to verify High Level Design:

Verification of Data Design

--Check whether sizes of data structure have been estimated appropriately.

--Check the provisions of overflow in a data structure.

--Check the consistency of data formats with the requirements.

--Check whether data usage is consistent with its declaration.

--Check the relationships among data objects in data dictionary.

--Check the consistency of databases and data warehouses with requirements in SRS.

Verification of Architectural Design

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

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

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

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

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

--Coupling and Module Cohesion.

Verification of User-Interface Design

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

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

--Check all the interfaces between human and computer.

--Check all the above interfaces for their consistency.

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

--Help Facility error messages and warnings.

Verification of Low Level Design (LLD):

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

--The tester verifies the LLD. The details and logic of each module is verified such that the high-level and low-level abstractions are consistent.

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

How to verify Low Level Design (LLD):

--Verify the SRS of each module.

--Verify the SDD of each module.

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

How to verify Code:

Coding is the process of converting LLD specifications into a specific language. This is the last phase when we get tthe operational software with the source code. Since LLD is converted into source code using some lanuguage, there is a possibility of deviation from the LLD. Therefore, the code must also be verified. The points against which the code must be verified are:

--Check that every design specification in HLD and LLD has been coded using traceability matrix.

--Examine the code against a language specification checklist.

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

--Misunderstood or incorrect Arithmetic precedence.

--Mixed mode operations

--Incorrect initialization

--Precision Inaccuracy

--Incorrect symbolic representation of an expression

--Different data types

--Improper or nonexistent loop termination

--Failure to exit.

Validation:

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

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

Validation Activities:
The Validation activities are divided into ***Validation Test Plan*** and ***Validaion Test Execution*** which are described as follows:

Validation Test Plan*:* It starts as soon as the first output of SDLC i.e the SRS is prepared. In every phase, the tester performs two parallel activities - Verification and Validation. For preparing a vlaidation test plan, testers must follow the following points:
--Testers must understand the current SDLC phase.
--Testers must study the relevant documents in the corresponding SDLC phase.
--On the basis of the understanding of SDLC phase and related documents, testers must prpepare the related test plans which are used at the time of validation testing.

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

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

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

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

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

***Unit Test Plan*:** This document describes the Test Plan in other words how the tests will be carried out. This will typically include the list of things to be Tested, Roles and Responsibilities, prerequisites to begin Testing, Test Environment, Assumptions, what to do after a test is successfully carried out, what to do if test fails, Glossary and so on.

Validaion Test Execution:

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

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

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

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

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

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

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

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

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

testing is to establish confidence in the system. Acceptance testing is most often focused on a validation type testing.

Dynamic Testing I

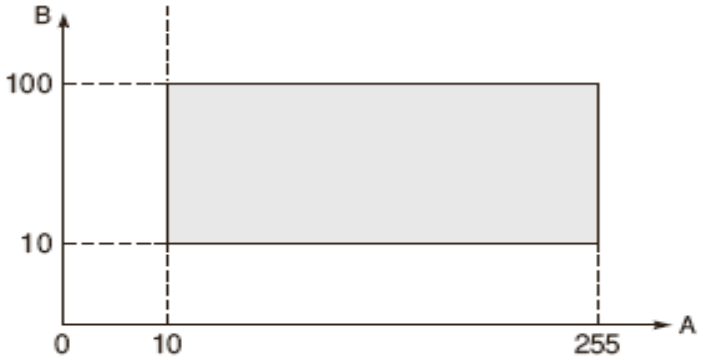
Black Box testing techniques: Boundary Value Analysis, Equivalence class Testing, State Table based testing, Decision table based testing, Cause-Effect Graphing based testing, Error guessing

Boundary Value Analysis:

Boundary value analysis(BVA) is based on testing at the boundaries between partitions.Here we have both valid boundaries (in the valid partitions) and invalid boundaries (in the invalid partitions). BVA offers several methods to design test cases:

-->Boundary Value Checking
-->Robustness Testing Method
-->Worst Case Testing Method.

Example 1:



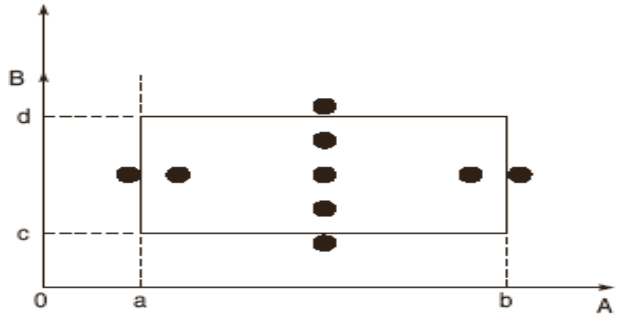
For example,if A is an integer between 10 to 255, and B is an integer between tthe range of 10 to 100 , then different methods of using BVA are as follows:

Boundary Value Checking:

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:

--Minimum value (Min)
--Value just above the minimum value (Min+)
--Maximum value (Max)
--Value just below the maximum value (Max-)

Anom, Bmin
Anom, Bmin+
Anom, Bmax
Anom, Bmax-
Amin, Bnom
Amin+, Bnom
Amax, Bnom
Amax-, Bnom
Anom, Bnom



Robustness Testing Method: In this method, we are going to add another values to the Boundary Cheking Values, they are:

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

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

Amax+, Bnom
Amin-, Bnom
Anom, Bmax+
Anom, Bmin-

Worst Case Testing Method:

We mcan again extend the concept of BVC by assuming more than one variable on the boundary. It is called *Worst Case testing method:*

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-}

Example 2:

A program reads an integer number within the range [1,100] and determines whether the number is a prime number or not. Design all test cases for this program using BVC, Robust testing and worst-case testing methods:

1: Test cases using BVC

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	53	Prime number

Min value = 1
Min$^+$ value = 2
Max value = 100
Max$^-$ value = 99
Nominal value = 50–55

2: Test cases using Robust Testing method:

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	53	Prime number

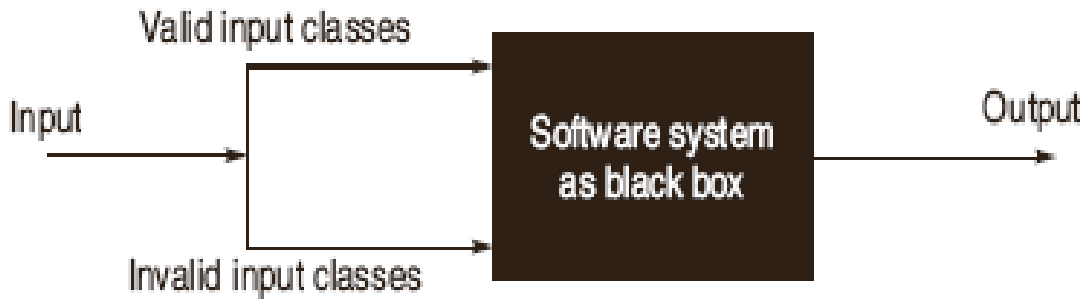
Min value = 1
Min$^-$ value = 0
Min$^+$ value = 2
Max value = 100
Max$^-$ value = 99
Max$^+$ value = 101
Nominal value = 50–55

Equivalance Class Testing:

Equivalence class partitioning is a specification-based or black-box technique. It can be applied at any level of testing and is often a good technique to use first. The idea behind this technique is to divide (i.e. to partition) a set of test conditions into groups or sets that can be considered the same (i.e. the system should handle them equivalently), hence 'equivalence class partitioning'.

In equivalence-partitioning technique we need to test only one condition from each partition. This is because we are assuming that all the conditions in one partition will be treated in the same way by the software. If one condition in a partition works, we assume all of the conditions in that partition will work, and so there is little point in testing any of these others. Similarly, if one of the conditions in a partition does not work, then we assume that none of the conditions in that partition will work so again there is little point in testing any more in that partition.

Identification of Equivalent Classes:



Different equivalent classes are formed by grouping inputs for which the behaviour pattern of the module is similar. The rationale for forming Equivalence classes like this is the assumption that if the specifications requiress exactly the same behaviour for each element in a class of values. Two types of classes can always be identified:

--Valid Equivalence class: The classes consider valid inputs to the programs.

--Invalid Equivalence class: The other class contains invalid inputs that will generate error conditions or unexpected behaviour of the program.

Identifying the Test Cases:

--Assign a unique identification number to each Equivalence class.

--Write a new test case covering as many of the uncovered valid Equivalence class 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 case.

Example: A program reads three numbers A, B and C with range [1,50] and prints largest number. Design all test cases for this program using equivalence class testing technique. Example test cases are:

I1 = {<A,B,C> : $1 \leq A \leq 50$}
I2 = {<A,B,C> : $1 \leq B \leq 50$}
I3 = {<A,B,C> : $1 \leq C \leq 50$}
I4 = {<A,B,C> : $A < 1$}
I5 = {<A,B,C> : $A > 50$}

I6 = {<A,B,C> : B < 1}
I7 = {<A,B,C> : B > 50}
I8 = {<A,B,C> : C < 1}

I9 = {<A,B,C> : C > 5

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 input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

State Table Based Testing:

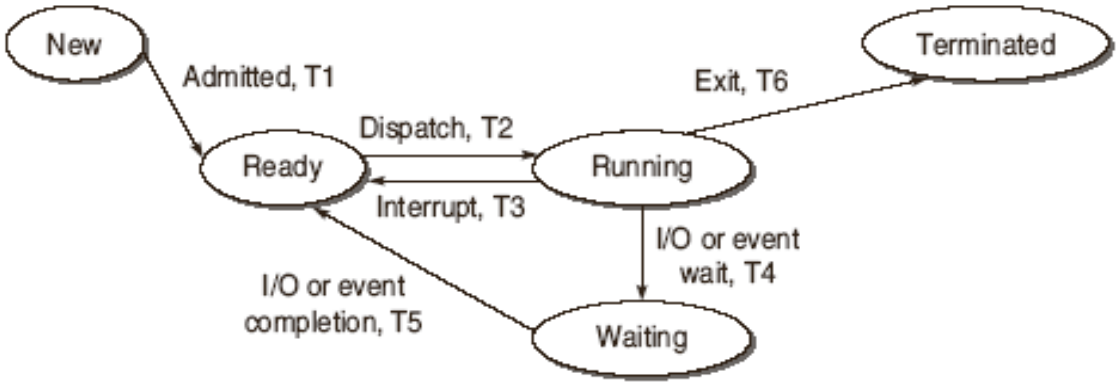
Tables are useful tools for representing and documenting many types of information relating to test case design. These are benificial for the applications which can be described using state transition diagrams and state tables.

Finite State Machine (FSM):

An FSM is a behavioural model whose outcome depends upon both previous and current inputs. FSM model can be prepared for software structure or software behaviour. And it can be used as a tool for functional testing.

State transition diagrams or State Graph:

A system or its components may have a number of states depending on its input and time.



State Table:

State graphs of larger systems may not be easy to understand. Therefore, state graphs are converted into tabular form for convinience sake, which are known as state tables. State table also specify, inputs, transitions and outputs. The following conventions are used for state table:

--Each row of the table corresponds to an input condition.

--Each column corresponds to an input condition.

--The box at the intersection of a row and a column specifies the next state (transition) and the output.

State\Input Event	Admit	Dispatch	Interrupt	I/O or Event Wait	I/O or Event Wait Over	Exit
New	**Ready/ T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready/ T1	**Running/ T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running/T2	Running/ T2	**Ready / T3**	**Waiting/ T4**	Running/ T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting / T4	Waiting/T4	Waiting / T4	**Ready / T5**	Waiting / T4

State Table Based Testing:

The procedure for converting state graphs and state tables into test cases is:

--Identify the States: The number of states in a state graph is the number of states we choose to recognize or model. In practice, the state is directly or indirectly recorded as a combination of vlaues of variables that appear in the database.

--Prpare state transition digram after understanding transitions between states: After having all the states, identify the inputs on each state and transitions between states and prepare the state graph. Every input state combination muast have a specified transition.

--Convert the state graph into the state table as discussed earlier.

--Analyse the state table for its completeness.

--Create the correspomding test cases from the state table.

Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

Decision Table Based Testing:

Boundary Value Analysis and Equivalance class parttioning methods do not consider combinations of input conditions. These methods consider the inputs separately. Decision Table is another useful mehod to represent the information in a tabular method. It has the speciality to consider complex combinations of input conditions and resulting actions.

Formation of Decision Table:

ENTRY

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

A decision table is formed with the following components:

Condition Stub: It is a list of input condiions for which the complex combination is made.

Action Stub: It is a list resulting actions which will be performed if a combination of input condition is satisfied.

Condition entry: It is a specific entry in the table correponding to input conditions mentioned in the condition stub.

Action Entry: It is the entry in the table for the resulting action to be performed when one rule is satisfied.

Test Case Design using Decision Table:

For Designing test cases from a decision table, following interpretations should be done:

--Interpret condition stubs as the inputs for the test case.

--Interpret action stubs as the expected output for the test case.

Example:
A program calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design the test cases using decision table testing.

ENTRY

		Rule 1	Rule 2	Rule3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Cause Effect Graphing Based Testing:

Cause and effect graph is a dynamic test case writing technique. Here causes are the input conditions and effects are the results of those input conditions. Cause-Effect Graphing is a technique which starts with set of requirements and determines the minimum possible test cases for maximum test coverage The goal is to reduce the total number of test cases still achieving the desired application quality.
Disadvantage: It takes time to model all your requirements into this cause-effect graph before writing test cases.

The following process is used to derive the tes cases:

Division of specification: The specifiacion is divided into workable pieces, as cause-effect graphing becomes complex when used on large specifiacitions.

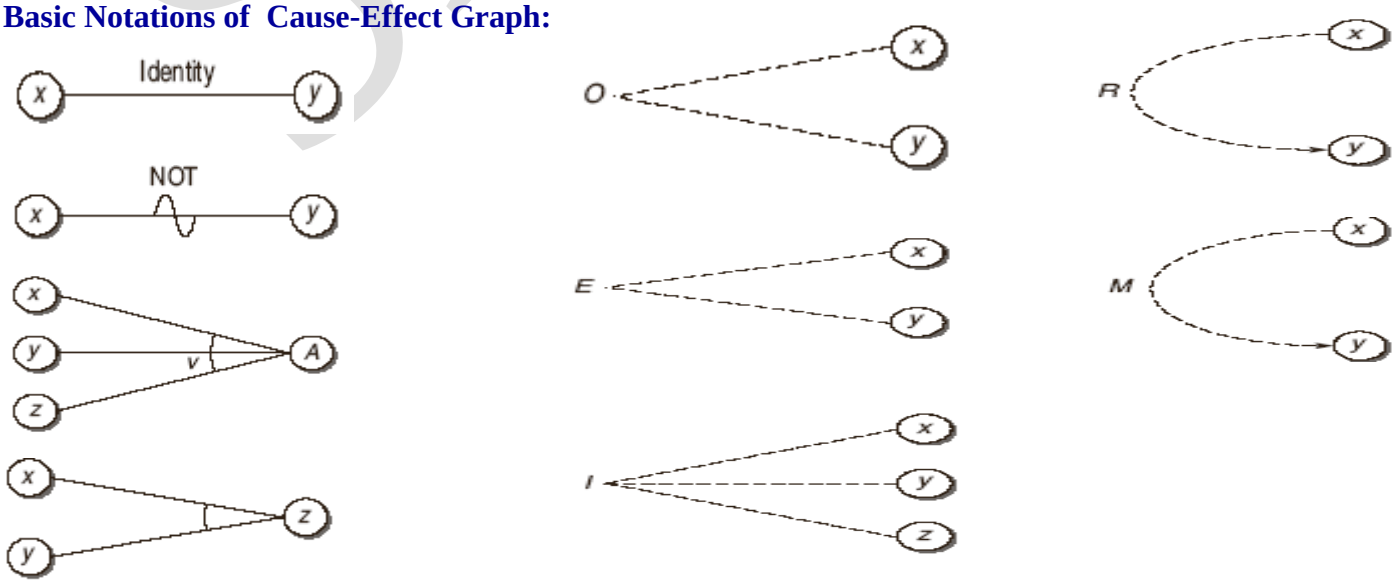
Identifiaction of causes and effects: A cause is a distinct input condition identified in the problem.

Transformation of specification into a cause-effect graph: Based on the analysis of the specification it is transformed into a boolean graph linking the causes and effetcs. This is the cause-effect graph.

Converison into decision table: The cause-effect graph obtained is converted into a limitedentry decision table by verifying state conditions in the graph.

Deriving test cases: The columns in the decision table are converted into tst cases.

Basic Notations of Cause-Effect Graph:



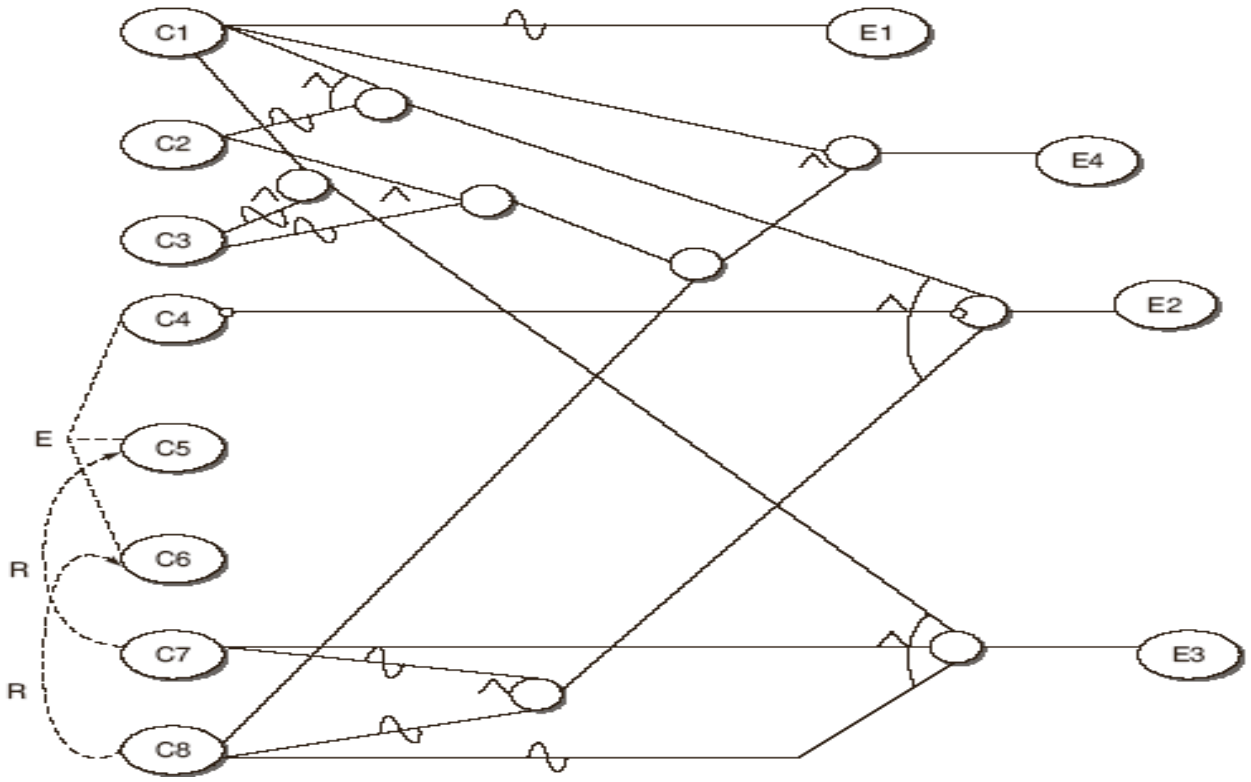
The operations are: AND, NOT, OR, AND, Exclusive (only one value can be 1), Inclusive (atleast x or y or z value must always be 1), Requires (if x to be 1, y must be 1), Mask (if x is 1 it forces y to be 0)

Example:
A program has been designed for the determination of nature of roots of a quadratic equation. Quadratic equation takes three input values from the range [0,100]. Design all test cases using Cause-Effect graphing technique.

The Causes and Effects are:

C1: a ≠ 0
C2: b = 0
C3: c = 0
C4: D > 0 where D is b2 – 4 * a * c
C5: D < 0
C6: D = 0
C7: a = b = c
C8: a = c = b/2
E1: Not a quadratic equation
E2: Real Roots
E3: Imaginary Roots
E4: Equal Roots

The Cause Effect graph is:



	R1	R2	R3	R4	R5	R6	R7
C1: a ≠ 0	T	T	T	T	T	T	F
C2: b = 0	F	I	I	T	F	F	I
C3: c = 0	I	F	I	T	F	F	I
C4: D > 0	T	F	F	F	F	F	I
C5: D < 0	F	T	F	F	T	F	I
C6: D = 0	F	F	T	T	F	T	I
C7: a = b = c	F	I	F	F	T	F	I
C8: a = c = b/2	F	F	I	F	F	T	I
A1: Not a quadratic equation							X
A2: Real roots	X						
A3: Imaginary roots		X			X		
A4: Equal roots			X	X		X	

Test Case ID	a	b	c	Expected Output
1	1	50	50	Real roots
2	100	50	50	Imaginary roots
3	1	6	9	Equal
4	100	0	0	Equal
5	99	99	99	Imaginary
6	50	100	50	Equal
7	0	50	30	Not a quadratic equation

Error Guessing:

Error Guessing is the preferred method when all the other methods fail. It is a very practical case wherin the tester uses his intution and makes a guess about where the bug can be. The tester does not have to use any particular testing technique. The basic idea is to make a list of possible errors in error prone situations and then develop the test cases. Thus there is no general procedure for this technique as it is largely an intutive and ad-hoc process.