

## Day 1: Introduction to Software Testing & Techniques

### Topics Covered:

- Introduction to S/W Testing
- Overview of the Course
- Software Testing Techniques
- Software Testing –
  - Quality Attributes
  - Types
  - Levels

**Testing** is the process of evaluating a system or its Components with the intent to find whether it satisfies the specific requirements or not. It is done in order to identify Gaps, Errors or Missing Requirements to the actual requirements. Its about executing program with intention to break the system | find the errors in the system.

**Software Quality** is something that the product meets established requirements; the capability of a software product to satisfy stated and implied needs when used under specified conditions

*Psychology of Tester should have an attitude to break the system which is Constructively Destructive*

### Product under Test

Basically, Software testing is not alone, its not tested in isolation. It interacts with other software, hardware and systems. It must be compatible with other software/systems like DBs, servers etc., This not only be compatible with others, but also others compatible with our product. The system should perform its intended functions correctly every time it is used. And it should be always available, and Downtime should be Minimized.

*Success of a product can be increased by increasing the probability of success, continuous improvement and measure which you cannot control*

## Testing:

A **test case** is a set of conditions or variables used to determine if a software application is functioning correctly. It includes specific inputs, execution conditions, and expected results to verify a particular feature or functionality of the software.

*While there are an infinite number of possible test cases for a function, we need to select only those test cases that provide confidence in the software's functionality.*

**Error** is a human action that produces an incorrect result. It is a mistake made by a developer or a user.

Ex: Typo error in the code or misunderstanding the requirements

**Fault** is a manifestation of an error in the software. It is an incorrect step, process or data definition in a program. Faults are the actual issues in the code that can cause the software to behave unexpectedly.

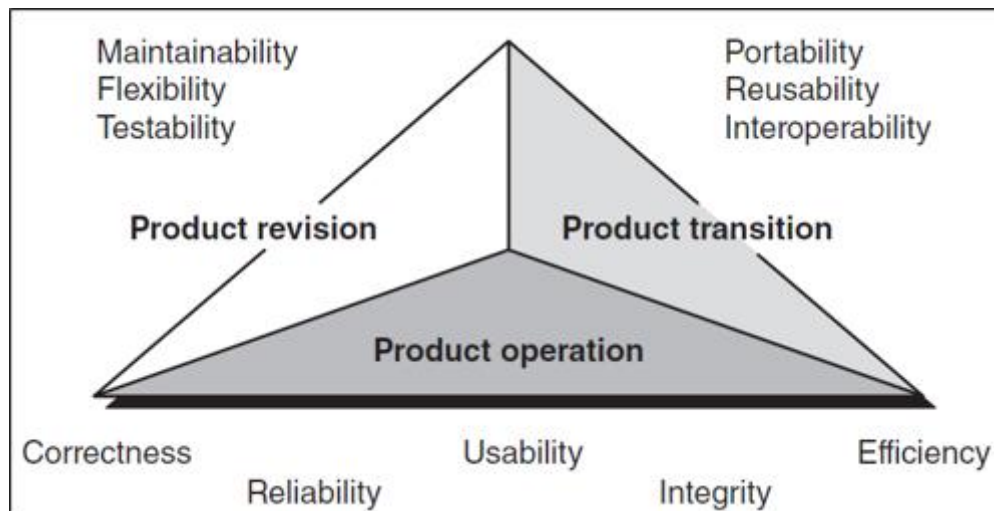
- Faults of **omission**: Occurs when something is missed out
- Faults of **commission**: Occurs when some representation is incorrect
- E.g – Message Box “The colour of your **Troser** is not \_\_\_\_\_”

**Failure** occurs when a Fault is executed

## Testing Techniques:

- Based on Engineers experience and intuition
  - Exploratory
  - Ad-hoc
- Specification Based Techniques
  - Equivalence Partitioning
  - Boundary Value Analysis
  - Decision Tables
  - Orthogonal Array Testing
  - Cause-Effect Graphing
- Code Based Techniques
  - Control Flow
  - Data Flow

## Quality Attributes:



### 1. Product Operation

These attributes focus on the software's behavior during its operation.

- ✚ **Correctness:** The extent to which the software performs its intended functions accurately.
- ✚ **Reliability:** The ability of the software to perform its required functions under stated conditions for a specified period.
- ✚ **Efficiency:** The software's ability to use system resources effectively and efficiently.
- ✚ **Integrity:** The protection of the software from unauthorized access and ensuring data security.
- ✚ **Usability:** The ease with which users can learn and use the software.




### 2. Product Revision

These attributes relate to the software's ability to undergo changes and improvements.

- ✚ **Maintainability:** The ease with which the software can be modified to correct faults, improve performance, or adapt to a changed environment.
- ✚ **Flexibility:** The ability of the software to adapt to changes in its environment or requirements.
- ✚ **Testability:** The ease with which the software can be tested to ensure it meets its requirements.

## 3. Product Transition

These attributes are concerned with the software's adaptability to new environments.

-  **Portability:** The ease with which the software can be transferred from one environment to another.
-  **Reusability:** The extent to which parts of the software can be used in other applications.
-  **Interoperability:** The ability of the software to interact with other systems or software.

## ISO 25010: Product Quality

This category focuses on the intrinsic properties of the software product itself. It includes eight characteristics:

1. Functional Suitability:
  - Functional Completeness: The degree to which the set of functions covers all the specified tasks and user objectives.
  - Functional Correctness: The degree to which the software provides correct results.
  - Functional Appropriateness: The degree to which the functions facilitate the accomplishment of specified tasks.
2. Performance Efficiency:
  - Time Behaviour: The response time and throughput rates of the software.
  - Resource Utilization: The amount and types of resources used by the software.
  - Capacity: The maximum limits of the software's parameters.
3. Compatibility:
  - Co-existence: The software's ability to perform its functions efficiently while sharing a common environment with other products.
  - Interoperability: The ability of the software to interact with other systems or components.
4. Usability:
  - Appropriateness Recognizability: How easily users can recognize whether the software is appropriate for their needs.
  - Learnability: How easily users can learn to use the software.
  - Operability: How easily users can operate and control the software.
  - User Error Protection: The software's ability to protect users from making errors.
  - User Interface Aesthetics: The aesthetic appeal of the user interface.

- Accessibility: The software's accessibility to users with disabilities.

## 5. Reliability:

- Maturity: The frequency of software failures.
- Availability: The degree to which the software is operational and accessible when required.
- Fault Tolerance: The software's ability to maintain performance in the event of faults.
- Recoverability: The ability to recover data and restore the software after a failure.

## 6. Security:

- Confidentiality: The protection of data from unauthorized access.
- Integrity: The protection of data from unauthorized modification.
- Non-repudiation: The ability to prove the occurrence of actions or events.
- Accountability: The ability to trace actions to the responsible entity.
- Authenticity: The ability to verify the identity of a user or system.




## 7. Maintainability:

- Modularity: The degree to which the software is composed of discrete components.
- Reusability: The degree to which components can be reused in other applications.
- Analyzability: The ease with which the software can be analysed for defects.
- Modifiability: The ease with which the software can be modified.
- Testability: The ease with which the software can be tested.

## 8. Portability:

- Adaptability: The ease with which the software can be adapted to different environments.
- Installability: The ease with which the software can be installed.
- Replaceability: The ease with which the software can replace other software in the same environment.

## Day 2: Mathematics and Formal Methods

-  Permutation and Combination
-  Propositional Logic
-  Discrete Math's
  - Sets theory
  - Graph theory

### Permutation and Combination

Permutation relates to the act of arranging all the members of a set into some sequence or order.

#### Repetition Allowed

Let's use the example with letters A, B, and C (so (  $n = 3$  )) and creating 2-letter combinations (so (  $r = 2$  )):

$$[ 3^2 = 9 ]$$

This matches the 9 combinations we listed earlier: AA, AB, AC, BA, BB, BC, CA, CB, and CC.

#### Without Repetition

Let's use an example with the same letters A, B, and C (so (  $n = 3$  )) and creating 2-letter combinations (so (  $r = 2$  )):

$$\frac{n!}{(n-r)!} = \frac{3!}{(3-2)!} = \frac{3*2*1}{1} = 6$$

So, there are 6 possible permutations without repetition: AB, AC, BA, BC, CA, CB

Combination is a way of selecting items from a collection, such that (unlike permutations) the order of selection does not matter.

#### Without Repetition

Let's use an example with the same letters A, B, and C (so (  $n = 3$  )) and creating 2-letter combinations (so (  $r = 2$  )):

$$\frac{n!}{r!(n-r)!} = \frac{3!}{2!(3-2)!} = \frac{6}{2(1)} = 3$$

If we allowed **repetition**, it would be a different situation where each item can be selected more than once. The number of combinations would be different, and the approach to calculate it would be:

$$\binom{n+r-1}{r} = \frac{(n+r-1)!}{r!(n-1)!}$$

Using the same values ( $n = 3$  and  $r = 2$ ):

$$\frac{(3+2-1)!}{2!(3-1)!} = \frac{4!}{2! \cdot 2!} = \frac{24}{4} = 6$$

So, there would be 6 combinations with repetition allowed: AA, AB, AC, BB, BC, CC.

Order Matters	Repetition Allowed	Formula
Yes (Permutation)	Yes	$P(n, r) = n^r$
Yes (Permutation)	No	$P(n, r) = \frac{n!}{(n-r)!}$
No (Combination)	No	$C(n, r) = \frac{n!}{r!(n-r)!}$
No (Combination)	Yes	$C(n+r-1, r) = \frac{(n+r-1)!}{r!(n-1)!}$

## Propositional Logic

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \oplus B$	$A \rightarrow B$
T	T	F	F	T	T	F	T
T	F	F	T	F	T	T	F
F	T	T	F	F	T	T	T
F	F	T	T	F	F	F	T

## Discrete Math:

Collection of things which have a common property

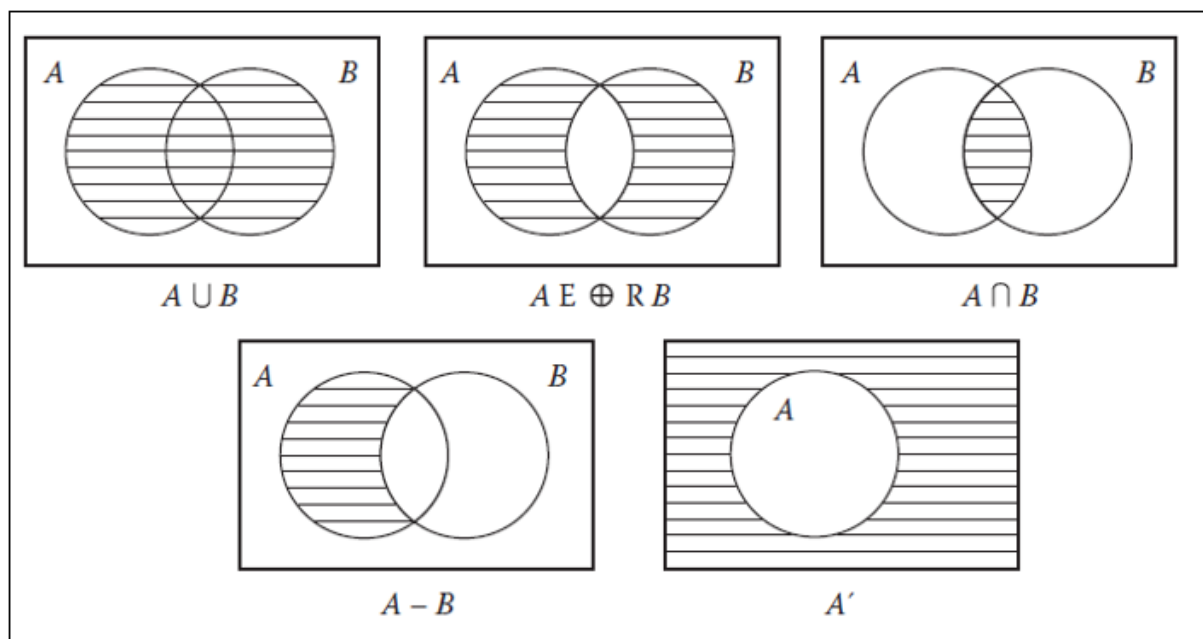
Their *union* is the set  $A \cup B = \{x: x \in A \vee x \in B\}$ .

Their *intersection* is the set  $A \cap B = \{x: x \in A \wedge x \in B\}$ .

The *complement* of  $A$  is the set  $A' = \{x: x \notin A\}$ .

The *relative complement* of  $B$  with respect to  $A$  is the set  $A - B = \{x: x \in A \wedge x \notin B\}$ .

The *symmetric difference* of  $A$  and  $B$  is the set  $A \oplus B = \{x: x \in A \oplus x \in B\}$ .



## Ordered Pair

An ordered pair is a pair of elements where the order matters.

- **Example:** (A, B) and (B, A) are different ordered pairs.

## Unordered Pair

An unordered pair is a pair of elements where the order does not matter.

- **Example:** {A, B} and {B, A} are the same unordered pair.

## Subset

A subset is a set where all elements are contained within another set.

- **Example:** If  $SetA = \{1, 2, 3\}$  and  $SetB = \{1, 2\}$ , then Set B is a subset of Set A, written as  $B \subseteq A$ .



## Proper Subset

A proper subset is a subset that is not identical to the original set; it contains some but not all elements of the original set.

- **Example:** If  $SetA = \{1, 2, 3\}$  and  $SetB = \{1, 2\}$ , then Set B is a proper subset of Set A, written as  $B \subset A$ .

## Equal Set

Two sets are equal if they contain exactly the same elements.

- **Example:** If  $SetA = \{1, 2, 3\}$  and  $SetB = \{3, 1, 2\}$ , then  $A = B$  because they contain the same elements.

## Set Partition

A set partition divides a set into non-overlapping subsets such that every element is included in exactly one subset.

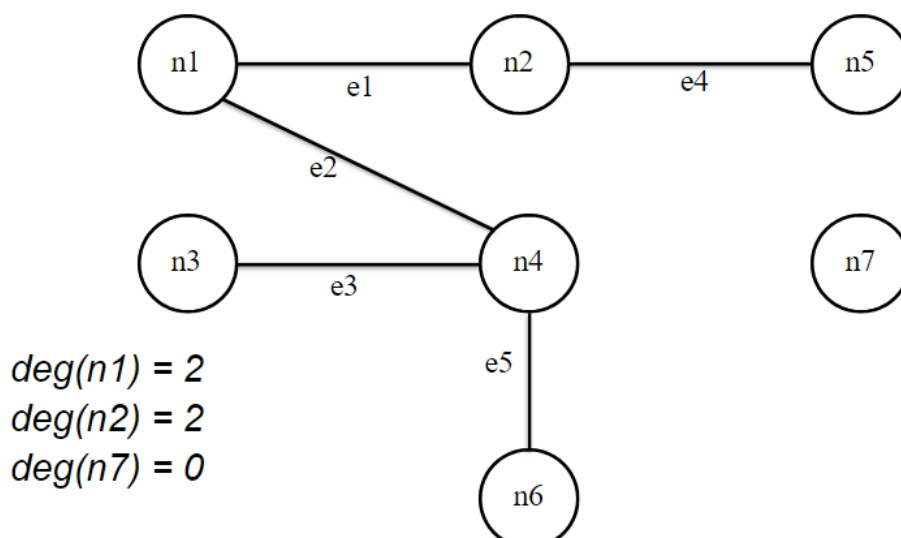
- **Example:** If  $SetA = \{1, 2, 3\}$ , a partition of Set A could be  $\{\{1\}, \{2, 3\}\}$ .

## Graph Theory:



## Degree of a Node

- The degree of a node in a graph is the number of edges that have that node as an endpoint  $\deg(n)$

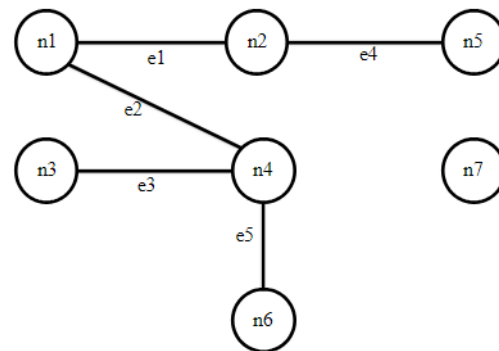


## Incidence Matrix



- The incidence matrix is a graph  $G=(V,E)$  with  $m$  nodes and  $n$  edges is a  $m \times n$  matrix, where the element in row  $i$ , column  $j$  is a 1 if and only if node  $i$  is an endpoint of edge  $j$ ; otherwise the element is 0

	$e1$	$e2$	$e3$	$e4$	$e5$
$n1$	1	1	0	0	0
$n2$	1	0	0	1	0
$n3$	0	0	1	0	0
$n4$	0	1	1	0	1
$n5$	0	0	0	1	0
$n6$	0	0	0	0	1
$n7$	0	0	0	0	0

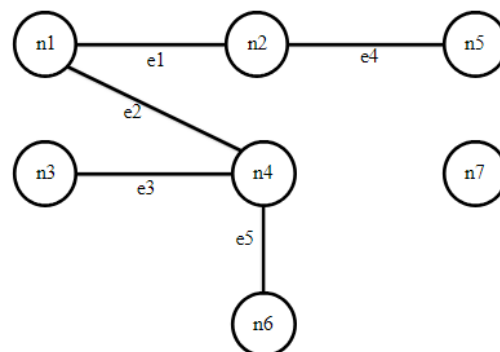


## Adjacency Matrix



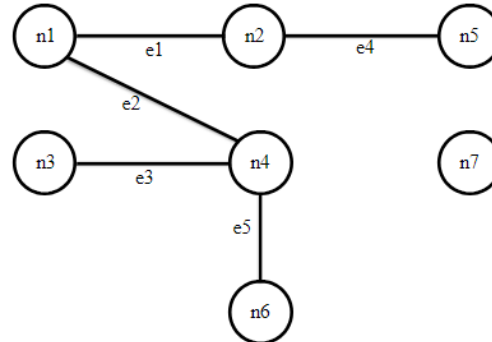
- Deals with connections
- The adjacency matrix of a Graph  $G=(V,E)$  with  $m$  nodes is an  $m \times m$  matrix, where the element in row  $i$ , column  $j$  is 1 if and only if an edge exists between node  $i$  and node  $j$ ; otherwise, the element is 0
- Useful for later graph theory concepts example: paths

	$n1$	$n2$	$n3$	$n4$	$n5$	$n6$	$n7$
$n1$	0	1	0	1	0	0	0
$n2$	1	0	0	0	1	0	0
$n3$	0	0	0	1	0	0	0
$n4$	1	0	1	0	0	1	0
$n5$	0	1	0	0	0	0	0
$n6$	0	0	0	1	0	0	0
$n7$	0	0	0	0	0	0	0



# Paths

- A path is a sequence of edges such that for any adjacent pair of edges  $e_i, e_j$  in the sequence, the edges share a common (node) endpoint



Path	Node Sequence	Edge Sequence
Between n1 and n5	n1, n2, n5	e1, e4
Between n6 and n5	n6, n4, n1, n2, n5	e5, e2, e1, e4
Between n3 and n2	n3, n4, n1, n2	e3, e2, e1

## Specification Based Testing, Boundary Value Analysis & Equivalence Class Testing

### What is Specification Based Testing?

Specification-based testing is a crucial black-box testing technique where the testing process is driven by the specifications or requirements of a system. It is also known as requirements-based or functional testing. This approach ensures that the software system behaves according to the specified requirements, without considering the internal workings of the system.

- Definition and Objective:** Specification-based testing aims to create test cases that cover all possible scenarios outlined in the specifications. The primary objective is to ensure that the system behaves as expected under various conditions as defined by the requirements.
- Techniques:** Several techniques are used in specification-based testing:
  - Equivalence Partitioning:** This technique involves dividing the input data into partitions where the system is expected to behave similarly. Each partition should produce similar results. For example, if an input field

accepts values from 1 to 100, these values can be partitioned into valid (1-100) and invalid (e.g., -1, 0, 101) sets.

- **Boundary Value Analysis:** Boundary values are the edges of input ranges where errors are most likely to occur. Testing just inside, on, and just outside the boundaries can uncover issues. Continuing with the previous example, the boundaries would be 1 and 100.
- **Decision Table Testing:** This technique uses decision tables to represent different combinations of inputs and their corresponding outcomes. It's beneficial for systems where multiple conditions can affect the output.
- **State Transition Testing:** This technique tests the system's behavior in different states. It is particularly useful for systems where outputs depend on the sequence of previous inputs, such as a state machine.

### 3. Advantages:

- **Coverage:** By focusing on specifications, this method ensures comprehensive coverage of all functional aspects of the system.
- **Defect Detection:** It helps identify discrepancies between the system's behaviour and its specifications early in the development cycle.
- **Simplicity:** Testers do not need to know the internal code structure, making it easier for non-developers to create test cases.

4. **Examples:** Consider a login system with requirements specifying that the system should allow users to log in with a valid username and password and reject invalid credentials. Test cases would be created to cover valid login attempts, invalid usernames, incorrect passwords, and boundary cases like empty fields.

## What is Boundary Value Analysis?

**Boundary Value Analysis (BVA)** is a widely-used black-box testing technique that involves testing at the boundaries of input ranges. It is based on the principle that errors are more likely to occur at the edges of input domains rather than in the middle. Here's a detailed breakdown of BVA, including its definition, examples, generalization, limitations, worst-case analysis, and its role in black-box testing, along with special value testing:

### Definition

Boundary Value Analysis is a testing technique where test cases are designed to include values at the boundaries. The technique ensures that all edge cases are tested and helps identify defects that occur at boundary values.

### Example

Consider a system that accepts input values ranging from 1 to 100. Using BVA, the test cases would include:

- Minimum boundary value: 1
- Just below the minimum boundary: 0
- Just above the minimum boundary: 2
- Maximum boundary value: 100
- Just below the maximum boundary: 99
- Just above the maximum boundary: 101

### Generalizing BVA

BVA can be generalized for various types of input ranges:

- **Numeric Range:** Testing minimum and maximum values and their immediate neighbors.
- **Date Range:** Testing boundary dates and days around those boundaries.
- **String Length:** Testing boundary lengths of input strings and their immediate neighbors.

### Limitations

- **Complex Boundaries:** BVA may not be effective for complex input domains with multiple boundaries.

- **Interdependent Parameters:** When input parameters are interdependent, BVA may not cover all possible interactions.
- **Non-Continuous Domains:** BVA is less effective for inputs that are non-continuous or have distinct categories.

## **Worst-Case Analysis**

Worst-case analysis involves testing the extreme boundaries and combinations of boundary values to ensure that the system can handle the most challenging scenarios. This often requires combining BVA with other techniques, like decision table testing, to cover all worst-case scenarios.

## **Black-Box Testing**

BVA is a black-box testing technique because it focuses on the input and output of the system without considering the internal code structure. It ensures that the system behaves correctly at boundary conditions as specified in the requirements.

## **Special Value Testing**

Special value testing involves testing values that are known to cause errors or have special significance, such as null values, empty strings, or specific characters. These values are chosen based on knowledge of common issues in similar systems.

## What is Equivalence Class?

### Equivalence Classes (EC)

Equivalence classes are used to partition input data into groups where each group represents a set of values that are processed similarly by the system. This helps in reducing the number of test cases while ensuring comprehensive coverage.

### Types of Equivalence Class Testing

#### 1. Weak Normal Equivalence Class Testing:

- **Definition:** This involves testing with all valid inputs.
- **Example:** If an input range is from 1 to 100, then selecting any valid value within this range (e.g., 25, 50, 75) for testing.

#### 2. Strong Normal Equivalence Class Testing:

- **Definition:** This involves testing with all valid inputs, but ensuring that one value from each sub-interval equivalence class is included.
- **Example:** For the input range 1 to 100, if we divide it into sub-intervals (e.g., 1-25, 26-50, 51-75, 76-100), we would select one value from each sub-interval (e.g., 10, 35, 60, 85).

#### 3. Weak Robust Equivalence Class Testing:

- **Definition:** This involves testing with one invalid input.
- **Example:** For the input range 1 to 100, selecting an invalid input like -5 or 150 to see how the system handles it.

#### 4. Strong Robust Equivalence Class Testing:

- **Definition:** This involves testing with more than one invalid input.
- **Example:** For the input range 1 to 100, selecting multiple invalid inputs like -5, 0, and 150 to thoroughly test the system's handling of invalid inputs.

### Formation of Equivalence Classes

A group forms an equivalence class if:

- They all test the same functionality or behaviour.
- If one test case catches a defect, the others in the group likely will too.
- If one test case doesn't catch a defect, the others in the group likely won't either.

## Criteria for Considering Inputs as Equivalent

We consider inputs as equivalent if:

- They involve the same input variable.
- They result in similar operations being performed by the program.
- They affect the same output variable.
- None force the program to do error handling, or all of them do.

## Equivalence Classes for Invalid Inputs

When dealing with invalid inputs, we consider:

- **Range in Numbers:** Identifying values outside the valid range (e.g., for a range of 1-100, -5 and 150 are invalid).
- **Membership in a Group:** Checking if the input belongs to an invalid category (e.g., entering a letter where a number is expected).
- **Responses to Lists and Menus:** Testing invalid selections in a list or menu.
- **Variables that Must be Equal:** Ensuring that inputs that should be equal are tested for inequality (e.g., password and confirm password fields).
- **Time-Determined Equivalence Classes:** Inputs based on time conditions (e.g., dates in the past or future).
- **Equivalent Output Events:** Identifying outputs that are considered equivalent.
- **Variable Groups that Must Calculate to a Certain Value or Range:** Ensuring that combined input values meet expected conditions.
- **Equivalent Operating Environments:** Testing in different but equivalent operating conditions (e.g., different browsers or operating systems).



## Domain Testing:

Domain testing is a software testing technique where a minimal number of inputs are used to verify the appropriate outputs of a system, ensuring that invalid input values are not accepted. The objective is to ensure that the system provides the required outputs while blocking invalid inputs.

## Domain Knowledge Importance

**Domain knowledge** is crucial for a tester because it allows them to understand the specific context and requirements of the system they are testing. This knowledge helps testers create more effective and relevant test cases. Here are a few examples:

- **Online Banking:** Testers need to understand activities like login, bill payment, and transfers. This includes knowledge of security measures, transaction processes, and user interface standards.
- **Retail Domains:** Knowledge of workflow at different levels, such as inventory management, point of sale, and customer relationship management. Understanding these processes helps testers ensure that all aspects of the retail system function correctly.

## Category Testing:

Category partitioning is a systematic technique used in software testing to create a comprehensive and efficient set of test cases by dividing the input domain into distinct categories or partitions. The objective is to achieve maximum test coverage with a minimal number of test cases, thereby ensuring that the software behaves correctly across different scenarios.

### Key Concepts of Category Partitioning

#### 1. Categories and Partitions:

- **Categories:** These represent different aspects or characteristics of the input data. For example, in an online form, categories could include input fields like "name," "age," and "email."
- **Partitions:** These are subsets of each category that group similar values together. Each partition represents a range or type of input values. For example, the "age" category could have partitions like "under 18," "18-65," and "over 65."

2. **Selection of Test Cases:** By strategically choosing representative values from each partition, testers can ensure that all relevant scenarios are covered. This helps in identifying defects that may occur at the boundaries or within specific partitions

### Steps in Category Partitioning

#### 1. Analyze Specification:

- **Description:** Carefully review the software requirements and specifications. This step ensures that you understand the functionality and behavior of the system.
- **Purpose:** To identify the various inputs, outputs, and system interactions that need to be tested.

#### 2. Identify Categories:

- **Description:** Determine the different categories or aspects of the input data that affect the system's behavior.
- **Purpose:** To group inputs based on their relevance and impact on the system. For example, in a form, categories could include fields like "username," "password," and "email."

### 3. Partition Categories:

- **Description:** Divide each identified category into partitions, which are subsets of the category. These partitions represent groups of similar values or conditions.
- **Purpose:** To create manageable and meaningful groups for testing. For example, the "age" category could have partitions like "under 18," "18-65," and "over 65."

### 4. Identify Constraints:

- **Description:** Determine any constraints or dependencies between the categories. Constraints could include rules that must be followed, such as specific input conditions that must or must not coexist.
- **Purpose:** To ensure that all relevant scenarios are covered, including those that involve interdependent inputs.

### 5. (Re) Write Test Specification:

- **Description:** Document the test cases based on the identified categories, partitions, and constraints. This step involves writing or updating the test specifications to reflect the partitions.
- **Purpose:** To provide a clear and detailed plan for testing that includes all necessary test cases.

### 6. Process Specification:

- **Description:** Review and refine the test specification to ensure completeness and accuracy. This step may involve peer reviews or consultations with stakeholders.
- **Purpose:** To validate the test specification and ensure that it aligns with the requirements and covers all critical scenarios.

### 7. Evaluate Generator Output:

- **Description:** If using automated test case generation tools, evaluate the generated test cases to ensure they meet the criteria and cover all necessary partitions.
- **Purpose:** To verify that the automated tool has produced accurate and relevant test cases.

### 8. Generate Test Scripts:

- **Description:** Create executable test scripts based on the test specification. This involves writing the actual test scripts that will be used to perform the tests.
- **Purpose:** To prepare the tests for execution, ensuring that they are ready to be run and can effectively validate the system's behavior.

**Combinatorial testing** is a software testing technique that aims to efficiently identify and test all possible combinations of input parameters. This approach is particularly important for complex systems where the number of potential input combinations can be vast. The need for combinatorial testing arises from the desire to ensure thorough test coverage while managing the practical constraints of time and resources.

## Need of Combinatorial



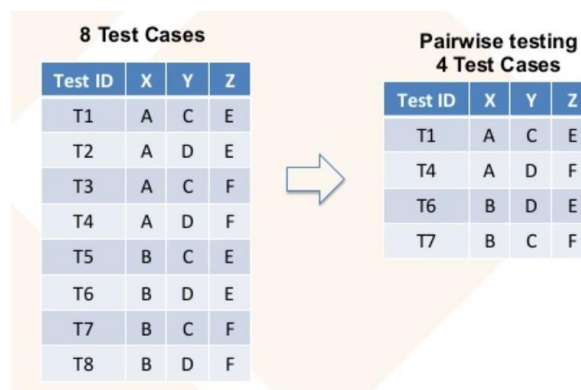
OS - XP, OS X, RHL (3)  
 Browser - IE, Firefox (2)  
 Protocol - IPV4, IPV6 (2)  
 CPU - Intel, AMD (2)  
 DBMS - Sybase, Oracle, MySQL (3)

Total Possible Test case  $3 \times 2 \times 2 \times 2 \times 3 = 72$

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPV4	Intel	MySQL
2	XP	Firefox	IPV6	AMD	Sybase
3	XP	IE	IPV6	Intel	Oracle
4	OS X	Firefox	IPV4	AMD	MySQL
5	OS X	IE	IPV4	Intel	Sybase
6	OS X	Firefox	IPV4	Intel	Oracle
7	RHL	IE	IPV6	AMD	MySQL
8	RHL	Firefox	IPV4	Intel	Sybase
9	RHL	Firefox	IPV4	AMD	Oracle
10	OS X	Firefox	IPV6	AMD	Oracle

t	#Configs	%age
2	10	14
3	18	25
4	36	50
5	72	100

**Pairwise Testing**, also known as all-pairs testing, is a combinatorial testing technique that involves testing all possible discrete combinations of pairs of input parameters. The goal is to efficiently identify defects that might occur due to the interaction of different parameter values.



## Advantages:

1. **Efficiency:** Reduces the number of test cases, saving time and resources.
2. **Defect Detection:** Effective at identifying defects caused by interactions between pairs of input parameters.

## Disadvantages:

1. **Limited Scope:** May miss defects caused by interactions involving more than two parameters.
2. **Assumption-Based:** Relies on the assumption that most defects are caused by pairs of inputs.

## Decision Table Testing:

A decision table, also known as a cause-effect table, is a tabular representation of the logical relationships between input conditions (causes) and their corresponding actions (effects). It is a useful technique in software testing for identifying and organizing test cases systematically.

### Component Specification Example

- **1st Character:** Must be A or B.
- **2nd Character:** Must be a digit.
- **Valid Conditions:**
  - If the 1st character is A or B and the 2nd character is a digit, the file is updated.
- **Invalid Conditions:**
  - If the 1st character is incorrect, message X12 is displayed.
  - If the 2nd character is not a digit, message X13 is displayed.

### Steps to Create a Decision Table

#### Step 1: Identify Input Conditions (Causes) and Actions (Effects)

- **Causes:**
  1. 1st Character (C1): A, B, Incorrect
  2. 2nd Character (C2): Digit, Non-digit

- **Effects:**

1. Update File (E1)
2. Display Message X12 (E2)
3. Display Message X13 (E3)

## Step 2: Develop a Cause-Effect Graph

Since we're using a decision table, we will skip the graphical representation and move directly to the table.

## Step 3: Transform the Cause-Effect Graph into a Decision Table

Input Conditions	C1: A	C1: B	C1: Incorrect	C2: Digit	C2: Non-digit
Actions					
E1: Update File	Yes	Yes	No	Yes	No
E2: Display X12	No	No	Yes	No	No
E3: Display X13	No	No	No	No	Yes

## Step 4: Convert Decision Table Rules to Test Cases

Each column of the decision table represents a test case.

Test Case	C1: 1st Character	C2: 2nd Character	Expected Action
TC1	A	Digit	Update File
TC2	B	Digit	Update File
TC3	Incorrect	Digit	Display Message X12
TC4	A	Non-digit	Display Message X13
TC5	B	Non-digit	Display Message X13
TC6	Incorrect	Non-digit	Display Message X12

## Software Testing – Week 5 and Week 6

### Code-Based Testing

#### Definition:

- Code-based testing, also known as white-box testing, involves designing tests based on the internal structure and logic of the source code. The input to the test design is the source code or the program structure itself.

#### Salient Features:

##### 1. More Rigorous than Specification Testing:

- Code-based testing is generally more thorough compared to specification-based testing because it examines the internal workings of the code. This allows for the identification of hidden errors that might not be apparent through specification testing alone.

##### 2. Lower Level than Specification Testing:

- This type of testing operates at a lower level, focusing on the individual components and their interactions within the code. It does not rely on the high-level requirements or specifications but rather on the actual implementation.

##### 3. Validation with Respect to Specification May Not Happen:

- Since the input for code-based testing is the code itself, it may not always validate whether the code meets the original specifications or requirements. The primary focus is on ensuring that the code functions correctly according to its design.

#### Key Techniques in Code-Based Testing:

- **Statement Coverage:** Ensures that every statement in the code is executed at least once.
- **Branch Coverage:** Ensures that every possible branch (decision) in the code is executed.
- **Path Coverage:** Ensures that all possible paths through the code are executed.
- **Condition Coverage:** Ensures that each condition in a decision takes on all possible outcomes at least once.

## Advantages:

- **Thoroughness:** Can uncover hidden errors and vulnerabilities within the code.
- **Optimization:** Helps in optimizing the code by identifying redundant or inefficient code segments.
- **Security:** Enhances security by identifying potential security flaws within the code.

## Disadvantages:

- **Complexity:** Can be complex and time-consuming, especially for large codebases.
- **Requires Detailed Knowledge:** Testers need a deep understanding of the code and its structure.
- **May Miss Specification Issues:** Since it focuses on the code, it might miss issues related to the original specifications or requirements.

## Code Based Testing

- Techniques
  - Statement Testing
  - Branch Testing
  - Multiple Condition Testing
  - Loop Testing
  - Path Testing
  - Modified Path Testing (McCabe Path)
  - Dataflow Testing
  - Transaction Flow Testing



## Statement Testing

- Basic Concept

Every Statement in the program (code) should be covered at least once during testing

- Types of Statement
  - An assignment Statement
  - An input statement
  - An output statement
  - A function/procedure/subroutine call
  - A return statement
  - A predicate of condition statements
  - IF-THEN-ELSE
  - WHILE-DO/DO-WHILE
  - SWITCH
  - A variable declaration is not a statement

## Example of Statement Testing

Consider a simple function in Python that calculates the sum of two numbers:

```
def add_numbers(a, b):
```

```
    result = a + b
```

```
    return result
```

## Test Cases for Statement Testing:

### 1. Test Case 1:

- **Input:** a = 2, b = 3
- **Expected Output:** 5
- **Executed Statements:**
  - result = a + b (with a = 2 and b = 3)
  - return result (with result = 5)

## 2. Test Case 2:

- **Input:** a = -1, b = 4
- **Expected Output:** 3
- **Executed Statements:**
  - result = a + b (with a = -1 and b = 4)
  - return result (with result = 3)

In this example, both test cases ensure that every statement in the add\_numbers function is executed at least once. This is a simple illustration, but statement testing can be applied to more complex functions and codebases to ensure thorough testing.

## Branch Testing

- Basic Concept

Every branch in the program (code) should be executed at least once during testing.

- What does this coverage constitute?
- IF-THEN-ELSE
- WHILE-DO
- SWITCH

## Example of Branch Testing

Consider a simple function in Python that determines if a number is positive, negative, or zero:

```
def check_number(num):
```

```
    if num > 0:
```

```
        return "Positive"
```

```
    elif num < 0:
```

```
        return "Negative"
```

```
    else:
```

```
        return "Zero"
```

## Test Cases for Branch Testing:

### 1. Test Case 1:

- **Input:** num = 5
- **Expected Output:** "Positive"
- **Executed Branches:**
  - if num > 0 (True)
  - return "Positive"

### 2. Test Case 2:

- **Input:** num = -3
- **Expected Output:** "Negative"
- **Executed Branches:**
  - if num > 0 (False)
  - elif num < 0 (True)
  - return "Negative"

### 3. Test Case 3:

- **Input:** num = 0
- **Expected Output:** "Zero"
- **Executed Branches:**
  - if num > 0 (False)
  - elif num < 0 (False)
  - else (True)
  - return "Zero"

In this example, each test case ensures that every possible branch (decision) in the check\_number function is executed at least once. This helps to verify that all logical paths in the code are functioning correctly.

## Salient Features

- More demanding
- When branch testing is satisfied the statement testing is also satisfied

## Multiple Condition Testing

1. This is testing of condition with complex predicates (OR, NOT and AND)
  2. IF C1 THEN, ELSE \ Branch testing ~ multiple condition
  3. IF (C1 AND C2 AND C3) THEN, ELSE \ Multiple condition
- In case of first condition there is a single condition so the values can be true or false
  - In case of third condition, it is a complex predicate made up C1 AND C2 AND C3.
  - To test this “Test all combinations of simple predicates”

## Example

input (x, y)

if (x>0) and (y<1) then z=1

P1 P2 else z=0

If (x>10) and (z>0) then u=1

Q1 Q2 else u=0

- Design test cases for P1, P2 and Q1, Q2
- Each P1 & P2 and Q1 & Q2 for 4 conditions in pairs

P1	P2				Q1	Q2
x>0	y<1	x	y	z	x>10	z>0
T	T	15	0	1	T	T
T	F	15	5	0	T	F
F	T	-1	0	0	F	T
F	F	-1	5	0	F	F

## Salient Features

- Very demanding testing technique
- Frequently used with high reliability system requirements
- Non-executable combination may exist

## Control Flow Graph

- A control Flow Graph consists of Nodes and Edges. Edges are between nodes and are directed
- A Node: A statement (i.e. executable atomic entity in a program)
- An assignment statement
- An input/output statement
- Predicate of a condition
- An Edge: An edge represents a flow of control between two nodes/statements
- A control flow graph can be used to represent nodes with software modules or functions to depict a full functionality
- 

**Loop testing** is a type of software testing that focuses on the validation of loops within the code. The primary goal is to ensure that loops function correctly under various conditions and edge cases. This type of testing is crucial because loops are often a source of errors, especially when dealing with complex logic or large datasets.

## Example of Loop Testing

Consider a simple function in Python that calculates the factorial of a number using a loop:

```
def factorial(n):
```

```
    result = 1
```

```
    for i in range(1, n + 1):
```

```
        result *= i
```

```
    return result
```

## Test Cases for Loop Testing:

### 1. Test Case 1: Zero Iterations

- **Input:**  $n = 0$
- **Expected Output:** 1
- **Executed Loop:**
  - The loop for  $i$  in range(1,  $n + 1$ ) does not execute because  $n = 0$ .

### 2. Test Case 2: One Iteration

- **Input:**  $n = 1$
- **Expected Output:** 1
- **Executed Loop:**
  - The loop executes once with  $i = 1$ .

### 3. Test Case 3: Multiple Iterations

- **Input:**  $n = 5$
- **Expected Output:** 120
- **Executed Loop:**
  - The loop executes five times with  $i$  taking values from 1 to 5.
  - Iteration 1:  $\text{result} = 1 * 1 = 1$
  - Iteration 2:  $\text{result} = 1 * 2 = 2$
  - Iteration 3:  $\text{result} = 2 * 3 = 6$
  - Iteration 4:  $\text{result} = 6 * 4 = 24$
  - Iteration 5:  $\text{result} = 24 * 5 = 120$

In this example, the test cases cover different scenarios for the loop: zero iterations, one iteration, and multiple iterations. This helps ensure that the loop behaves correctly under various conditions.

**Path testing** is a type of software testing that involves ensuring that all possible paths through a given piece of code are executed at least once. The goal is to verify that every potential route through the code works correctly and produces the expected outcomes. This type of testing is particularly useful for identifying logic errors and ensuring comprehensive coverage of the code.

## Example:

Consider a simple function in Python that checks if a number is even or odd and then performs an additional check if the number is positive:

```
def check_number(num):  
    if num % 2 == 0:  
        if num > 0:  
            return "Even and Positive"  
        else:  
            return "Even and Non-Positive"  
    else:  
        if num > 0:  
            return "Odd and Positive"  
        else:  
            return "Odd and Non-Positive"
```

## Paths and Test Cases for Path Testing:

### 1. Path 1:

- **Condition:**  $\text{num} \% 2 == 0$  and  $\text{num} > 0$
- **Input:**  $\text{num} = 4$
- **Expected Output:** "Even and Positive"

### 2. Path 2:

- **Condition:**  $\text{num} \% 2 == 0$  and  $\text{num} \leq 0$
- **Input:**  $\text{num} = -2$

- **Expected Output:** "Even and Non-Positive"

### 3. Path 3:

- **Condition:**  $\text{num} \% 2 \neq 0$  and  $\text{num} > 0$
- **Input:**  $\text{num} = 3$
- **Expected Output:** "Odd and Positive"

### 4. Path 4:

- **Condition:**  $\text{num} \% 2 \neq 0$  and  $\text{num} \leq 0$
- **Input:**  $\text{num} = -1$
- **Expected Output:** "Odd and Non-Positive"

In this example, the test cases cover all possible paths through the `check_number` function, ensuring that each logical route is tested and verified.

## Code Coverage

Given a set of test cases for a program, they constitute **Node Coverage** if, when executed on the program, every node in the program graph is traversed. Denote this level of coverage as  $G_{\text{node}}$ , where the  $G$  stands for program graph.

- Given a set of test cases for a program, they constitute **Edge Coverage** if, when executed on the program, every edge in the program graph is traversed. Denote this level of coverage as  $G_{\text{edge}}$ .

**McCabe Path Testing**, also known as Basis Path Testing, is a structured testing methodology developed by Thomas McCabe. It uses the cyclomatic complexity metric to determine the number of independent paths through a program's source code. This approach ensures thorough testing by focusing on the control flow structure of the software.

### Example:

Consider a simple function in Python that checks if a number is positive, negative, or zero:



```
def check_number(num):
```

```
    if num > 0:
```

```
        return "Positive"
```

```
    elif num < 0:
```

```
        return "Negative"
```

```
    else:
```

```
        return "Zero"
```

## Cyclomatic Complexity Calculation:

- Number of edges (E): 5
- Number of nodes (N): 4
- Number of connected components (P): 1

Using the formula:

$$V(G) = E - N + 2P$$

$$= 5 - 4 + 2$$

$$= 3$$

The cyclomatic complexity is 3, indicating three independent paths.

## Independent Paths and Test Cases:

### 1. Path 1:

- **Condition:** num > 0
- **Input:** num = 5
- **Expected Output:** "Positive"

### 2. Path 2:

- **Condition:** num < 0
- **Input:** num = -3
- **Expected Output:** "Negative"

### 3. Path 3:

- **Condition:** num == 0

- **Input:** num = 0
- **Expected Output:** "Zero"

This example demonstrates how McCabe Path Testing ensures that all possible paths through the check\_number function are tested, providing comprehensive coverage of the code.

## Data Flow Testing

- Data Flow testing refers to forms of structural testing that focus on the point at which variables receive values and the point at which these values are used (or referenced)
- Data Flow testing serves as a “reality check” on path testing

Data Flow testing provides,

- A set of basic definitions
- A unifying structure of test coverage metrics

## Define/Use (Reference) Anomalies:

- A variable that is defined but never used (referenced)
- A variable that is used but never defined
- A variable that is defined twice before it is used
- Static Analysis: Finding faults in code without executing it

## D-U (Define/Use) Testing –

### Definitions :

Program P has a program graph  $G(P)$  and a set of program variables  $V$

- $G(P)$  has single entry and single exit
- Set of all paths in P is  $PATHS(P)$
- P – Program
- $G(P)$  – Graph of the Program

Nodes are program statements, and edges represent flow of control

- V - Variables
- PATHS(P) - Set of all paths

## Example:

Consider a simple function in Python that calculates the sum of an array of numbers:

```
def sum_array(arr):  
    total = 0 # Definition of 'total'  
    for num in arr:  
        total += num # Use of 'total'  
    return total # Use of 'total'
```

## Define-Use Pairs:

### 1. Definition:

- total = 0 (Line 2)

### 2. Uses:

- total += num (Line 4)
- return total (Line 5)

## Test Cases for Define-Use Testing:

### 1. Test Case 1:

- **Input:** arr = [1, 2, 3, 4, 5]
- **Expected Output:** 15
- **Executed Define-Use Pairs:**
  - total is defined at Line 2 and used at Lines 4 and 5.

### 2. Test Case 2:

- **Input:** arr = []
- **Expected Output:** 0
- **Executed Define-Use Pairs:**

- total is defined at Line 2 and used at Line 5 (since the loop does not execute).

In this example, the test cases ensure that the variable total is properly defined before it is used and that it is used correctly throughout the function. This helps identify any potential issues related to variable definitions and uses.

In the context of define-use testing, **defining nodes** and **usage nodes** refer to specific points in the code where variables are defined (assigned a value) and used (referenced), respectively.

## Defining Node

A defining node is a point in the code where a variable is assigned a value. This is where the variable gets its initial or updated value.

### Example:

```
def calculate_area(radius):
```

```
    pi = 3.14 # Defining node for 'pi'
```

```
    area = pi * radius * radius # Defining node for 'area'
```

```
    return area
```

In this example:

- pi = 3.14 is a defining node for the variable pi.
- area = pi \* radius \* radius is a defining node for the variable area.

## Usage Node

A usage node is a point in the code where a variable is referenced or used. This is where the variable's value is utilized in some operation or decision.

### Example:

```
def calculate_area(radius):
```

```
    pi = 3.14
```

```
    area = pi * radius * radius
```

```
    return area # Usage node for 'area'
```

In this example:

- return area is a usage node for the variable area.

## Define-Use Pair

A define-use pair consists of a defining node and a corresponding usage node for the same variable. Define-use testing ensures that every variable is properly defined before it is used and that it is used correctly throughout the code.

### Example:

```
def calculate_area(radius):  
    pi = 3.14 # Defining node for 'pi'  
    area = pi * radius * radius # Defining node for 'area'  
    return area # Usage node for 'area'
```

In this example:

- The define-use pair for pi is pi = 3.14 (defining node) and pi \* radius \* radius (usage within the expression).
- The define-use pair for area is area = pi \* radius \* radius (defining node) and return area (usage node).

Define-use testing helps identify potential issues related to variable definitions and uses, ensuring that variables are correctly handled throughout the code.

## Salient Features

- Very demanding and effort intensive
- Requires effort to design test cases
- Best used where reliability requirement is high
- Capability of detecting good defects

## Code Based Testing

- Statement Testing
- Branch Testing
- Multiple Condition Testing
- Loop Testing
- Path Testing

- Modified Path Testing (McCabe Path)
- Dataflow Testing
- Transaction Flow Testing

## Act of Measurement

Measurement is the first step that leads to control and eventually to improvement. If you can't measure something, you can't understand it. If you can't understand it, you can't control it. If you can't control it, you can't improve it.

- Measure
- Understand
- Control
- Improve



## Millers Test Coverage Metrics

Metric	Description of Coverage
$C_0$	Every Statement
$C_1$	Every DD-Path (DD-Path = Decision to Decision Path)
$C_{1p}$	Every predicate to each outcome
$C_2$	$C_1$ coverage + loop coverage
$C_d$	$C_1$ coverage + every dependent pair of DD-paths
$C_{MCC}$	Multiple Condition Coverage
$C_{ik}$	Every program path that contains up to $k$ repetitions of a loop (usually $k=2$ )
$C_{stat}$	"Statistically significant" fraction of paths
$C_{infinity}$	All possible execution paths

## Day 7: Model Based Testing

Model-based testing (MBT) is a testing methodology that uses models to represent the desired behavior of a system under test. These models act as blueprints or simulations of the software's functioning, and they are used to automatically generate test cases.

MBT is a black box testing uses requirement model for test case generation. Uses UML state diagrams.

Sequence of steps

- Model the system
- Identify the threads of system behavior in the model
- Transform threads into test cases
- Execute the test cases (on actual system) and record the results
- Revise the model(s) as needed and repeat the process

Two Fundamental types of Requirement Specification Models

Describing Structure: What system **is**

- Data Flow Diagram
- E-R Models
- Hierarchy Charts
- Class and Object Diagram

2. Describing Behaviour: What system **Does**

- Decision Table
- **Finite State Machines**
- **State Charts**
- **Petri Nets**

## Finite State Machines (FSMs)

Finite State Machines are mathematical models that represent a system's states and transitions. They are especially useful in MBT for modeling software with distinct states and transitions based on inputs. An FSM consists of:

- **States:** Different conditions or modes of the system.
- **Transitions:** Changes from one state to another based on events or conditions.
- **Initial State:** The starting point of the system.
- **Accepting States:** States that signify the successful completion of a process.

## Petri Nets

Petri nets are a more advanced modeling technique used for concurrent and distributed systems. They help visualize and analyze the flow of information and control. A Petri net consists of:

- **Places:** Represent conditions or states.
- **Transitions:** Represent events that may occur, changing the states.
- **Tokens:** Represent the presence or absence of a condition. Tokens move from place to place through transitions.
- **Arcs:** Connect places to transitions or transitions to places.

Petri nets are particularly powerful for modeling and analyzing systems with concurrent processes and resource sharing.

## State Charts

State charts are an extension of finite state machines, introduced to handle more complex scenarios with hierarchical states and parallel transitions. They are used to model reactive systems. State charts extend FSMs with:

- **Hierarchical States:** States within states, allowing for a more organized representation of complex systems.
- **Orthogonal Regions:** Parallel states that can operate independently.
- **Events and Actions:** Triggers that cause transitions and activities that occur due to transitions or states.

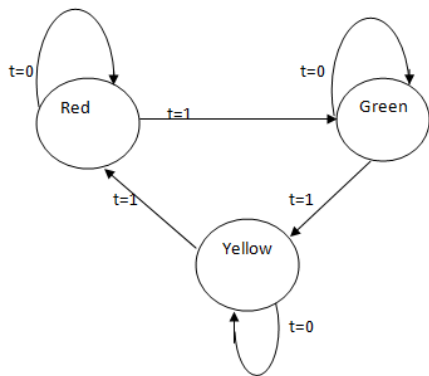
State charts are often used in systems engineering and software design to model behaviours that cannot be easily represented by simple FSMs.



## Finite State Machine (FSM) Example

For a traffic light, we have three states: **Red**, **Green**, and **Yellow**. The transitions between these states occur based on a timer.

1. **Red State:** The traffic light starts in the Red state. After a certain amount of time, it transitions to the Green state.
2. **Green State:** The light turns Green, allowing cars to pass. After a set time, it transitions to the Yellow state.
3. **Yellow State:** The light turns Yellow, indicating that it will soon turn Red. After a short duration, it transitions back to the Red state.



t	Current State	Next State
0	R: Red	R: Red
1	R: Red	G: Green
0	G: Green	G: Green
1	G: Green	Y: Yellow
0	Y: Yellow	Y: Yellow
1	Y: Yellow	R: Red

## Petri Net Example

For the same traffic light system, we use a Petri net to model concurrent processes and resource sharing. The Petri net consists of places, transitions, tokens, and arcs.

1. **Places:** Represent the Red, Green, and Yellow states.
2. **Transitions:** Represent the change from one light to another based on the timer.
3. **Tokens:** Indicate which light is currently active.

In this Petri net, a token starts in the Red place. When the timer fires, the token moves to Green, then to Yellow, and back to Red. Here's a simplified representation:

## State Chart Example

In a State chart, we can model the traffic light system with more complexity, such as handling emergency states or pedestrian signals.

1. **Hierarchical States:** The traffic light has a main state (Normal Operation) and a sub-state (Emergency Mode) that overrides normal behavior.

2. **Normal Operation:** The traffic light cycles through Red, Green, and Yellow states as described above.
3. **Emergency Mode:** If an emergency vehicle is detected, the traffic light switches to Emergency Mode, prioritizing the route for the emergency vehicle.

```
[Normal Operation]
|
v
[Red] -- (timer) --> [Green] -- (timer) --> [Yellow] -- (timer) --> [Red]
|
v
[Emergency Mode]
|
v
[Green for Emergency Vehicle]
```

## Mealy and Moore Machine

Feature	Mealy Machine	Moore Machine
Output	Depends on current state and current input	Depends only on current state
State Diagram	Outputs shown on transitions	Outputs shown inside states
Response to Inputs	Faster response to inputs	More stable output, but slower response
Complexity	Often simpler state diagrams	Often more complex state diagrams
Output Timing	Outputs change immediately with input changes	Outputs change only on state transitions
Design Flexibility	Can handle more immediate reaction to inputs	More predictable and easier to debug