



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# Module 1: Agenda

## Module 1: Introduction to Software Testing & Techniques

Topic 1.1

Introduction to Software Testing

Topic 1.2

Overview of the Course

Topic 1.3

Software Testing Techniques

Topic 1.4

Software Testing – Quality Attributes, Types & Levels



# **Topic 1.1: Introduction to Software Testing**

# Software Testing Methodologies

---



## Software Testing & Methodologies

- What is your view?
- What do you think it is all about?
- What do you wish to learn?
- Why do you want to learn?
- Is it important? How important is it?
- Why do you do it?
- What does it tell you?
- Is there a career in it?

# A definition

---

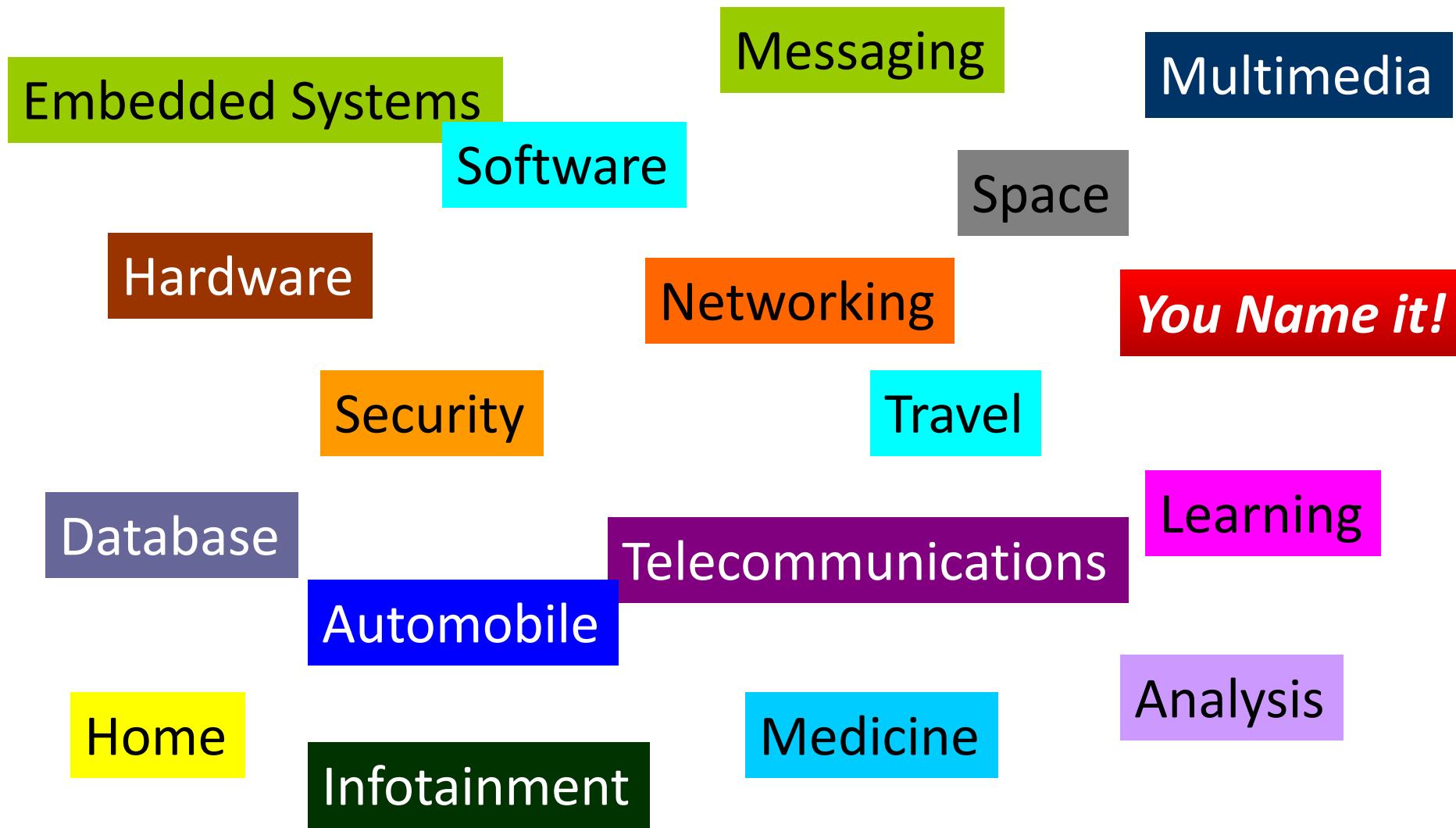
Testing is a process of executing a program with the intent of finding errors

# Why do we test?

---

- Make a judgement about quality or acceptability
- To discover Problems

# What world are we talking of?



# What world are we talking of?



Embedded Systems

Messaging

Multimedia

Software is (nearly)  
everywhere. Thus we are  
talking about that  
“everything”.

Data

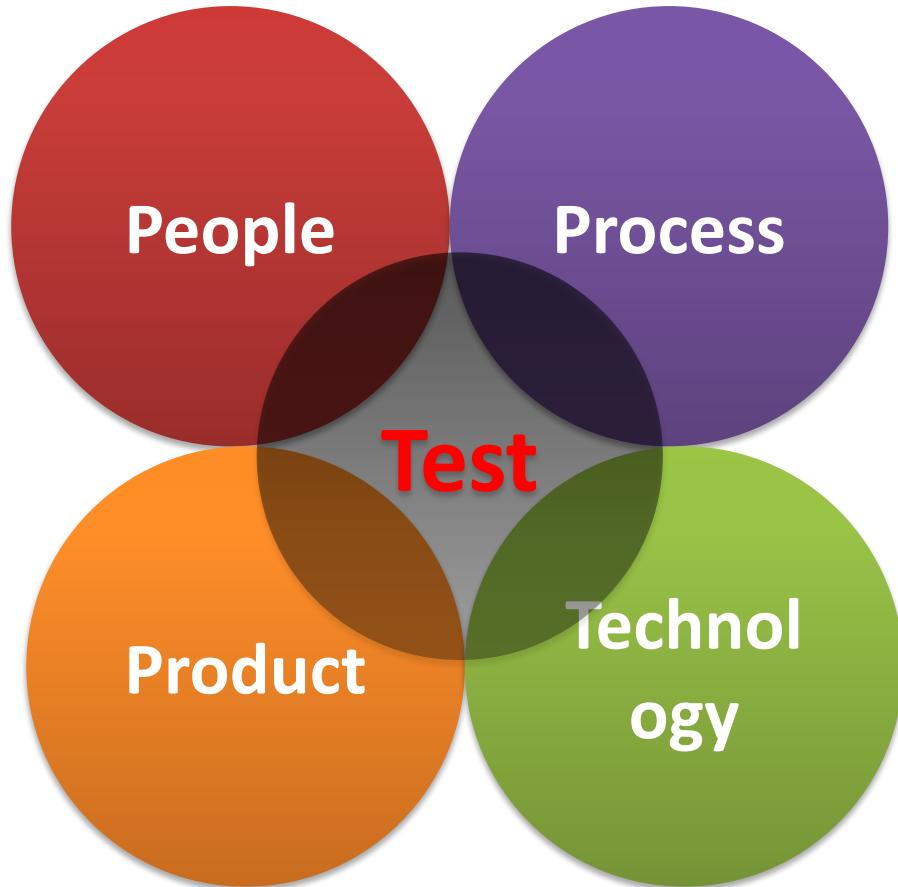
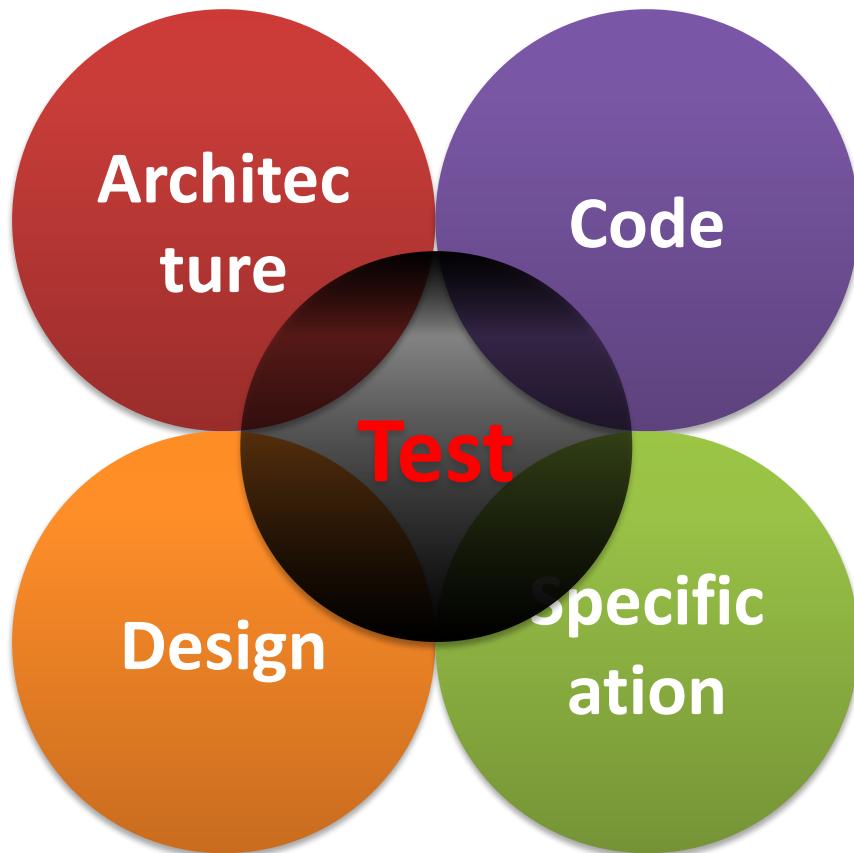
Home

Infotainment

Medicine

Analysis

# Building Blocks



# Psychology of Test

---

## Psychology of test

- Attitude to break the system
- Constructively destructive
- How come it works!

## The Medical Lab

- Reports normal  Cannot detect the problem
- A failure would trigger the way to treat

We do not hope for failure we work for failures not to occur

# Product

---

## Product Under Test

- Software is not alone
- Works with various systems
- Various systems work with each other

## Most vital

- System must work right, always!
- Available, always!

# Process

---

## Steps to success

- Increase probability of success
- Bring generality and consistency
- Measure: What you cannot measure you cannot control!
- Continuously improve

Even making tea is a process; To make good tea requires good process

# Technology

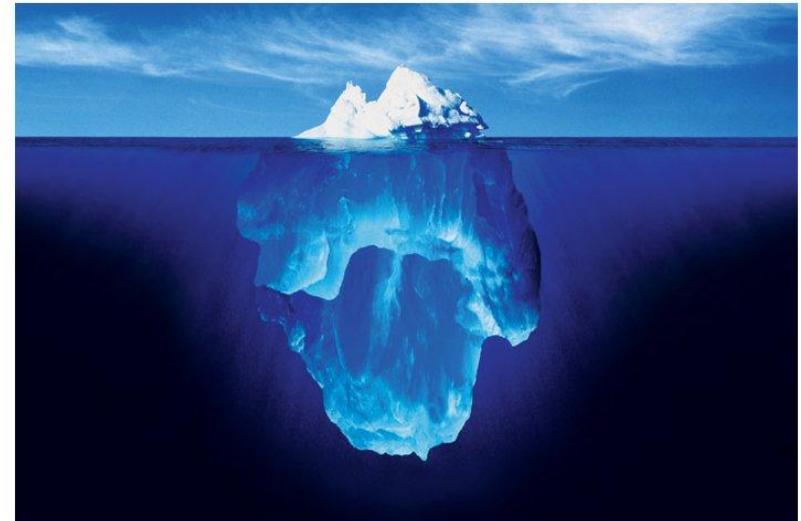
---

Something all of us look for and  
wish to excel

**Software Testing is a *highly*  
*technical* activity**

# Depth of STM

- Lets look at the depth of the subject
- Lets explore
- Lets look at the facets



# Views of STEM



# Software Engineering

---

## IEEE Definition

- Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of Software; that is the application of engineering to software (2) The study of approaches as in (1).
- Software Engineering is the establishment and use of a sound engineering principle in order to obtain economically software that is reliable and works efficiently on real machine. **[Fritz Bauer]**
- **Where did Software Engineering come from?**
- **Rather why was there a thought of Software Engineering?**

# Software Engineering

---

Software Engineering came in the 60s; with attempts to develop large software systems various problems occurred

- Cost Overruns
- Late delivery
- Lack of reliability
- Inefficient Systems
- Performance problems
- Lack of usability

The aim was to overcome the above issues and much more...



# Goals of Software Engineering

1. To improve quality of software
2. To improve the productivity of developers and software teams

And many more...

**Our focus is on 1 as a part of our course**

# Software Quality Attributes

- **Reliability**
- **Efficiency**
  - Speed
  - Resource management
- **Usability**
  - User friendliness
  - Intuitive
- **Maintainability:** Should be easy to maintain
- **Scalability:** Should scale as per the requirements of the user
- **Portability:** Should work on various platforms/systems/hardware
- **Security:** Should be secure from attacks
- **Testability:** Should be testable

# Software Quality Attributes

---

- What is important to User?
- What is important to the Engineers?
- What is important for Mission Critical System and a Word Processor?
  - Compare say Space Shuttle software and a Text Editor



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# Topic 1.2: Overview of the Course

# Learning Principles

---

Make everything as simple as possible, but not simpler

- Albert Einstein

I hear and I forget, I see and I remember, I do and I understand

- Confucius

“... You can know the name of a bird in all the languages of the world, but when you’re finished, you’ll know absolutely nothing whatever about the bird. You’ll only know about humans in different places, and what they call the bird, So let’s look at the bird and see what it’s doing – that’s what counts”

– Richard P Feynman (Book: What Do You Care What Other People Think)

# Building Blocks – A View

SW Products  
SW Test Tools

Test Research

## Test Techniques Application

Functional, Behavioral, IOT, Usability, Integration, Unit, Performance, Stress

## Test Techniques Development

EC, BVA, DT, CEG, McCabe, OATS, Data Flow, Control Flow

### Math

Set theory  
Discrete  
Combinatorial

### Concepts

SW Arch, Design, Data Structures

### Problem

Code coverage  
Quality  
Zero Defects

# Ask the Questions!

- What

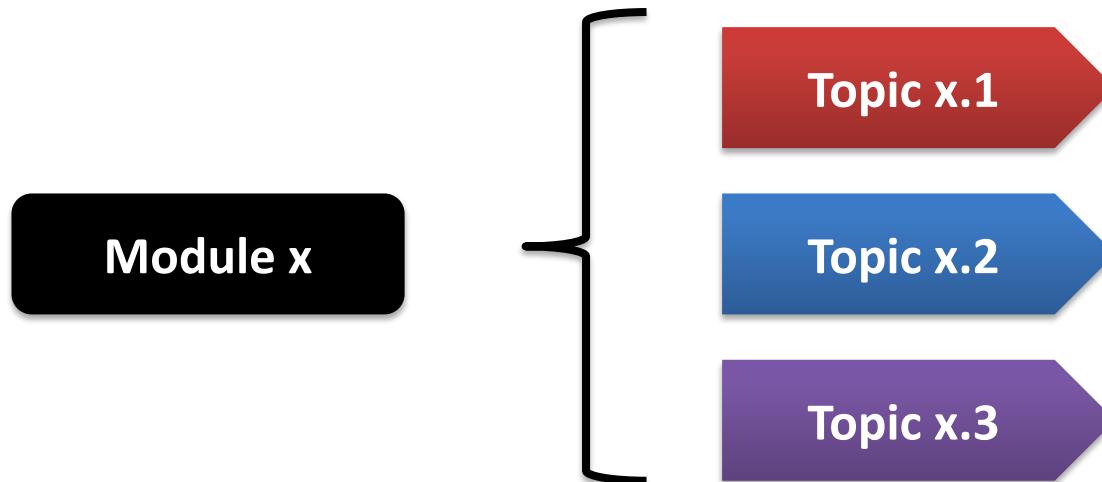
- When

- How

- Why

# Our Course Structure

- Modules
  - Course is divided into Modules
- Topics
  - Each module is divided into Topics



- Self Study & Quick Review
  - Every Module has Self Study & a Quick Review

# Modules

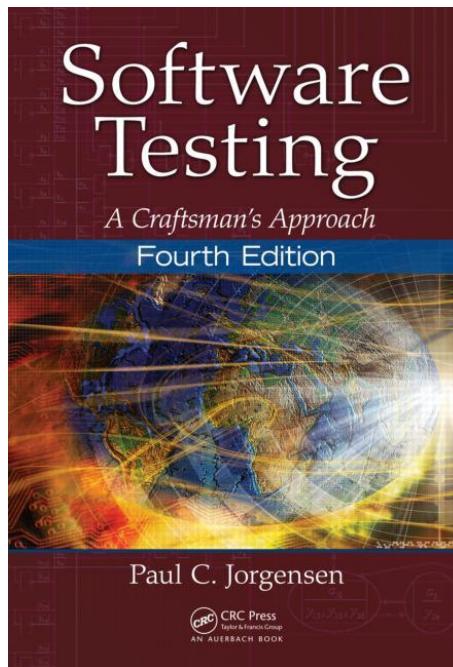
Module No	Module Title	Objectives
1	Introduction to Software Testing & Techniques	Introduce the course and course handout. Bring a perspective of need and motivation for this course. Provide an overview of the course, quality attributes, levels and types of Testing
2	Mathematics and Formal Methods	Provide a base to the software testing techniques in form of mathematics and formal methods. Review topics of permutation/combination, discrete mathematics and graph theory. Focus is on the relevance to software testing.
3	Specification Based Testing	Bring an approach to look at the system from specification perspective. Learn the relevant techniques for testing specifications – Equivalence Class, Boundary Value Analysis, Combinatorial, Decision Tables and Domain Testing
4	Code Based Testing	Take a code level approach to testing and assuring quality. Learn the relevant techniques for testing code – Path Based Testing and Data Flow Testing
5	Model Based Testing	Introduce Model Based Testing. Various Model for Software testing, their choice and techniques. Learn Finite State Machine, Petri Nets and State Charts. Learn to use these to derive testing cases

# Modules

Module No	Module Title	Objectives
6	Object Oriented Testing	Understand the issues in OO Software Testing. Learn techniques and sublets of Unit, Integration and Systems Testing of OO Software. GUI Testing for OO Software
7	Integration & System Testing	Overview and need for Integration and Systems Testing of Software. Learn the techniques of Integration and Systems Testing
8	Life-Cycle Based Testing	Provide an overview from a life-cycle perspective of Software and Software Products. Agile Testing and Agile Model-Driven Development. Role of Test engineers in life-cycle-based testing
9	Test Adequacy & Enhancement	Learn the need for test adequacy and need for enhancement of test cases. Various techniques and criteria for measuring of test adequacy (data and control flow). Using the criteria to enhance test cases.
10	Test Case Minimization, Prioritization and Optimization	Explore and understand the need for minimization and prioritization. Review the regression test problem. Selection of test cases for regression.

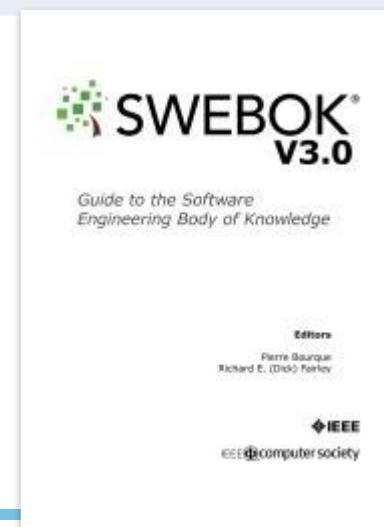
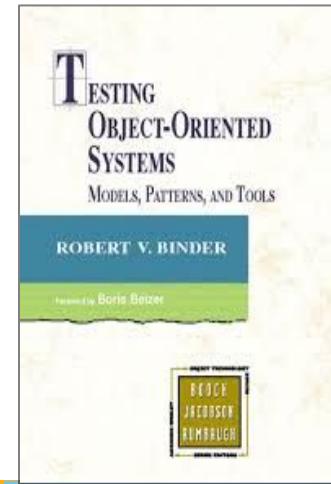
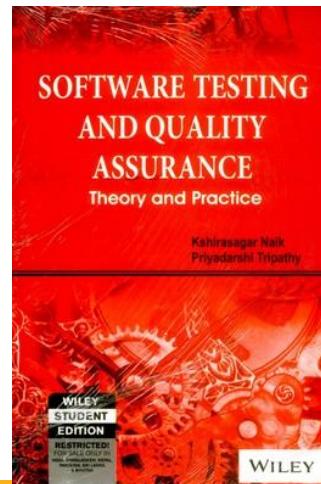
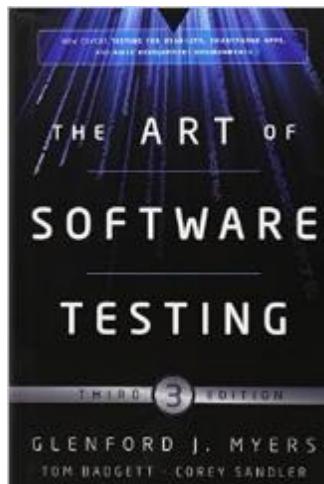
# Text Books

T1	Software Testing – A Craftsman's Approach, Fourth Edition, Paul C Jorgenson, CRC Press
T2	Foundations of Software Testing, Second Edition, Aditya P Mathur, Pearson



# References

- |    |   |
|----|---|
| R1 | The Art of Software Testing, Third Edition, Glenford J. Myers, Tom Badgett, Corey Sandler,                        |
| R2 | Software Testing and Quality Assurance – Theory and Practice, Kshirasagar Naik, Priyadarshi Tripathy, Wiley, 2013 |
| R3 | Testing Object Oriented Systems: Models, Patterns and Tools, Robert V Binder, Addison Wesley                      |
| R4 | Guide to Software Engineering Body of Knowledge, Version 3, IEEE  |





# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# **Topic 1.3: Software Test Techniques**

# Test Techniques

---

**Based on Engineers experience and intuition**

- Exploratory
- Ad-hoc

**Specification Based Techniques**

- Equivalence Partitioning
- Boundary Value Analysis
- Decision Tables
- ...

**Code Based Techniques**

- Control Flow
- Data Flow
- ...

# Test Techniques

---

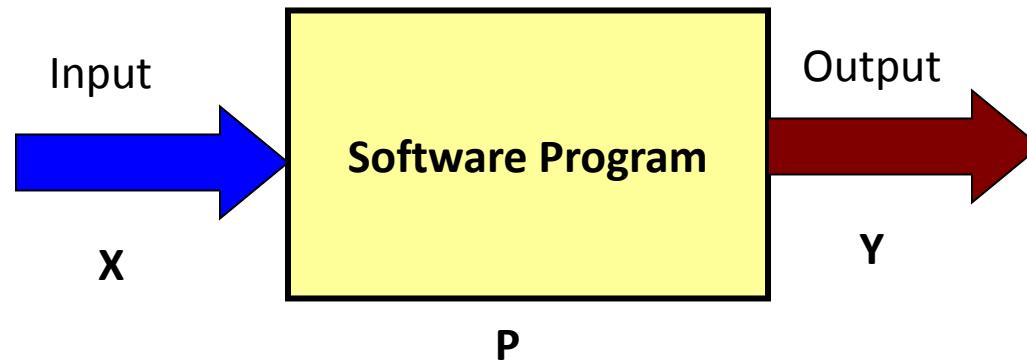
## Techniques based on nature of application

- Object Oriented
- Component-based
- Web-based
- GUI
- Protocol Conformance
- Real Time Systems

## Usage based testing

- Operational Profile
- Reliability Engineered Testing

# Testing Methods



## Testing methods

Based on the source of information used for testing

- No information is used
- Specification (Example: Requirements specification)
- Design or LLD or Source Code (Internals of a program)

# A Simple Example

---

```
#include <iostream> int main() { int a, b, c; // initialize the  
three number to zero a=b=c=0; int max=0; // Indicate the program  
purpose std::cout << "Program computes max of three integers" <<  
std::endl; // Ask for input of numbers std::cout << "Enter the  
values for a, b, & c one on each line" << std::endl; std::cin >>  
a; std::cin >> b; std::cin >> c; // Start with a as max max=a;  
//Logic to find the max. Compare with other two. if(a<b) { max =  
b; } if(max<c) { max=c; } // Output the result of the  
computation std::cout << "Max of three is " << max << std::endl;  
} // End of program
```

# A Simple Example

```
#include <iostream>

int
main()
{
    int a, b, c;
    // initialize the three number to zero
    a=b=c=0;
    int max=0;

    // Indicate the program purpose
    std::cout << "Program computes max of three integers" << std::endl;

    // Ask for input of numbers
    std::cout << "Enter the values for a, b, & c one on each line" << std::endl;
    std::cin >> a;
    std::cin >> b;
    std::cin >> c;

    // Start with a as max
    max=a;

    //Logic to find the max. Compare with other two.
    if(a<b) {
        max = b;
    }
    if(max<c) {
        max=c;
    }

    // Output the result of the computation
    std::cout << "Max of three is " << max << std::endl;
}

// End of program
```

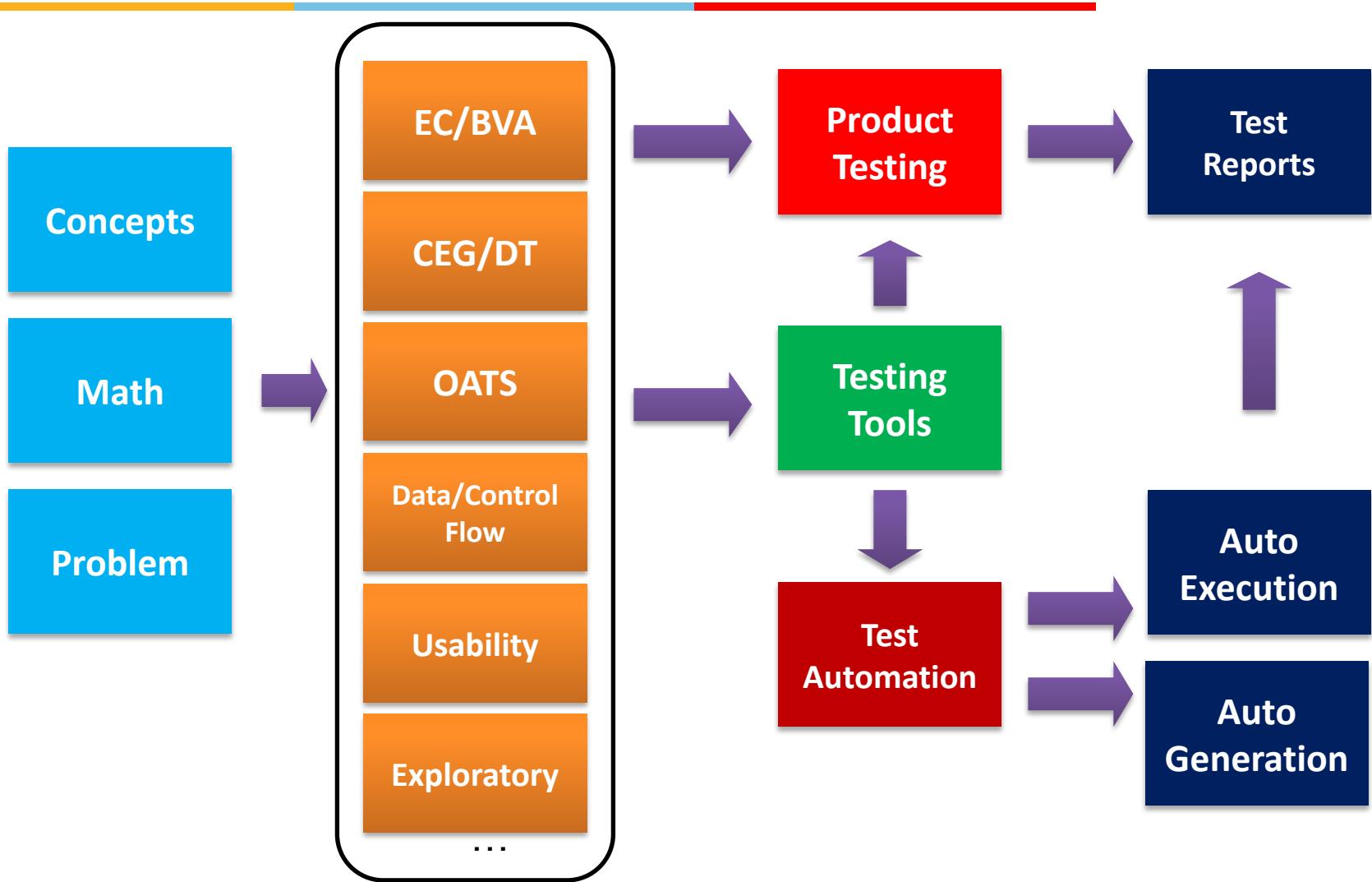
# A Simple Example – Output

---

```
~/dev/stm $ ./a.out
Program computes max of three integers
Enter the values for a, b, & c one on each line
3
4
5
Max of three is 5
~/dev/stm $ ./a.out
Program computes max of three integers
Enter the values for a, b, & c one on each line
34
54
89
Max of three is 89
~/dev/stm $ ./a.out
Program computes max of three integers
Enter the values for a, b, & c one on each line
56
57
99
Max of three is 99
~/dev/stm$
```

---

# Progression





# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# **Topic 1.4: Software Testing – Quality Attributes, Types & Levels**

# High Level Design

---

## Modular

- Structure chart based
- Broken down into various modules and has call relationships i.e. (set of modules + call relationships)
- Executes in a sequence
- Uses a modular design pattern

## Object Oriented Design

- Class diagram = Set of Classes + relationships
- Inheritance
- Association
- Aggregation

# Low Level Design

---

## Low Level Design

- Major Algorithms
- Data structures

## Implementation

- Choice of programming language
- Coding
- Low level algorithms
- Low level data structures
- Specific code constructs

# Basic Definitions

## Error

- An error is a mistake. Errors propagate. A requirements error may (will) get magnified as design and still amplified in later phases

## Fault

- A fault is a result of an error. Fault aka. Defect, is an expression of error, where representation is a mode of expression ex: narrative text, dataflow diagrams etc
  - Faults of omission: Occurs when something is missed out
  - Faults of commission: Occurs when some representation is incorrect

## Failure

- A failure occurs when a fault executes.
  - How do we relate this to faults of commission and omission?

# Basic Definitions

---

## Incident

- An incident occurs when a failure occurs. An incident is the symptom associated with a failure that alerts the user the occurrence of a failure

## Test

- Testing is concerned with errors, faults, failures and incidents. A test is an act of exercising testing cases with two goals,
  - to find failures
  - to demonstrate correct execution

## Test Case

- A test case has an identity and is associated with a program behaviour. A test case also has a set of inputs and a list of expected outputs

# Good Test Case

---

- High probability of finding a defect which is yet to be discovered
- It is not redundant
- “Best of the Breed”.
- Neither too simple nor too complex

# A Test Case

---

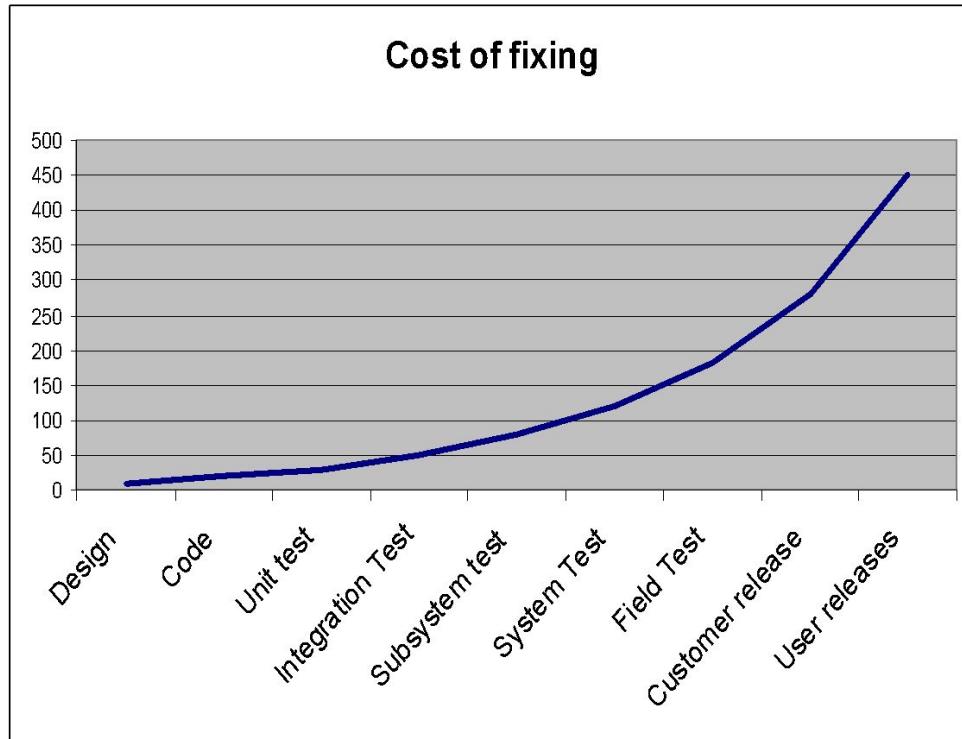
- **A unique ID**
- **An input**
  - Precondition: Circumstances that hold prior to execution of the test case
  - Actual inputs: The actual inputs for a particular test case/method
- **Expected Output**
  - Postconditions: Circumstances that hold after the execution of the test
  - Actual outputs: The actual outputs
- **Verdict**: A final PASS/FAIL/INDETERMINATE statement for the test activity

# Goals of SW Testing

---

- To show that the software system satisfies the requirements and performs as expected. The requirements may be explicit or implicit.
  - Explicit: User Interface, Specified Output
  - Implicit: Error handling, performance, reliability, security
- To have “confidence” in the software system. To assure that the software works. To demonstrate that the Software works.
- To find defects
- To prevent defects
- Ensure software quality

# Cost of fixing



- Software Testing (exhaustive) is a very time consuming activity
- Software testing can be most expensive activity in Software development and maintenance

# Testing Types & Levels

---

## Test Levels based on Target of Test

- Unit testing
- Integration testing
- Sub-system testing
- System testing
- Acceptance testing
- Alpha/Beta Testing
- Field testing

## Types based on execution

- Dynamic Testing (Execution based testing)
- Static Testing (No execution is involved)

# Testing Types & Levels

---

## Test Levels based on Objective of Testing

- Acceptance
- Installation testing
- Alpha/Beta Testing
- Conformance/IOT
- Reliability
- Regression
- Performance
- Stress
- Usability

# Test Suites

---

- Test Suite is a set of test cases for a particular software system or a product
- A *typical* Test Suite contains
  - Random tests
  - Specification based tests
  - Code Based tests



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# STM MODULE 1

## ADDITIONAL CONTENT

# Levels of Testing

1. Unit Testing
2. Integration Testing
3. System Testing
4. Acceptance testing

# Unit Testing

- Purpose of Unit Testing is to ensure that module/component/function is performing the intended task error free.
- White box testing technique is used to detect the error(s) or malfunctioning while performing unit testing.
- Developers are responsible for finding and resolving the errors in units.

# Testing Methodologies

## 1. White Box testing:

- Knowledge of the internal program design and code required.
- Tests are based on coverage of code statements, branches, paths, conditions.
- It is also called as behavioral testing

## 2. Black Box Testing:

- No knowledge of internal program design or code required.
- Tests are based on requirements and functionality.
- This testing is also known as functional testing.

# Verification Testing

- Verification is the process of manually examining / reviewing a document. The document may be SRS, design document, code or any document prepared during any phase of software development.
- As per IEEE, “verification is the process of evaluating the system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase” [1]

# Validation Testing

- Validation testing is done to ensure that software product being developed is as per the requirements of the customer.
- Validation is achieved through V-Shape software development life cycle model.
- This is a dynamic testing which is accomplished using:  
Unit Testing,  
Integration Testing,  
System Testing &  
Acceptance Testing.

# Regression Testing

- Regression testing is the process of re-testing the modified parts of the software and ensuring that no new errors have been introduced into previously tested source code due to these modifications.
- Regression testing is used to test the modified source code and other parts of the source code that may be affected by the amendment.

# Reference

- [1] IEEE, “Standard Glossary of Software Engineering Terminology”, 2001.



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# Module 2: Agenda

---

## Module 2: Mathematics & Formal Methods

Topic 2.1

Permutations & Combinations

Topic 2.2

Propositional Logic

Topic 2.3

Discrete Math

Topic 2.4

Graph Theory

# Math for Test Engineers

---

- For Test Engineers – Know our focus
- Testing is a craft; math are the craftsman's tools
- Bring *Rigor, Precision* and *Efficiency*
- Our treatment of math
  - Largely informal – What is required for Test Engineers and not for mathematicians
  - Our focus is discrete mathematics
- Aim is
  - To make test engineers better at their craft



# **Topic 2.1: Permutations & Combinations**

# Permutation & Combination

---

- Selecting several things out of a larger group
- Two aspects to look at
  - Order
  - Repetition

# Combination

---

- Order does not matter
- Example: Fruits in a fruit salad. It does not matter in which the fruits are put into the salad. It could be Apple, Banana and Strawberry or any other order



# Permutation

---

- Order does matter
- Example: A lock which opens with a sequence of digits. We call it the combination lock. It is indeed a permutation lock
  - Sequence 437 (347 will never work!)

Indeed a permutation lock



# Permutation

---

- Repetition – Yes & No
- Example
  - Repetition allowed: Digits in the permutation lock may repeat like “333” or “557”
  - Repetition now allowed: First three standings in a running race

# Permutation

- Choosing r things out of n

## Repetition

- n possibilities for each of the r choices

$$n^r$$

$${}^n C_r$$

## Without Repetition

- Possibilities reduce with every selection

$$\frac{n!}{(n - r)!}$$

# Combination

Choosing r things out of n

**Repetition allowed**

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

**Without Repetition**

$$\frac{n!}{r! (n - r)!}$$



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



## Topic 2.2: Propositional Logic

# Propositional Logic

---

- A proposition is a sentence that is either TRUE or FALSE
- Given a proposition, it is always possible to tell if it is T or F
- Propositional logic has operations, expressions, and identities
- Logical Operators
- Logical Expressions
- Logical Equivalence

# Propositional Logic

<b>p</b>	<b>q</b>	<b><math>p \wedge q</math> (AND)</b>	<b><math>p \vee q</math> (OR)</b>	<b><math>\sim p</math> (NOT)</b>
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

<b>p</b>	<b>q</b>	<b><math>p+q</math> (EX-OR)</b>	<b><math>p \rightarrow q</math> (IF-THEN)</b>
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



## Topic 2.3: Discrete Math

# Propositional Logic

---

- A proposition is a sentence that is either TRUE or FALSE
- Given a proposition, it is always possible to tell if it is T or F
- Propositional logic has operations, expressions, and identities
- Logical Operators
- Logical Expressions
- Logical Equivalence

# Propositional Logic

<b>p</b>	<b>q</b>	<b><math>p \wedge q</math> (AND)</b>	<b><math>p \vee q</math> (OR)</b>	<b><math>\sim p</math> (NOT)</b>
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

<b>p</b>	<b>q</b>	<b><math>p+q</math> (EX-OR)</b>	<b><math>p \rightarrow q</math> (IF-THEN)</b>
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T



## Topic 2.3: Discrete Math

# Set Theory

Collection of things which have a common property

- Things that one wears (Specific activity wear)
- Sports kit for badminton
- Months in a year or months with 31 days

Listing  
Elements

$$Y = \{April, June, September, November\}$$

Decision rule

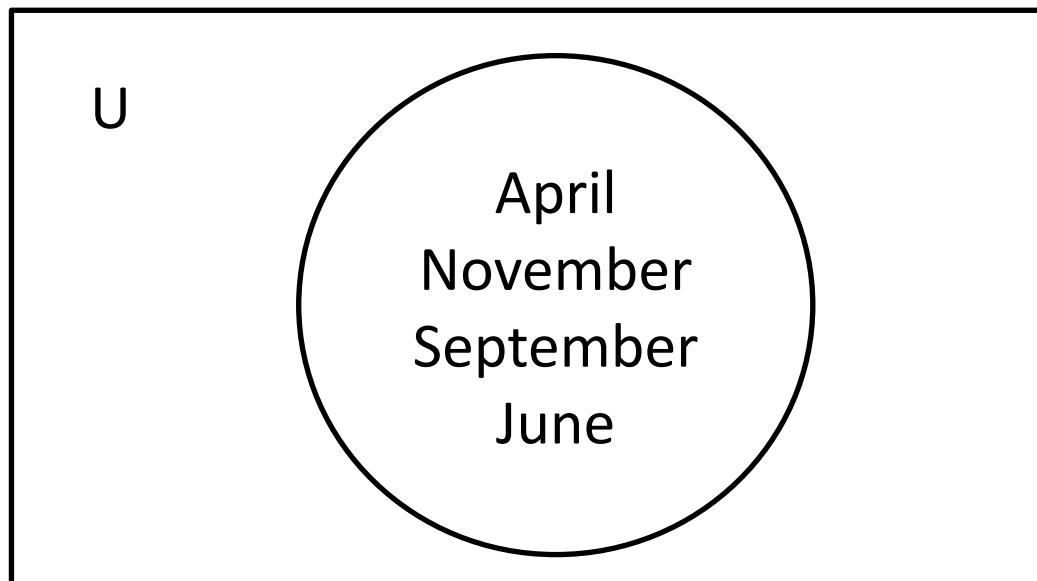
$$Y = \{year: 1800 \leq year \leq 2014\}$$

Decision rule

$$S = \{sales: \text{the } 15\% \text{ commission rate applies to the sale}\}$$

# Venn Diagrams

- Picture(s) for Sets
- A set of depicted as a circle with interior of the circle corresponds to the elements of the set



**Venn diagram of 30 day month**

# Set Operations

---

- Union is the set  $A \cup B = \{x: x \in A \vee x \in B\}$
- Intersection is the set  $A \cap B = \{x: x \in A \wedge x \in B\}$
- Complement of A is the set  $A' = \{x: x \notin A\}$
- Relative complement of B WRT A is the set  
$$A - B = \{x: x \in A \wedge x \notin B\}$$
- Symmetric difference of A and B is the set  
$$A \bigoplus B = \{x: x \in A \oplus x \in B\}$$

***Refer to Venn Diagrams of basic sets in T1***

# Set Operations

- Unordered pair  $(a, b)$
- Ordered pair  $< a, b >$

What is the difference?

- Cartesian Product

$$A \times B = \{< x, y > : x \in A \wedge y \in B\}$$

# Set Relations

---

- **Subset**
  - A is subset of B if and only if all elements of A are also in B
- **Proper subset**
  - A is a proper subset of B if and only if there is at least one element in B which is not in A
- **Equal Sets**
  - Each is a subset of the other

Look up the notations in the book T1 Chapter 3

# Set Partitions

$n^r$



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# Topic 2.4: Graph Theory

# Graph

---

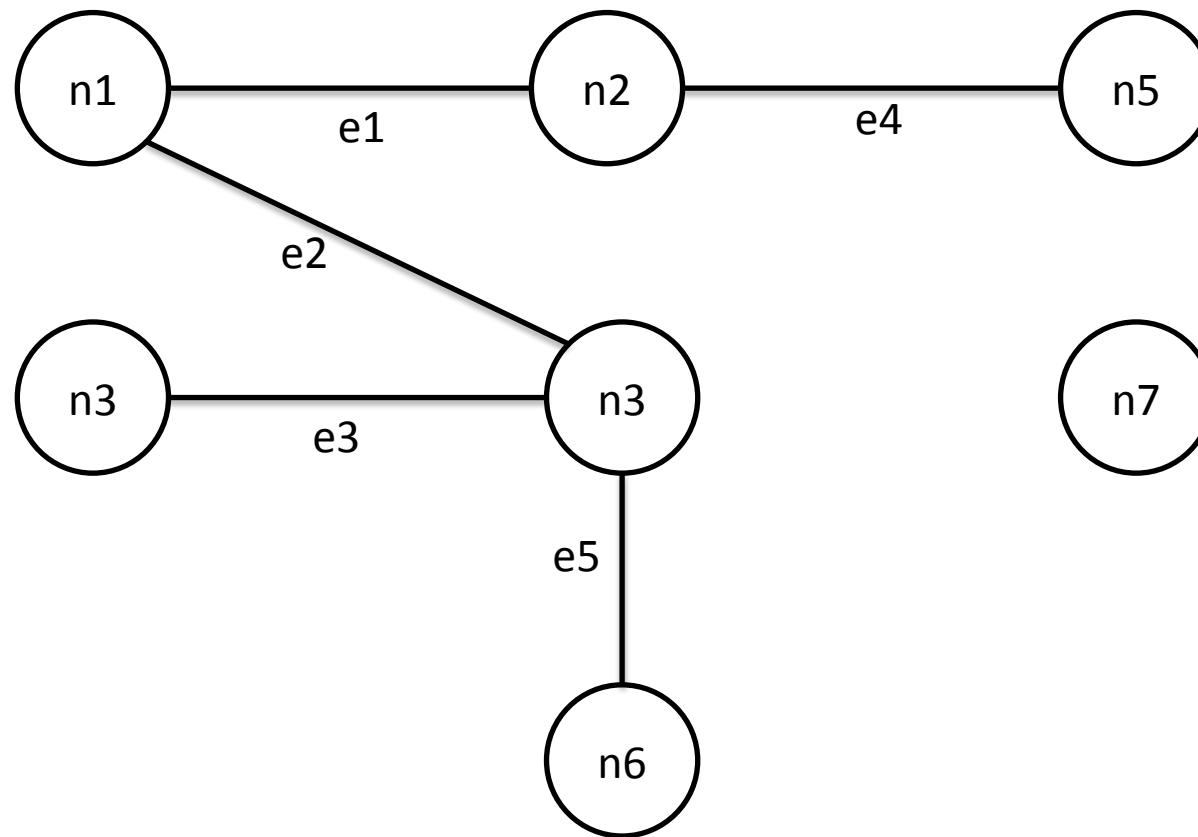
- A graph (also known as linear Graph) is an abstract mathematical structure defined from two sets – set of nodes and set of edges that form connections between nodes
- Example: Computer Network
- Definition
- *A Graph  $G = (V,E)$  is composed of a finite (and nonempty) set  $V$  of nodes and a set of  $E$  of unordered pairs of nodes*

$$V = \{n_1, n_2, n_3, \dots, n_m\}$$

$$E = \{e_1, e_2, e_3, \dots, e_p\}$$

# A Graph

- Nodes and Edges Sets
- Connection between nodes



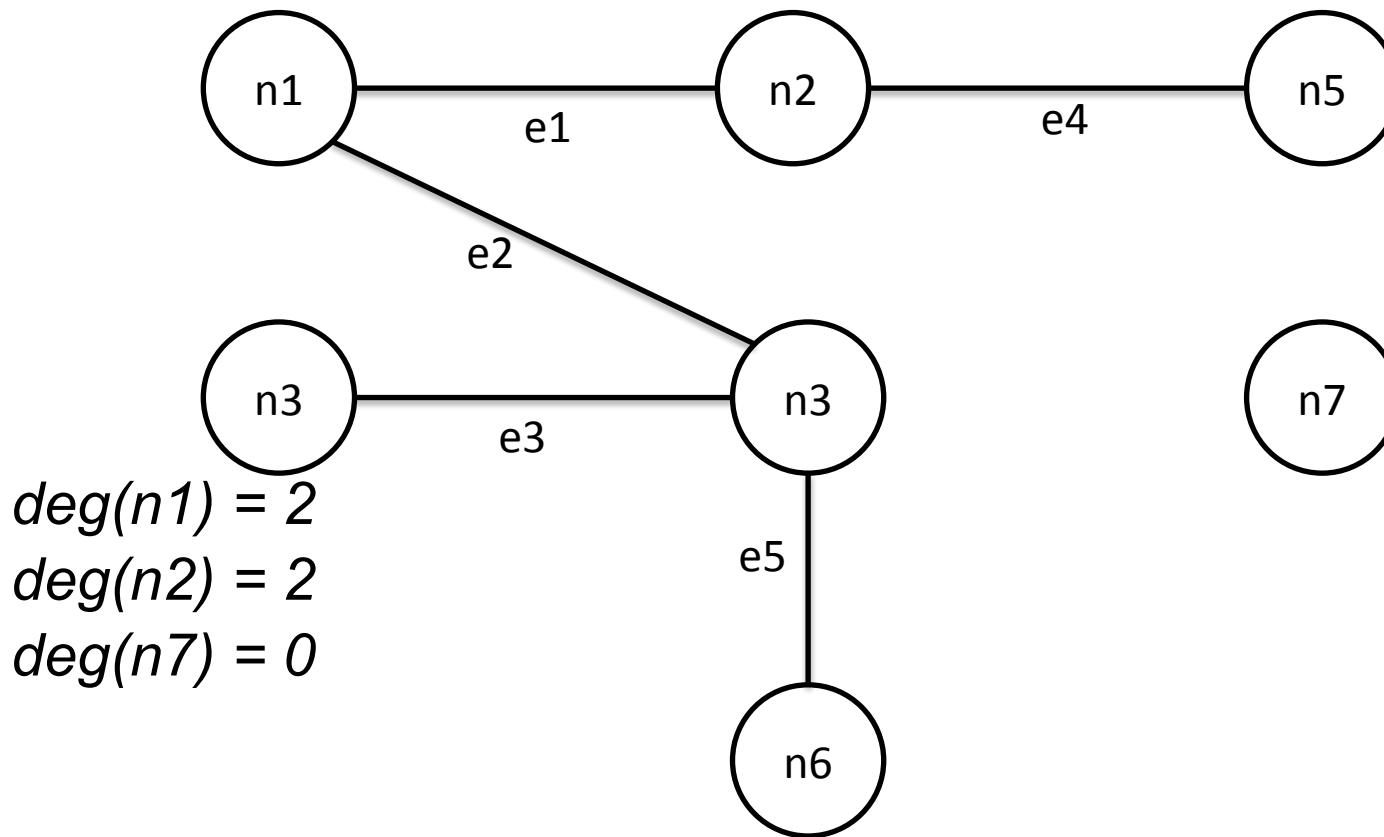
# Use of representation

---

- Nodes as program statements
- Edges
  - Flow of control
  - Define/use relationships

# Degree of a Node

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



# Use of Degree of Node

---

- Indicates Popularity
- Social scientists
  - Social interactions
  - Friendship/communicates with
- Example:
  - Graph with nodes are objects and edges are messages; degree can represent the extent of integration testing that is appropriate for the object

# 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



# Use of this representation

- Degree of node is zero
- Unreachable node

# 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

	$n1$	$n2$	$n3$	$n4$	$n5$	$N6$	$n7$
$n1$	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

# Use of this representation

---

- Deals with connections
- Useful for later graph theory concepts example: paths

# Paths

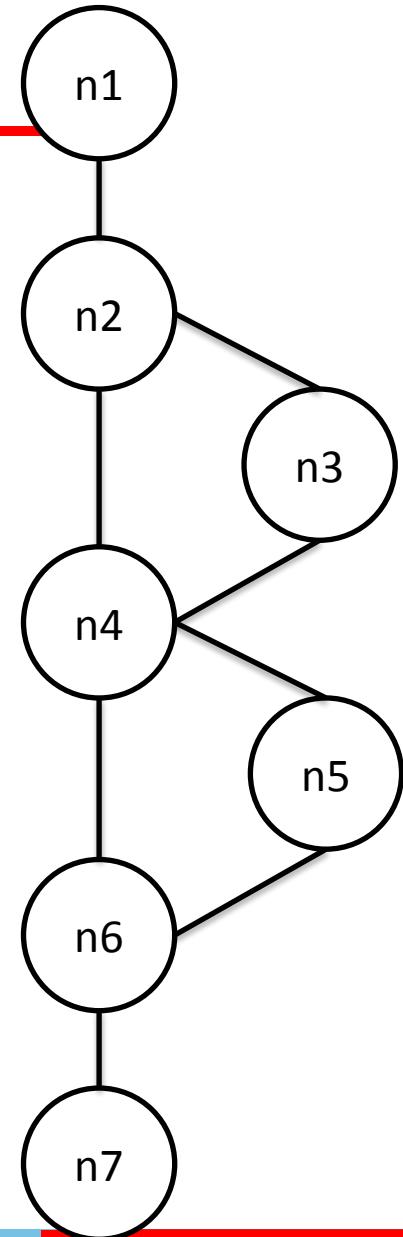
---

- 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

# Graph

- n1 represents a series of statements
- n7 also represents a series of statement
- Is this the correct representation?
- How does this help us get to testing?
- What does this help in?





# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

## **Set Theory**

The set is represented by listing all the elements comprising it. The elements are enclosed within braces and separated by commas.

Example 1 – Set of vowels in English alphabet,  $B = \{a, e, i, o, u\}$

Example 2 – Set of odd numbers less than 20,  $A = \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19\}$

## **Set Builder Notation**

The set is defined by specifying a property that elements of the set have in common. The set is described as  $P = \{x : a(x)\}$

Example 1 – The set {a, e, i, o, u}

is written as –

$P = \{y : y \text{ is a vowel in English alphabet}\}$

Example 2 – The set {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}

is written as –

$A = \{y : 1 \leq y < 20 \text{ and } (y \% 2) \neq 0\}$

If an element  $y$  is a member of any set  $A$ , it is denoted by  $y \in A$

and if an element  $x$  is not a member of set  $A$ , it is denoted by  $x \notin A$ .

## **Cardinality of a Set**

Cardinality of a set A, denoted by  $|A|$ , is the number of elements of the set.

The number is also referred as the cardinal number. If a set has an infinite number of elements, its cardinality is  $\infty$

Example –  $|\{1,2,3,4\}|=4$ ,  $|\{1,2,3,4,\dots\}|=\infty$ .

### **Types of Sets**

Sets can be classified into many types. Some of which are finite, infinite, subset, universal, proper set, etc.

#### **Finite Set**

A set which contains a definite number of elements is called a finite set.

Example –  $A = \{y \mid y \in N \text{ and } 50 > y > 40\}$ .

#### **Infinite Set**

A set which contains infinite number of elements is called an infinite set.

Example –  $A = \{y \mid y \in N \text{ and } y > 20\}$

#### **Subset**

A set B is a subset of set C (Written as  $B \subseteq C$ ) if every element of B is an element of set C.

Example 1 – Let,  $C = \{1,2,3,4,5,6\}$

and  $B = \{1,2\}$ .

Here set B is a subset of set C as all the elements of set B are in set C.

#### **Proper Subset**

The term “proper subset” can be defined as “subset of but not equal to”. A Set B is a proper subset of set C (Written as  $B \subset C$ ) if every element of B is an element of set C and  $|B| < |C|$ .

## **Universal Set**

It is a collection of all elements in a particular context or application. All the sets in that context or application are essentially subsets of this universal set. Universal sets are represented as U.

Example – We may define U as the set of all sport events. In this case, set of all format of cricket is a subset of U

, set of all lawn tennis is a subset of U

, and so on.

## **Empty Set or Null Set**

An empty set contains no elements. It is denoted by  $\emptyset$ .

As the number of elements in an empty set is finite, empty set is a finite set. The cardinality of empty set or null set is zero.

Example –  $S = \{ x \mid x \in N$

And  $7 < x < 8 \} = \emptyset$ .

## **Venn Diagrams**

In the late 1800's, an English logician named John Venn developed a method to represent relationship between sets. He represented these relationships using diagrams, which are now known as Venn diagrams. A Venn diagram represents a set as the interior of a circle. Often two or more circles are enclosed in a rectangle where the rectangle represents the universal set.

Venn Diagram in case of two elements

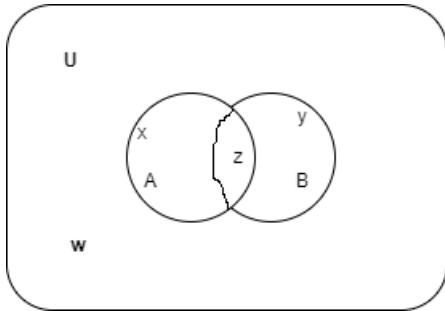
Where;

$x$  = number of elements that belong to set A only

$y$  = number of elements that belong to set B only

$z$  = number of elements that belong to set A and B both ( $A \cap B$ )

$w$  = number of elements that belong to none of the sets A or B is



From the above figure, it is clear that

$$n(A) = x + z ;$$

$$n(B) = y + z ;$$

$$n(A \cap B) = z;$$

$$n(A \cup B) = x + y + z.$$

$$\text{Total number of elements} = x + y + z + w.$$

### **Set Operations**

Set Operations include Set Union, Set Intersection, Set Difference, Complement of Set, and Cartesian Product.

#### **Set Union**

The union of sets A and B (denoted by  $A \cup B$ ) is the set of elements which are in A, in B, or in both A and B. Hence,  $A \cup B = \{x | x \in A \text{ OR } x \in B\}$ .

Example – If  $A = \{10, 11, 12, 13\}$

and  $B = \{13, 14, 15\}$

, then  $A \cup B = \{10, 11, 12, 13, 14, 15\}$

#### **Set Intersection**

The intersection of sets A and B (denoted by  $A \cap B$ ) is the set of elements which are in both A and B. Hence,  $A \cap B = \{x | x \in A \text{ AND } x \in B\}$ .

## **Set Difference/ Relative Complement**

The set difference of sets A and B (denoted by  $A-B$ ) is the set of elements which are only in A but not in B. Hence,  $A-B=\{x|x\in A \text{ AND } x\notin B\}$

Example – If  $A=\{10,11,12,13\}$

and  $B=\{13,14,15\}$ ,

then  $(A-B)=\{10,11,12\}$

and  $(B-A)=\{14,15\}$ .

Here, we can see  $(A-B)\neq(B-A)$

## **Cartesian Product / Cross Product**

The Cartesian product of n number of sets  $A_1, A_2, \dots, A_n$

denoted as  $A_1 \times A_2 \times \dots \times A_n$

can be defined as all possible ordered pairs  $(x_1, x_2, \dots, x_n)$

where  $x_1 \in A_1, x_2 \in A_2, \dots, x_n \in A_n$

Example – If we take two sets  $A=\{a,b\}$

and  $B=\{1,2\}$

The Cartesian product of A and B is written as –  $A \times B=\{(a,1),(a,2),(b,1),(b,2)\}$

The Cartesian product of B and A is written as –  $B \times A=\{(1,a),(1,b),(2,a),(2,b)\}$

## **Types of Graph:**

Directed Graph and Undirected graph.

Other types of graph

There are also some other types of graphs, which are described as follows:

**Null Graph:** A graph will be known as the null graph if it contains no edges. With the help of symbol  $N_n$ , we can denote the null graph of  $n$  vertices.

**Simple graph** or undirected graph.

**Multi-Graph:** A graph will be known as a multi-graph if the same sets of vertices contain multiple edges. In this type of graph, we can form a minimum of one loop or more than one edge.

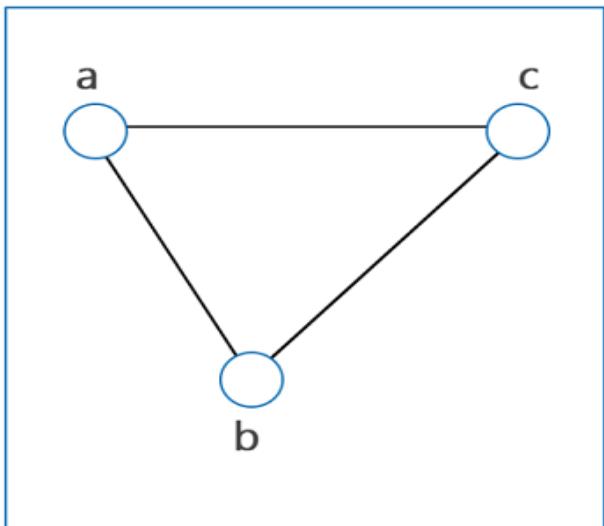
**Cycle Graph:** A graph will be known as the cycle graph if it completes a cycle. It means that for a cycle graph, the given graph must have a single cycle. With the help of symbol  $C_n$ , we can denote a cycle graph with  $n$  vertices.

Connected Graph

Disconnected Graph

**Complete Graph:** A graph is called complete graph if each pair of vertices is connected with exactly one edge. A simple graph will be called a complete graph if there are  $n$  numbers of vertices which are having exactly one edge between each pair of vertices. With the help of symbol  $K_n$ , we can indicate the complete graph of  $n$  vertices. In a complete graph, the total number of edges with  $n$  vertices is described as  $n*(n-1)/2$ .

The diagram of a complete graph is described as follows:



Propositional Logic is concerned with statements to which the truth values, “true” and “false”, can be assigned. The purpose is to analyze these statements either individually or in a composite manner.

### Prepositional Logic – Definition

A proposition is a collection of declarative statements that has either a truth value "true" or a truth value "false". A propositional consists of propositional variables and connectives. We denote the propositional variables by capital letters (P, Q, etc.).

Some examples of Propositions are given below –

"Man is Mortal", it returns truth value "TRUE"

" $10 + 18 = 12 - 5$ ", it returns truth value "FALSE"

The following is not a Proposition –

"P is less than 12". It is because unless we give a specific value of A, we cannot determine whether the statement is true or false.

### Connectives

In propositional logic generally we use five connectives which are –

Disjunction/OR ( $\vee$ )

Conjunction/AND ( $\wedge$ )

Negation/ NOT ( $\neg$ )

Implication /if-then ( $\rightarrow$ )

Bidirectional Implication / If and only if ( $\Leftrightarrow$ ).

## **OR (v)**

The OR operation of two propositions A and B (written as  $A \vee B$ ) is true if at least any of the propositional variable A or B is true.

The truth table is as follows –

A	B	$A \vee B$
True	True	True
True	False	True
False	True	True
False	False	False

## **AND ( $\wedge$ )**

The AND operation of two propositions A and B (written as  $A \wedge B$ ) is true if both the propositional variable A and B is true.

The truth table is as follows –

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False

## **Negation ( $\neg$ )**

The negation of a proposition A (written as  $\neg A$ ) is false when A is true and is true when A is false.

The truth table is as follows –

A	$\neg A$
True	False

True	False
False	True

### Implication / if-then ( $\rightarrow$ )

An implication  $A \rightarrow B$  is the proposition “if A, then B”. It is false if A is true and B is false. Rest of the cases are true.

The truth table is as follows –

A	B	$A \rightarrow B$
True	True	True
True	False	False
False	True	True
False	False	True

### If and only if ( $\Leftrightarrow$ )

$A \Leftrightarrow B$  is bi-conditional logical connective which is true when p and q are same, i.e. both are false or both are true.

The truth table is as follows –

A	B	$A \Leftrightarrow B$
True	True	True
True	False	False
False	True	False

False	False	True
-------	-------	------

### Tautologies

A Tautology is a formula which is always true for every value of its propositional variables. E.g.  
 $P \vee \neg P$ .

### Contradictions

A Contradiction is a formula which is always false for every value of its propositional variables  
E.g.  $Q \wedge \neg Q$ .

### Logical Equivalence

Two propositions p and q are logically equivalent if their truth tables are the same.

1. In a class, there are 33 boys and 21 girls. The teacher wants to select 1 boy and 1 girl to represent the class for a function. In how many ways can the teacher make this selection?

Solution Here the teacher is to perform two operations:

- (i) Selecting a boy from among the 33 boys and
- (ii) Selecting a girl from among 21 girls.

The first of these can be done in 33 ways and second can be performed in 21 ways. By the fundamental principle of counting, the required number of ways is  $33 \times 21 = 693$

### **Arrangements or Permutations**

Distinctly ordered sets are called arrangements or permutations.

The number of permutations of  $n$  objects taken  $r$  at a time is given by:

$$nPr = n! / (n - r)!$$

where  $n$  = number of objects

$r$  = number of positions

E.g:

There are 5 athletes in a race.

- a) In how many different orders can the athletes finish the race?

Solution:  $5.4.3.2.1 = 5!$  or  $5P5$

- b) In how many ways 1<sup>st</sup>, 2<sup>nd</sup> and 3<sup>rd</sup> position are possible?

Solution:  $5.4.3 = 60$  or  $5P3$

### **Combination**

$$nCr = nPr/r! = n! / (n-r)!r!$$

Question: In how many ways 2 English books, 5 Hindi books, 2 Mathematics books and 5 Physics books can be arranged on a shelf so that all books of the same subjects are together.

Solution First we take books of a particular subject as one unit. Thus there are 4 units which can be arranged in  $4! = 24$  ways.

Now in each of arrangements, English books can be arranged in  $2!$  ways, Hindi books in  $5!$  ways, Mathematics books in  $2!$  ways and Physics books in  $5!$  ways. Thus the total number of ways  $= 4! \times 2! \times 5! \times 2! \times 5! = (4.3.2.1) * (2.1) * (5.4.3.2.1) * (2.1) * (5.4.3.2.1) = 24 * 2 * 120 * 2 * 120$ .



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# Module 3: Agenda

## Module 3: Specification Based Testing – (1/2)

Topic 3.1

Specification Based Testing – Overview

Topic 3.2

Equivalence Class

Topic 3.3

Boundary Value Analysis

Topic 3.4

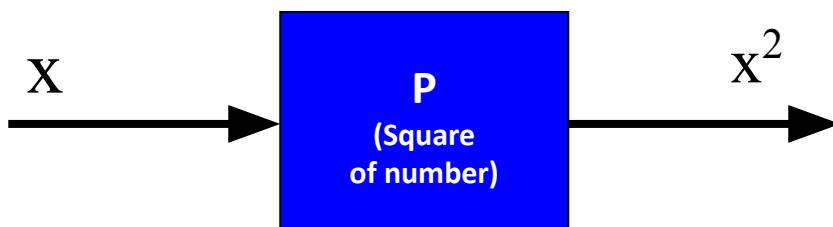
Examples & Case Study



# **Topic 3.1: Specification Based Testing – Overview**

# The Concept

- A black box test technique
- Based on specifications
- Independent of implementation
- Focus
  - Functional testing
  - Behaviour
  - Input & corresponding output



- Implementation may be,**
- A. Multiplication ( $x \times x$ )
  - B. successive addition ( $x + x \dots x$  times)

# Approaches & “View”

---

- Purpose is to uncover defects
- Demonstrate the system works (Treat this as a by-product!)
- Validate that it functions per specifications
- Works as specified – always!

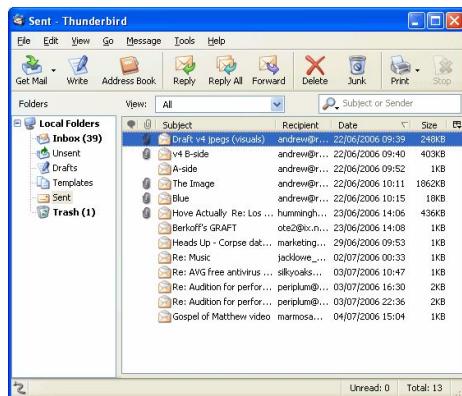
# Perspectives

---

- Customer/Client
- Alpha/Beta User
- End User/Consumer
- Development Engineer
- Architect
- Product Manager
- Maintenance Engineer
- ...

# Examples

- Automated Teller Machine
- Tea/Coffee Vending Machine
- Washing Machine
- Contacts – Mobile Phone Application
- Messaging – Mobile Phone Application
- Email – Webmail/App/Client
- ...





# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# **Topic 3.2: Equivalence Class Partitioning**

# Examples

---

## Problem 1

- Design Test Cases for a Software Program that takes in an input of up to 1000 numbers, finds the maximum and output is the max number

## Problem 2

- Design and Discuss test cases for a function returns the max of 3 numbers. The numbers must be integer, else it returns an error

# Equivalence Class

---

- What is an equivalence class?
- How is it useful to us as test designers?

# Equivalence Class

---

- EC forms a partition of a set (input domain), where partition refers to a collection of mutually disjoint subsets (subdomains) when the union is an entire set
- Two important implications
  - The fact that the entire set is represented provides a form of completeness
  - The disjointedness ensures a form of non-redundancy

# Equivalence Class

---

Reduces the potential redundancy

- The subsets are determined by an Equivalence relation, the elements have something in common
- Idea of EC is to identify (at least) one test case from each EC

***Choice of EC is a challenge!***

# EC Types

---

- **Equivalence Classes (EC) - Types**
- Weak Normal (WN)
- Strong Normal (SN)
- Weak Robust (WR)
- Strong Robust (SR)

**Types which ensure that we choose the “correct” set of test cases from the ECs we come up with**

# EC - Example

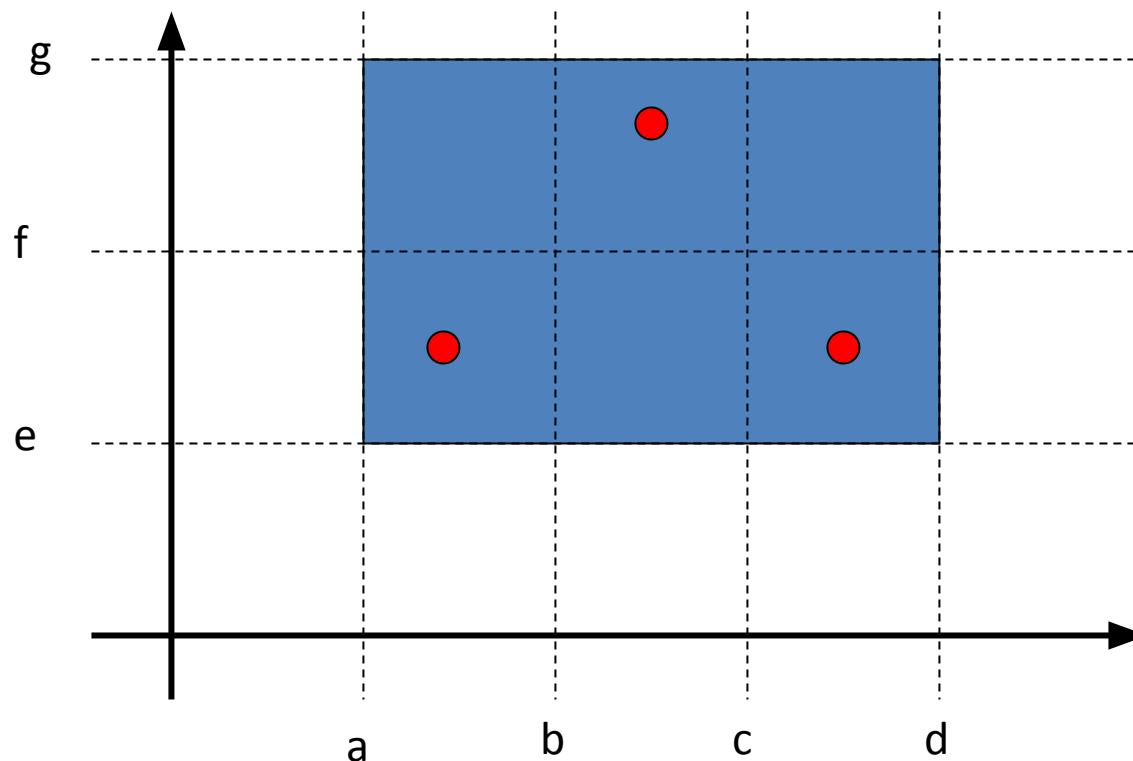
- A program takes 2 inputs  $x_1$  and  $x_2$ 
    - $a \leq x_1 \leq d$
    - $e \leq x_2 \leq g$
  - We have the intervals
    - $[a, b), [b, c), [c, d]$   $\square x_1$
    - $[e, f), [f, g]$   $\square x_2$
- 
- [  $\square$  closed interval endpoint
  - (  $\square$  open interval endpoint
  - < >  $\square$  Ordered pair
  - ( )  $\square$  Unordered pair

# EC – Weak Normal

---

- One variable from each EC
- A systematic way of deriving the EC
- Same number of weak EC test cases as classes in the partition with the largest number of subsets
- Based on a single fault assumption
- Testing valid subdomains
- Assumption
  - Input variables are independent
  - One dimensional valid subdomains
- Selects tests from one dimensional (one variable) subdomains

# EC – Weak Normal

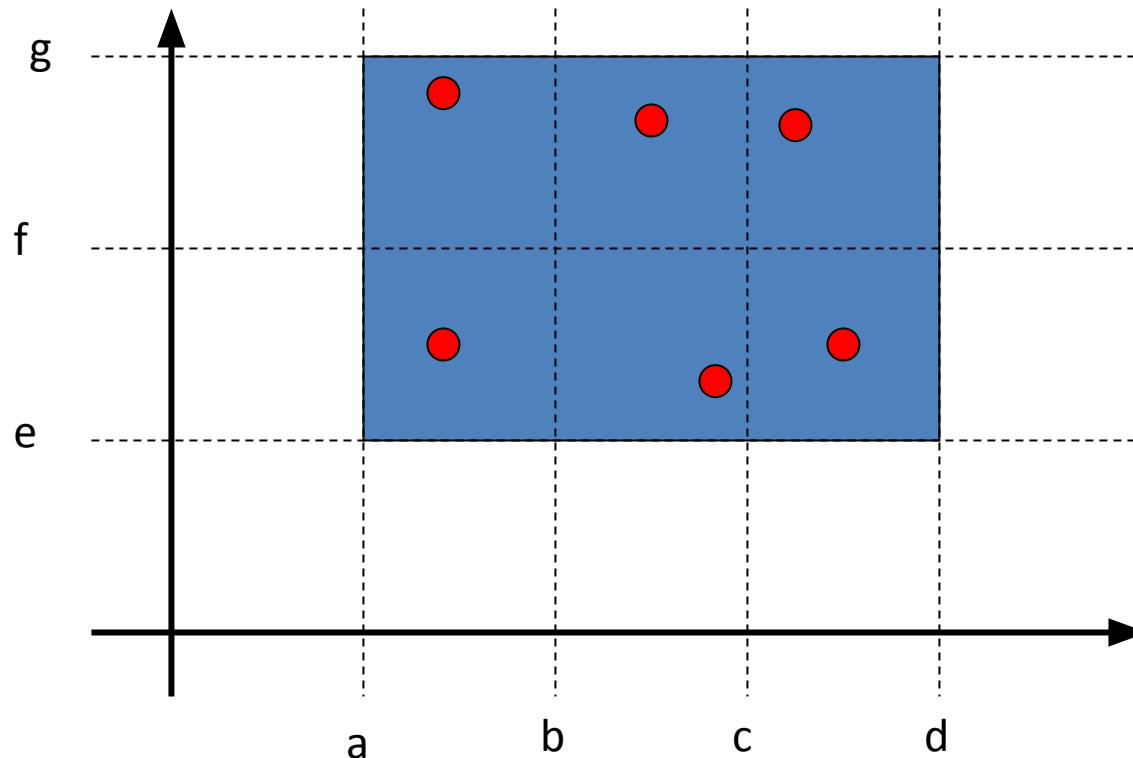


# EC – Strong Normal

---

- Based on a multiple fault assumption
- We derive test cases out of the Cartesian product of equivalence classes
- Notion of “completeness”
- Testing valid subdomains
- Assumption
  - Input variables are related
  - Multidimensional subdomains. (Example)
- Test selection: Select at least one test from each of the multidimensional sub domain

# EC – Strong Normal

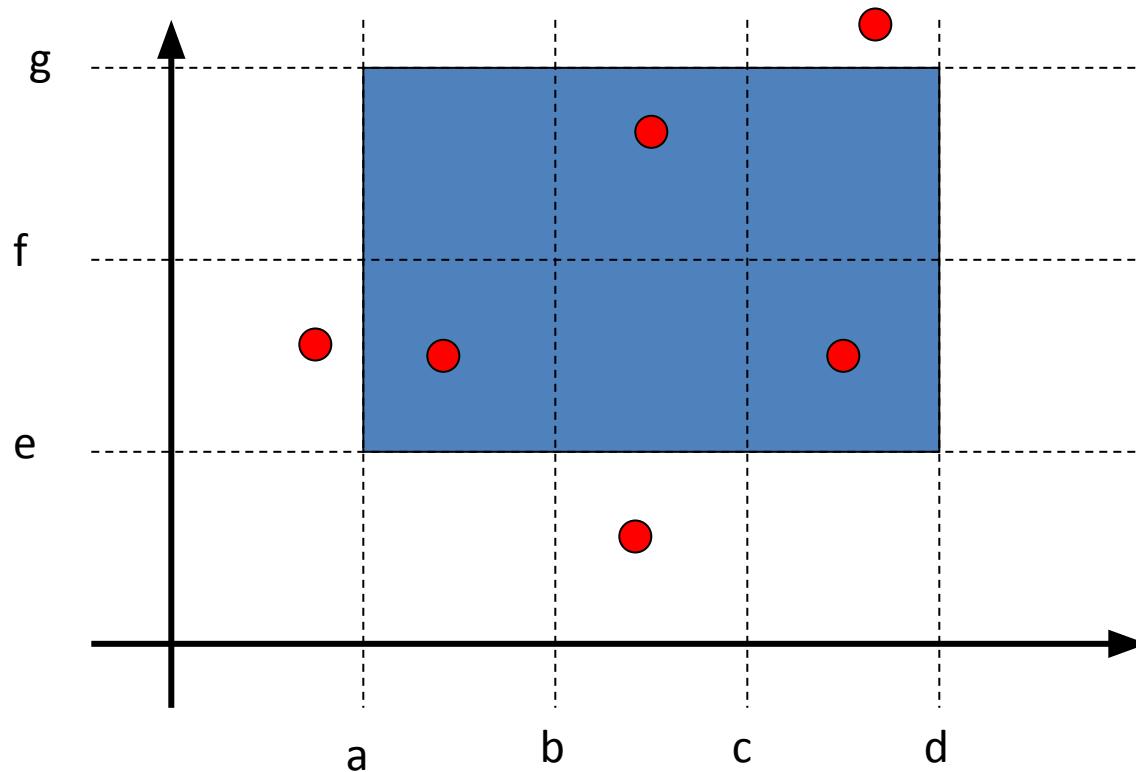


# EC – Weak Robust

---

- Weak Robust is counter-intuitive.
- Robust comes from the consideration of invalid values
- Weak refers to the single fault assumption
- A test case should have one invalid value and the remaining values should be valid
- One dimensional invalid subdomains

# EC – Weak Robust

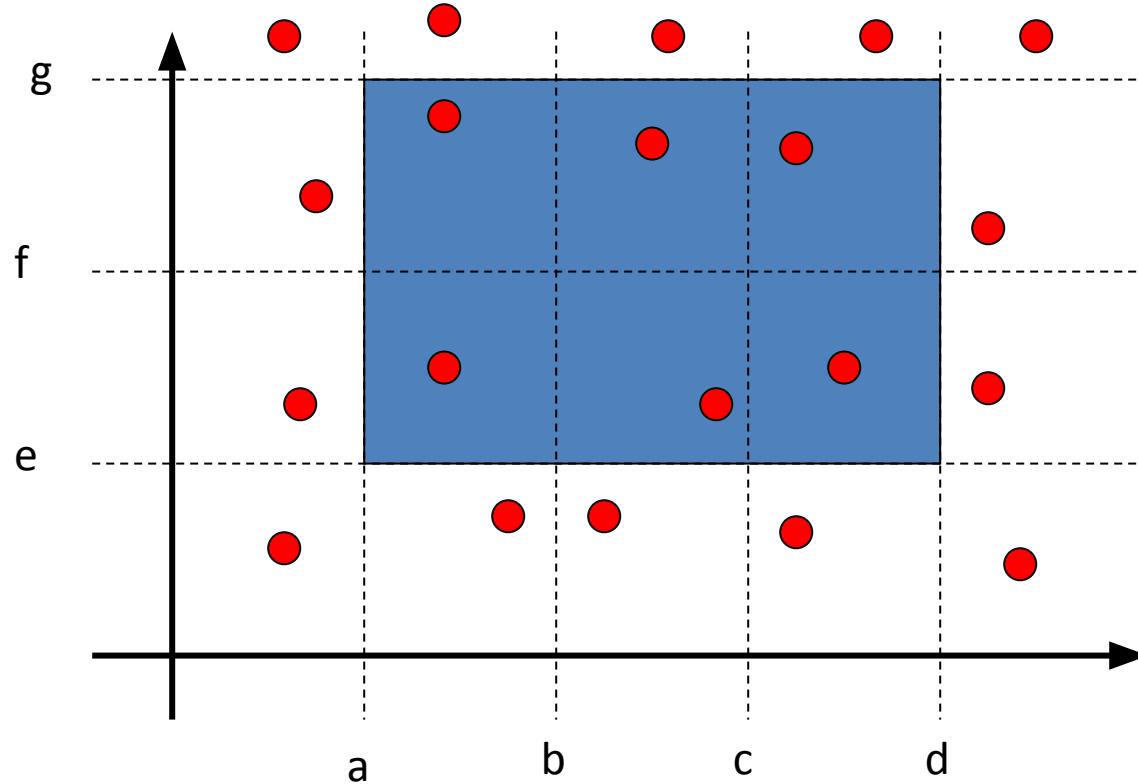


# EC – Strong Robust

---

- Robust comes from consideration of invalid values for inputs
- Strong refers to the multiple fault assumption

# EC – Strong Robust



# EC - Characteristics

---

- A group forms a EC if
  - They all test the same thing
  - If one test case catches a defect, the others probably will too
  - If one test case doesn't catch a defect, the others probably won't either
- What makes us consider them as equivalent
  - They involve the same input variable
  - They result in similar operations in the program
  - They affect the same output variable
  - None force the program to do error handling or all of them do

# Recommendations for Identifications of EC

- Equivalence class for invalid inputs
- Looks for Range in numbers
- Look for membership in a group
- Analyse responses to lists and menus
- Looks for variables that must be equal
- Create time-determined equivalence classes
- Look for equivalent output events
- Look for variable groups that must calculate to a certain value or range
- Look for equivalent operating environments

Ref: Testing Computer Software, Kaner, Falk and Nguyen, Chapter 7

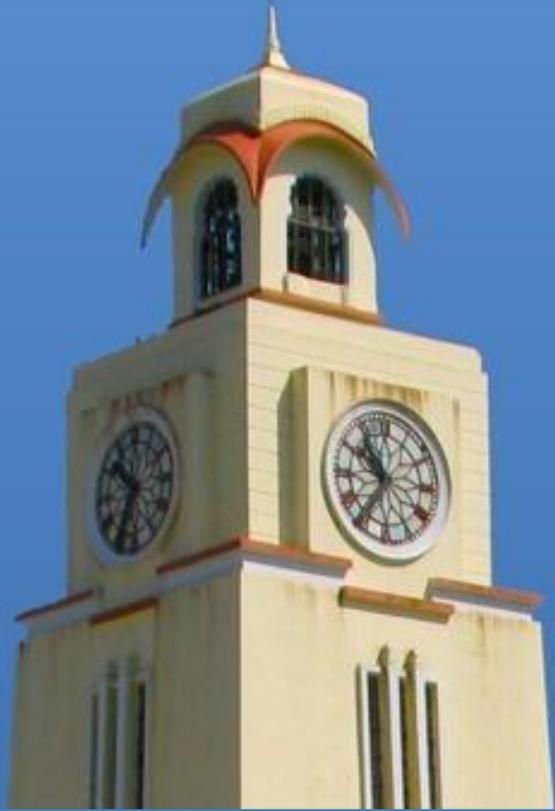


# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# **Topic 3.3: Boundary Value Analysis**

# Boundary Value Analysis

---

- Boundary Value Analysis focuses on the boundary of the input space to identify test cases
  - Rationale is, errors tend to occur near the extreme value of the input variable
- Examples
  - Loop counters off by 1
  - Inputs at the boundary of ranges.  $10 < x < 100$

# Boundary Value Analysis

---

- Idea of BVA is to use input variable values at
  - Their minimum
  - Just above the minimum
  - A nominal value
  - Just below their maximum
  - At their maximum

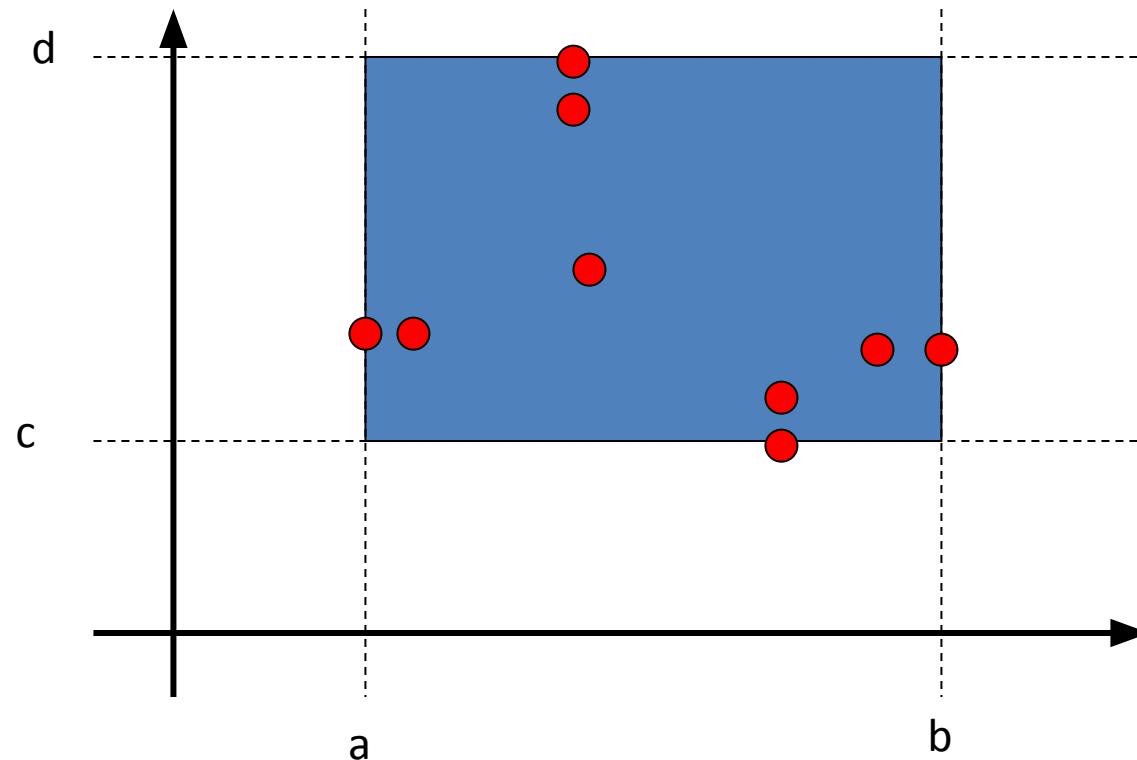
# BVA – Explore the types

## Example

- A program takes 2 inputs  $x_1$  and  $x_2$ 
  - $a \leq x_1 \leq b$
  - $c \leq x_2 \leq d$
- We have the intervals
  - $[a, b] \quad \square \quad x_1$
  - $[c, d] \quad \square \quad x_2$

# BVA

BVA test cases for a function of two variables – single fault assumption



# Generalising BVA

---

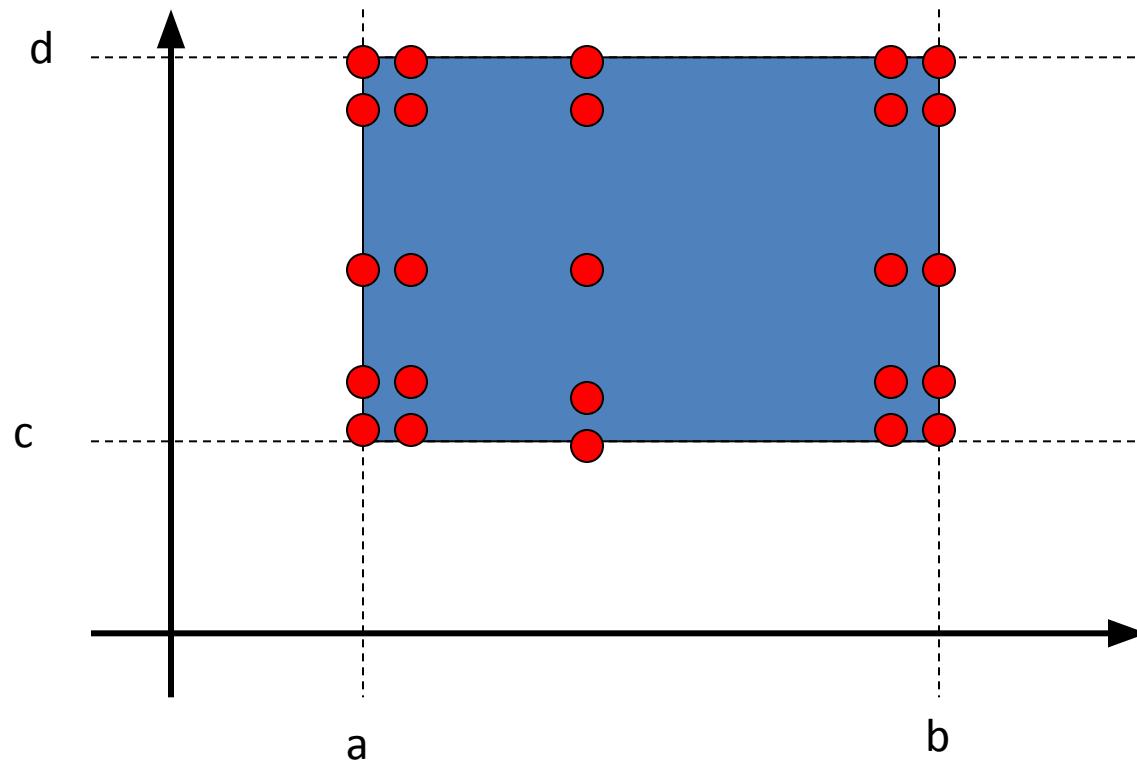
- **Two ways**
  - Number of input variables
  - Ranges
- **Variable generalization**
  - Hold one at the nominal value and let the other variable assume min, min+, nom, max- and max. i.e.  $4n+1$  test cases

# BVA Limitations

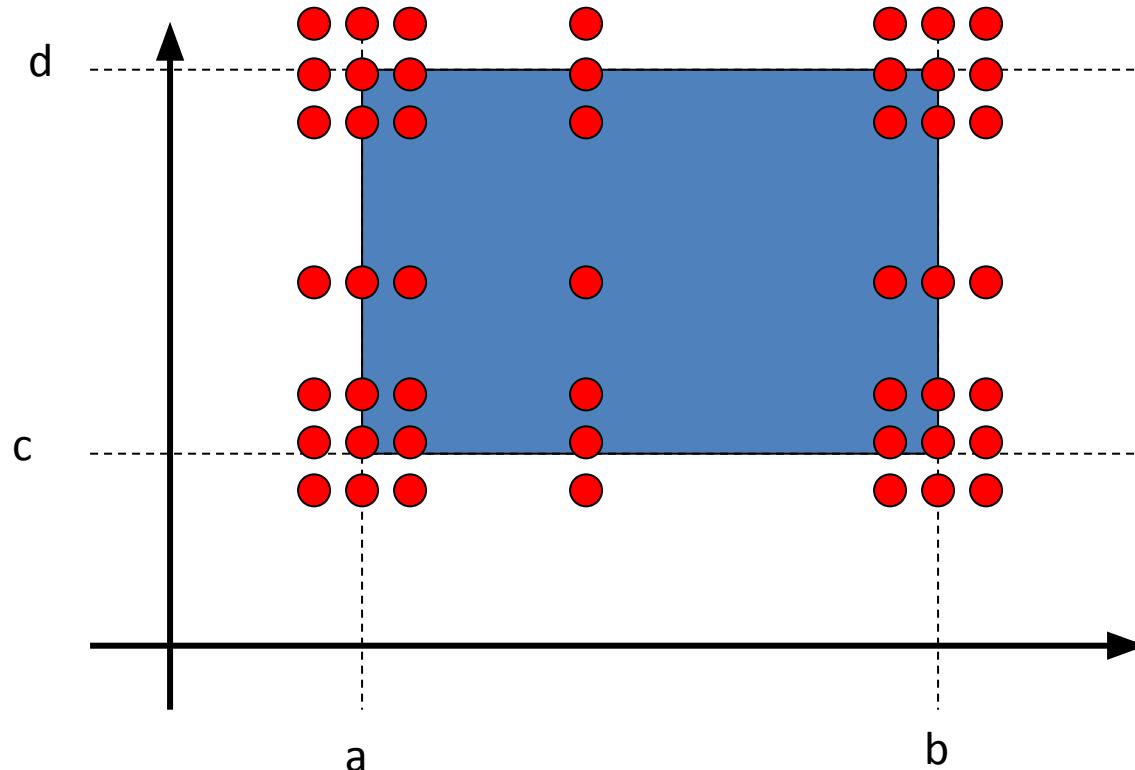
---

- BVA works well when the program to be tested is a function of several independent variables that represent bounded physical quantities
- No consideration to the functionality or semantic meaning of variables

# BVA – Worst Case Analysis



# BVA – Robust Worst Case



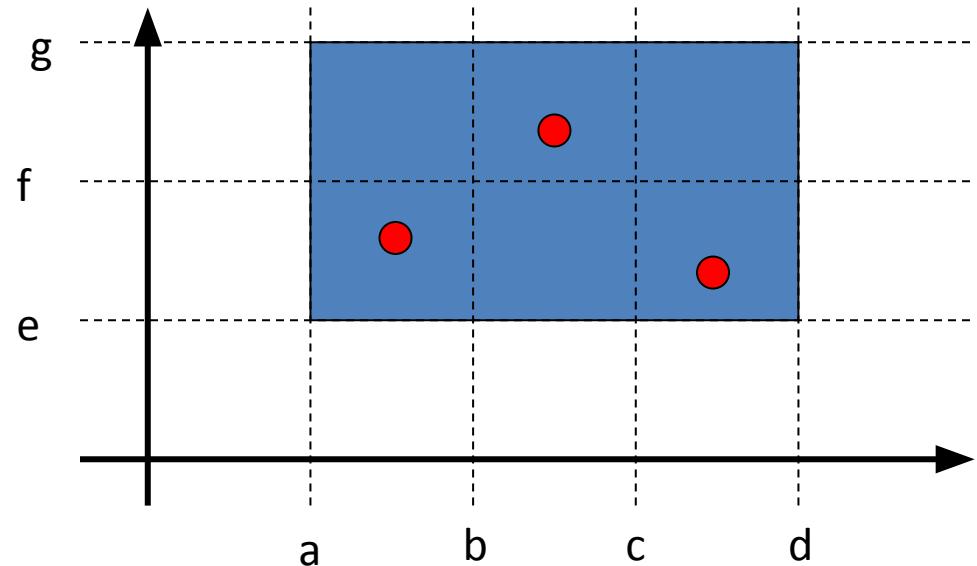
# BVA – Special Value Testing

---

- Practiced form of functional testing
- Most intuitive and least uniform
- Use of Test Engineer's domain knowledge
  - *Gut feel*
  - Ad hoc testing

# Edge Testing

- ISTQB Advanced Level Syllabus (ISTQB, 2012) describes a hybrid of BVA and EC
- Edge Testing
- Faults near the boundaries of the classes
- Normal & Robust for Edge Testing





# Software Testing Methodologies



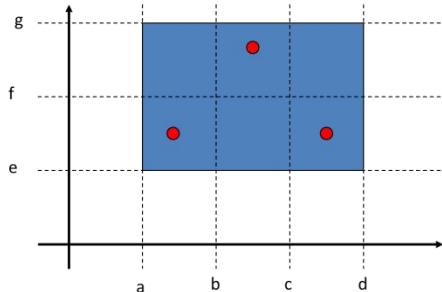
**BITS** Pilani

Prashant Joshi

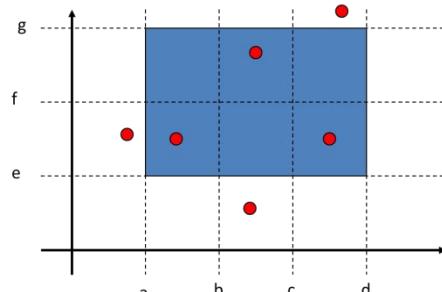


## **Topic 3.4: Examples & Case Study**

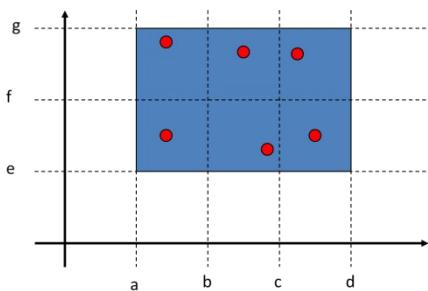
# EC & BVA



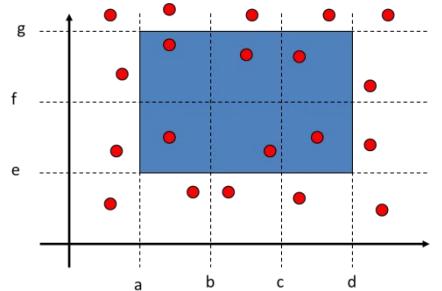
Weak Normal



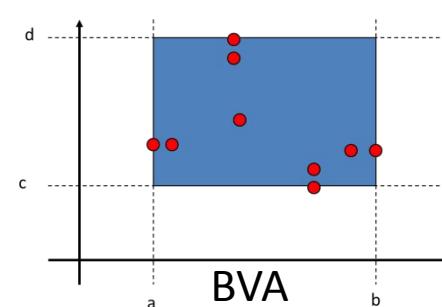
Weak Robust



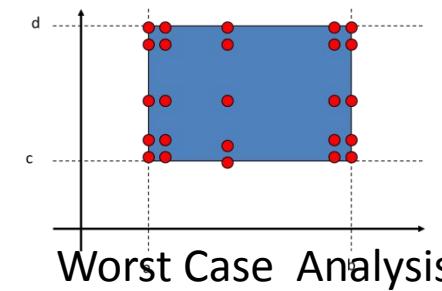
Strong Normal



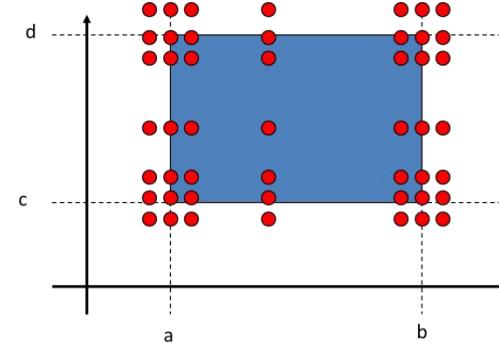
Strong Robust



BVA



Worst Case Analysis



a b

# Examples

---

## Problem 1

- Design Test Cases for a Software Program that takes in an input of up to 1000 numbers, finds the maximum and output is the max number

## Problem 2

- Design and Discuss test cases for a function returns the max of 3 numbers. The numbers must be integer, else it returns an error

# Example – Max of 3 numbers

#	Input Condition	Valid Sub-domain		Invalid Sub-domain	
1	Number a	$0 \leq a < 100$	(1)	$a < 0$ $a > 100$	(2) (3)
2	Number b	$0 \leq b < 100$	(4)	$b < 0$ $b > 100$	(5) (6)
3	Number c	$0 \leq c < 100$	(7)	$c < 0$ $c > 100$	(8) (9)

- Choose the subdomains to satisfy for a specific type of EC or BVA
- Choose an input to form the test case

# Process for Test Case Creation



- Create a table with valid and in-valid subdomains
- Number the rows
- Based on the focus (WN, SN, WR, SR of EC) pick the combination of the rows (valid and in-valid subdomains)
- Choose a value and outcome which will form a test case

**Repeat any or all steps to arrive at coverage and completeness as required for the problem at hand**

# Example – Max of 3 numbers

#	Input Condition	Valid Sub-domain		Invalid Sub-domain	
1	Number a	$0 \leq a < 100$	(1)	$a < 0$ $a > 100$	(2) (3)
2	Number b	$0 \leq b < 100$	(4)	$b < 0$ $b > 100$	(5) (6)
3	Number c	$0 \leq c < 100$	(7)	$c < 0$ $c > 100$	(8) (9)
4	Max	a b c	(10) (11) (12)		?
5	Two equal	a & b b & c c & a	(13) (14) (15)		?
6	All three equal	a, b &c	(16)		

# Examples - discuss

---

- Discuss various test cases and design approaches
- What types of faults are anticipated?
- Are the requirements sufficient?
- Any assumptions made? How were the assumptions made?
- It is recommended that code should be written for both to understand the problem better.

# The Triangle Example

---

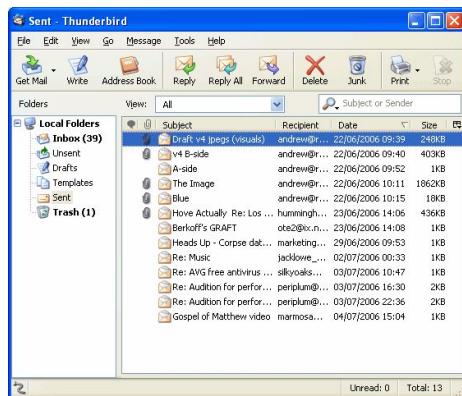
## Problem Statement

- A program takes an input of  $a$ ,  $b$  and  $c$ , which are three sides of a triangle. Based on the length of the three sides the following output is generated,
  1. Not a Triangle
  2. Equilateral triangle
  3. Isosceles Triangle
  4. Scalene Triangle

Variants (a) Type of triangle (b) Which side is the hypotenuse?  
(c) Area of the triangle

# Examples

- Automated Teller Machine
- Tea/Coffee Vending Machine
- Washing Machine
- Contacts – Mobile Phone Application
- Messaging – Mobile Phone Application
- Email – Webmail/App/Client
- ...





# Software Testing Methodologies



**BITS** Pilani

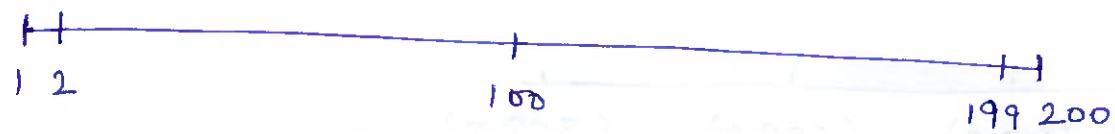
Prashant Joshi

BVAProblem 1:

for a program "Cube" which takes y as I/P and prints cube of y as O/P. The range of y is from 1 to 200.

Solution:

Its BVA would be achieved as:



<u>Test Case</u>	<u>I/P</u>	<u>Expected O/P</u>
1.	1	1
2.	2	8
3.	100	1,00,000
4.	199	78,80,599
5.	200	8 0,0 0,0 00

Problem 2:

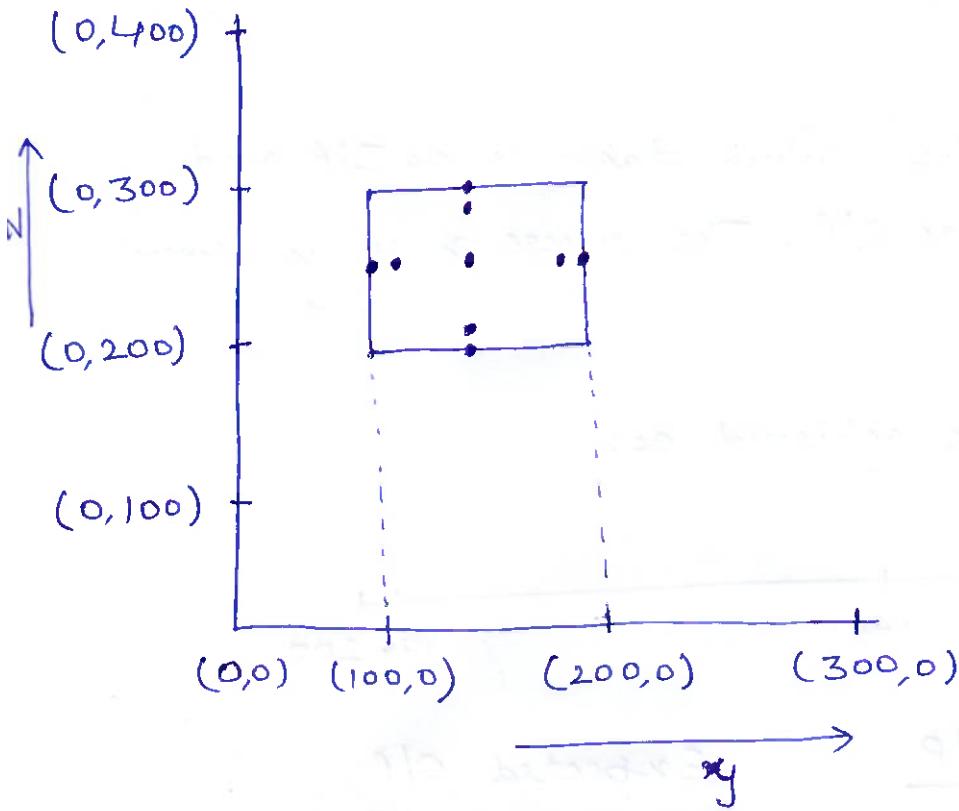
for a program "Multiplication" with two inputs y and z with range of both as:

$$100 \leq y \leq 200$$

$$200 \leq z \leq 300$$

Solution:

(2)

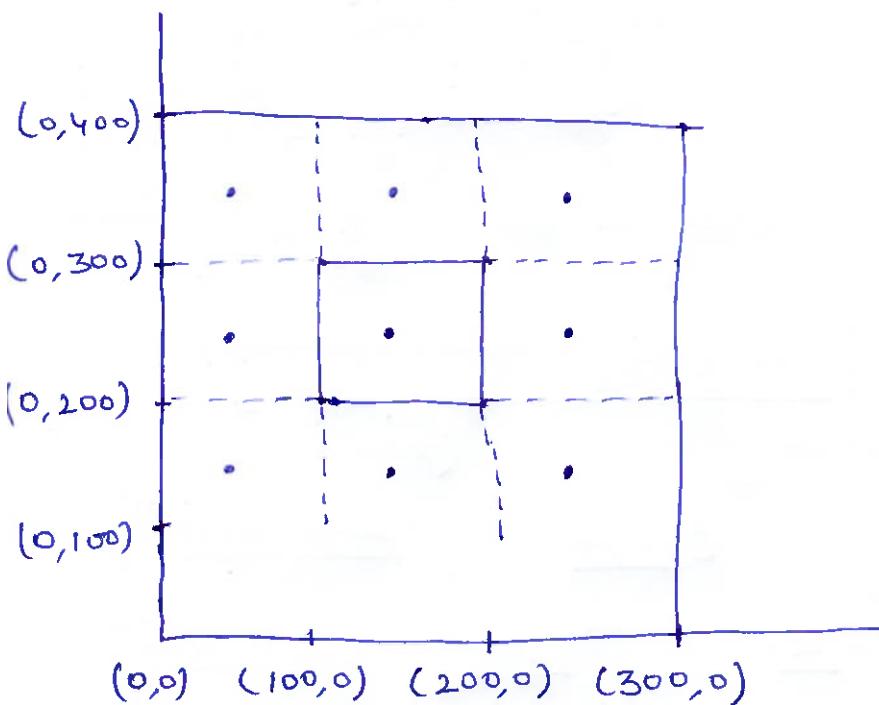


<u>Test Case</u>	<u>y</u>	<u>z</u>	<u>Expected O/P</u>
1.	150	200	30,000
2.	150	201	30,150
3.	150	250	37,500
4.	150	299	44,850
5.	150	300	45,000
6.	100	250	25,000
7.	101	250	25,250
8.	199	250	49,750
9.	200	250	50,000

## Equivalence Class Testing

(3)

EC for same problem (Problem 2 discussed in BVA)  
Can be drawn as:



<u>Test Case</u>	<u>Y</u>	<u>Z</u>	<u>Expected Output</u>
1.	103	200	20,600
2.	60	150	Invalid Input
3.	160	140	Invalid Input
4.	265	160	Invalid Input
5.	70	210	Invalid Input
6.	75	310	Invalid Input
7.	110	320	Invalid Input
8.	220	250	Invalid Input
9.	275	325	Invalid Input



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# Module 4: Agenda

## Module 4: Specification Based Testing – (2/2)

Topic 4.1

Domain Testing

Topic 4.2

Combinatorial

Topic 4.3

Decision Table Based Testing

Topic 4.4

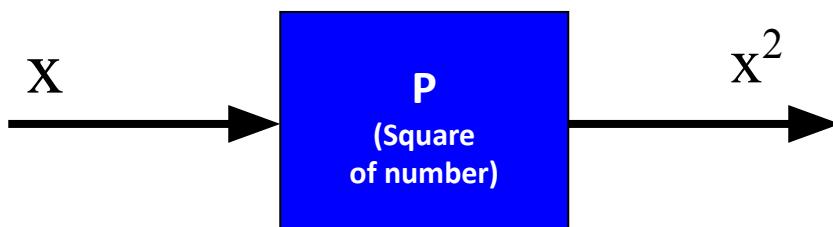
Examples & Case Study



# Topic 4.1: Domain Testing

# The Concept

- A black box test technique
- Based on specifications
- Independent of implementation
- Focus
  - Functional testing
  - Behaviour
  - Input & corresponding output



- Implementation may be,**
- A. Multiplication ( $x \times x$ )
  - B. successive addition ( $x + x \dots x$  times)

# Approaches & “View”

---

- Purpose is to uncover defects
- Demonstrate the system works (Treat this as a by-product!)
- Validate that it functions per specifications
- Works as specified – always!

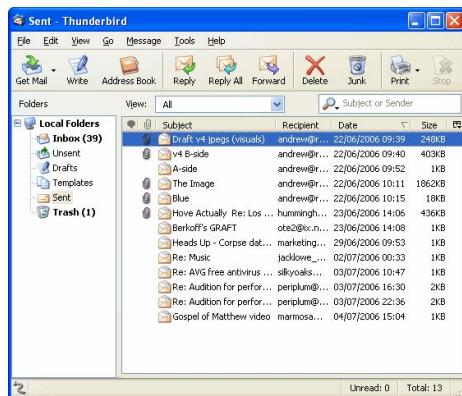
# Perspectives

---

- Customer/Client
- Alpha/Beta User
- End User/Consumer
- Development Engineer
- Architect
- Product Manager
- Maintenance Engineer
- ...

# Examples

- Automated Teller Machine
- Tea/Coffee Vending Machine
- Washing Machine
- Contacts – Mobile Phone Application
- Messaging – Mobile Phone Application
- Email – Webmail/App/Client
- ...



# What is it?

---

- It is a systematisation of the Equivalence Partitioning and Boundary Value Analysis
- Introduces concept of Test Specification which allows the test engineer to take a closer look at the specifications
- Allows division of tasks for larger systems

# Category Partitioning Method

---

- Systematic approach to generation of test cases from requirements
- Mix of manual and automated steps
- Ref: T2 Chapter 3 Section 3.5

# CP Method Steps

---

1. Analyse Specification
  2. Identify Categories
  3. Partition Categories
  4. Identify Constraints
  5. (Re) write test specification
  6. Process specification
  7. Evaluate generator output
  8. Generate test scripts
-

# 1: Analyse Specification

---

- Identify the functional unit that can be tested separately
- For larger systems, break down into subsystems, that can be tested independently

## 2: Identify Categories

---

- For each testable unit, analyse the specification and isolate inputs
- Identify objects in the environment
- Determine characteristics (category) of each parameter and environment object
  - Explicit characteristics
  - Implicit characteristics

# 3: Partition Categories

---

- Identify cases against which the functional unit must be tested (cases = choices)
- Partition categories into at least two subsets – correct values and incorrect values
  - Valid subdomain
  - Invalid subdomain

# 4: Identify Constraints

---

- Test for functional unit consists of a combination of choices for (a) parameter and (b) environment object
- Identify possible and not-possible combinations
- Thus, specify the constraints

# 5: (Re) write test specification

---

- Based on previous steps a test specification can now be written
- Write the complete test specification
- Make use of a TSL which can help automate the process

# 6: Process Specification

---

- TSL or test specification is used to generate test frames
- Test frames are not test cases

# 7: Evaluate Generator Output

---



- Examine test frames for redundancy or missing cases
- Iterative process and steps

# 8: Generate Test Scripts

---

- Generate test cases from test frames
- Generate test scripts from test cases;  
typically a group of test cases



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# Topic 4.2: Combinatorial

# Need of Combinatorial

---

- Software to be test in various environments
- Various combination factors
- Need for combinations of inputs
- Application
  - High Reliability needs
    - » Work on various configurations (Windows, Mac, Linux)
  - Interoperability
    - » Various Clients & Servers (MMS)

# Modelling The Input

---

- Program P
  - Input variables: X and Y
  - X can take one value from {a, b, c}
  - Y can take one value from {d, e, f}
  - Leads to:
  - 9 factor combinations ( $3^2$ )
- 
- For large number of variables and values of each variable these combinations can be very large
  - If we imagine one test case per combination; we will have a huge number of test cases

# Model the input

---

- Fault Model (Interaction Faults)
  - Two or more variables play a role
  - One or more specific values play a role
- Unique combinations
  - Latin Squares
  - Pairwise Testing
  - Orthogonal Array

**In this module we focus on the use of combinatorial alone; Focus on the reduction and specific techniques for optimization will be discussed in later Module**



# Software Testing Methodologies



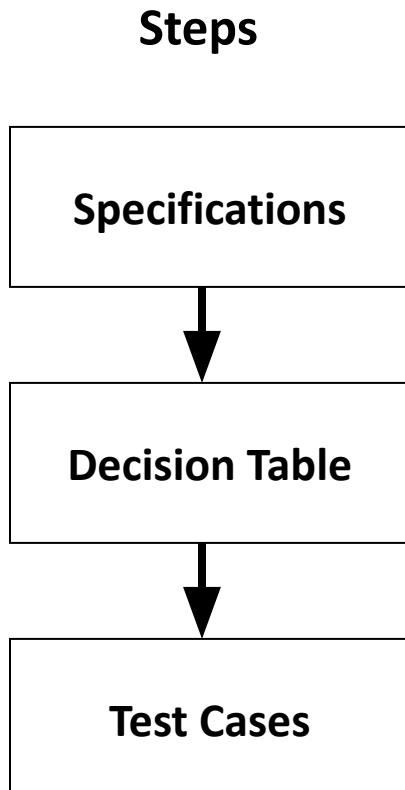
**BITS** Pilani

Prashant Joshi



# **Topic 4.3: Decision Table Based Testing**

# Decision Tables



- One of the most systematic approaches of designing Test Cases
- Decision Tables are rigorous – enforce logical rigor
- Related Method – Cause Effect Graphing
- Ideal for situations with number of combinations of actions are taken under varying sets of condition
- Structure of describing rules

# DT Technique

Stub	Rule 1	Rule 2	Rule 3, 4	Rule 5	Rule 6	Rule 7, 8
C1	T	T	T	F	F	F
C2	T	T	F	T	T	F
C3	T	F	---	T	F	---
A1	X	X		X		
A2	X				X	
A3		X		X		
A4			X			X

T: True

F: False

---: Don't Care

<blank>: Not applicable

X: Action takes place

## Notion of completeness

- N conditions  $\square 2^N$  rules
- Actions can be defined by user

# DT - Example

---

Component Specification: Input of 2 characters such that

1. The 1<sup>st</sup> character must be A or B.
2. The 2<sup>nd</sup> character must be a digit.
3. If the 1<sup>st</sup> character is A or B and the 2<sup>nd</sup> character is a digit the file is updated.
4. If the 1<sup>st</sup> character is incorrect, message X12 is displayed.
5. If the 2<sup>nd</sup> character is not a digit, message X13 is displayed.

**Develop test cases using Decision table technique**

# DT Example Solution

Stub	R1	R2	R3	R4	R5	R6	R7	R8
C1: 1st Char is A	T	T	T	T	F	F	F	F
C2: 1st Char is B	T	T	F	F	T	T	F	F
C3: 2nd char is Digit	T	F	T	F	T	F	T	F
A1: File is updated			X		X			
A2: Message X:12							X	X
A3: Message X:13					X		X	
A4: Impossible	X	X						

## Each Rule one Test Case (minimum)

Test Case #	Input	Rule
Test Case 1	A5	R3
Test Case 2	AC	R4
Test Case 3	B5	R5
Test Case 4	BC	R6
Test Case 5	C5	R7
Test Case 6	CD	R8

Note that it may not be possible to execute or even design test cases for the impossible action rule. But to begin with never ignore these conditions. Design them and then evaluate if they can be included in the Test Suite.



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



## **Topic 4.4: Examples & Case Study**

# Test Techniques

- Equivalence Class
- Boundary Value Analysis
- Domain Partitioning
- Combinatorial
- Decision Table

# Examples – Solve them

---

## TV Remote Control Software

- Software is designed for the remote control which has typical controls and options. Using DT design and develop test cases

## A web based shopping cart checkout logic

- By number of items
- By weight of items
- By value of items

# Examples – Solve them

---

An insurance renewal premium

- By Age
- By Number of claims
- By Value of claims

Folder and File Name generation for a  
Digital Camera

- Use of a Serial number
- Use of Date
- Use of Date+time

# Browsers on Operating Systems

---



- Browsers (Firefox, Chrome, Safari and MyBrowser) are to be tested on various operating systems (Windows, Linux) and on various platforms like PC, Mobile. On Mobile phones the operating systems to be considered are (iOS, WP8, Android)
- Come up with a strategy and test cases

# School Attendance System

- An attendance sub-system is developed which is part of a School Management System. At the beginning of the year the Class teacher creates the roster for the class by pulling in the details from Master Data Base. The fields that teacher uses are Full Name and Class/Division. Over that she builds a local database with entry of Nickname, Months of attendance, Attendance, Reasons for absence in case of absence (Sick, Informed Leave, School program, Competitions). Following reports are generated at the end of the week:
  - Classes attendance report
  - Student attendance report
  - Absenteeism report
  - Student absenteeism report
- Design a set of test cases to test such a sub-system



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

**Test Script:**

Set of instructions that would be performed on system under test to test the functions of a software i.e whether the software/system works as per the expectation(s).

**Test Frame:**

A test frame is an include file (.inc) that serves as a central repository of information about the system being tested. Moreover, it contains all the data structures related to the test cases and test scripts.

**Combinatorial:**

Combinatorial is needed to develop test cases in the situations where combination of different scenarios is to be tested e.g to test an application (such as MS Teams) runs on all platforms and on all the browsers or not. Further, numerous QoS attributes need to be ensured that they have been met. The QoS which need to be focused here, but are not limited to, are: reliability, interoperability and security.

**Pair wise Testing:**

Testing all the pairs using combinatorial method, generally used to test all the possible discrete combinations of parameters involved e.g testing password creation module with constraints that it must contain numeric values and alphabetical values (upper case & lower case).

**Latin Square:**

It is an  $n \times n$  array filled with  $n$  different symbols, each occurring exactly once in each row and once in each column.

$C_1$ : $x, y, z$ are sides of $\Delta$	N	Y						
$C_2$ : $x = y$	-	Y	Y					
$C_3$ : $x = z$	-	Y	Y	N	N	Y	Y	N
$C_4$ : $y = z$	-	Y	N	Y	N	Y	N	Y
$a_1$ : Not a $\Delta$	X							
$a_2$ : Scalene								
$a_3$ : Isosceles					X		X	X
$a_4$ : Equilateral	X							
$a_5$ : Impossible		X	X		X			



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# Module 5: Agenda

## Module 5: Code Based Testing (1/2)

Topic 5.1

Code Based Testing Overview

Topic 5.2

Path Testing

Topic 5.3

Examples

Topic 5.4

Case Study



# **Topic 5.1: Code Based Testing Overview**

# Code Based Testing

---

- Input to test design is source code or a program structure
- Salient Features
  - More Rigorous than specification testing WRT code
  - Lower Level than specification.
  - Validation WRT specification may not happen as input is code

# 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 or condition statements
    - IF-THEN-ELSE
    - WHILE-DO/DO-WHILE
    - SWITCH
- A variable declaration is not a statement

# Statement Testing

## Example

```

int F(int x)
1   y=0;
2,3  If  (x<1) { y=1 };
      else {
4,5    if(x<2) { y=2 };
          else
6,7    if  (x>7) { y=7 };
      }
8   return y;
}

```

**Test #1: x=0**

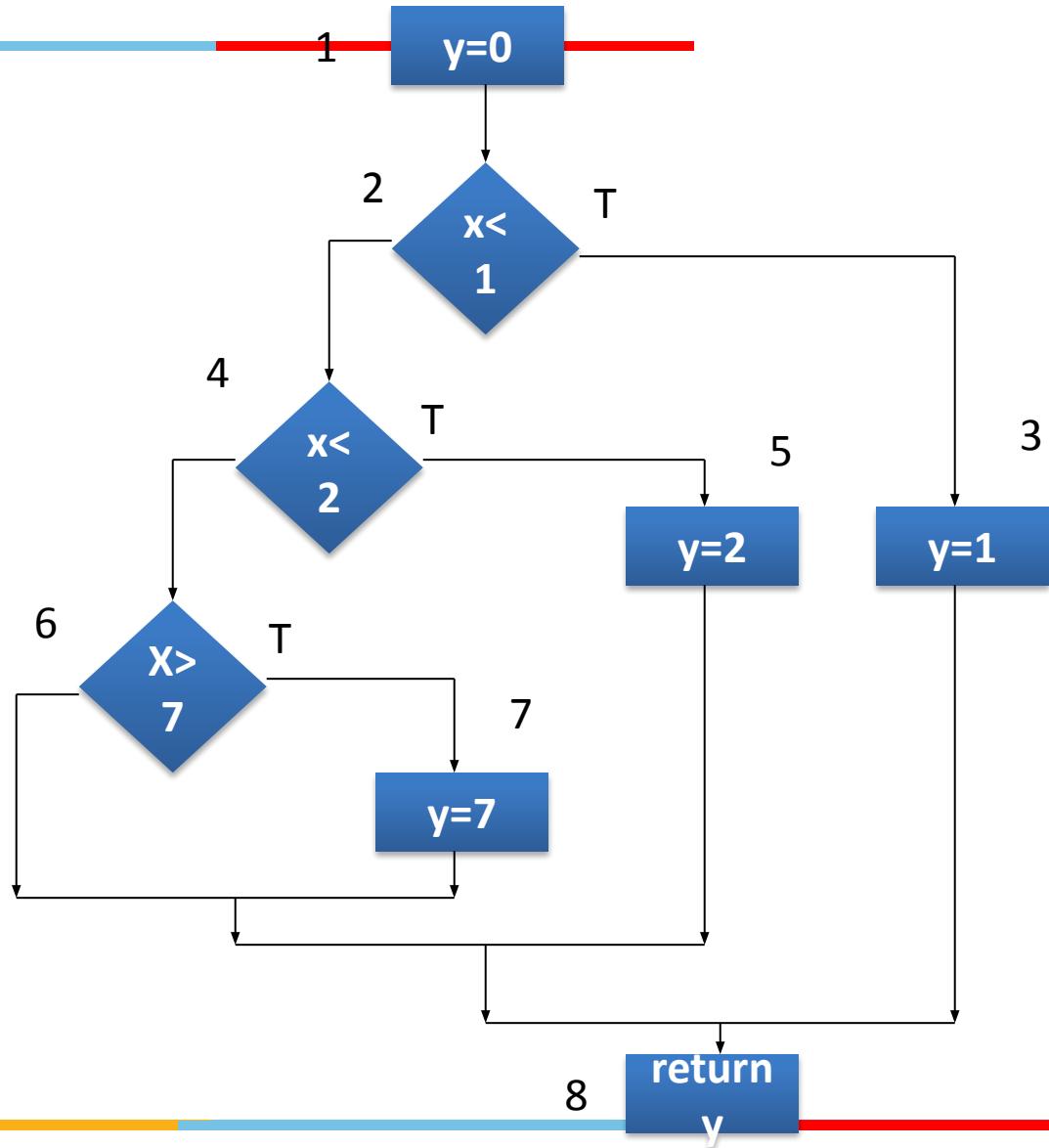
**1, 3, 8**

**Test #2:x=1**

**1, 2, 5, 8**

**Test #3:x=10**

**1, 2, 4, 7, 8**



# 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
  - Review the earlier example and check what branches we cover with the 3 test cases
    - Check with Test #4:x=5

# Branch Testing

IF Statement

  IF (condition) THEN, ELSE

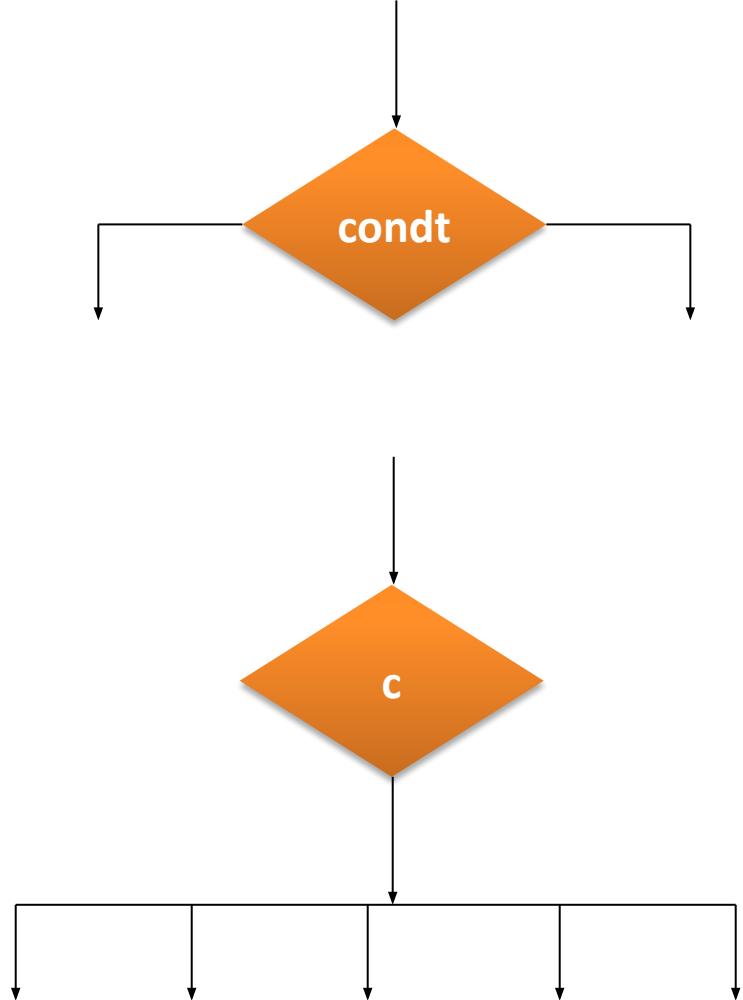
SWITCH

  SWITCH-CASE

Salient Features

  More demanding

  When branch testing is satisfied  
  the statement testing is also  
  satisfied



# Multiple Condition Testing

- This is testing of condition with complex predicates (OR, NOT and AND)
- IF C1 THEN, ELSE  Branch testing ~ multiple condition
- 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 second condition, it is a complex predicate made up C1 AND C2 AND C3.
- To test this “Test all combinations of simple predicates”

# Multiple Condition

---

## 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 and Q2  
Each P1 and P2, and Q1 and Q2 for 4 conditions in pairs

---

# Multiple Condition

## 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
```

P1	P2
x>0	y<1
T	T
T	F
F	T
F	F

Q1	Q2
x>10	z>0
T	T
T	F
F	T
F	F

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

	x	y
Test #1	15	0
Test #2	15	5
Test #3	-1	0
Test #4	-1	5

Ensure that the case worked out is for values of x and y to evaluate Q1 and Q2 and not a single statement under consideration alone

	x	y
Test #1	15	0
Test #2	15	5
Test #3	5	0
Test #4	-1	0

# Multiple Condition

---

## Salient Features

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



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



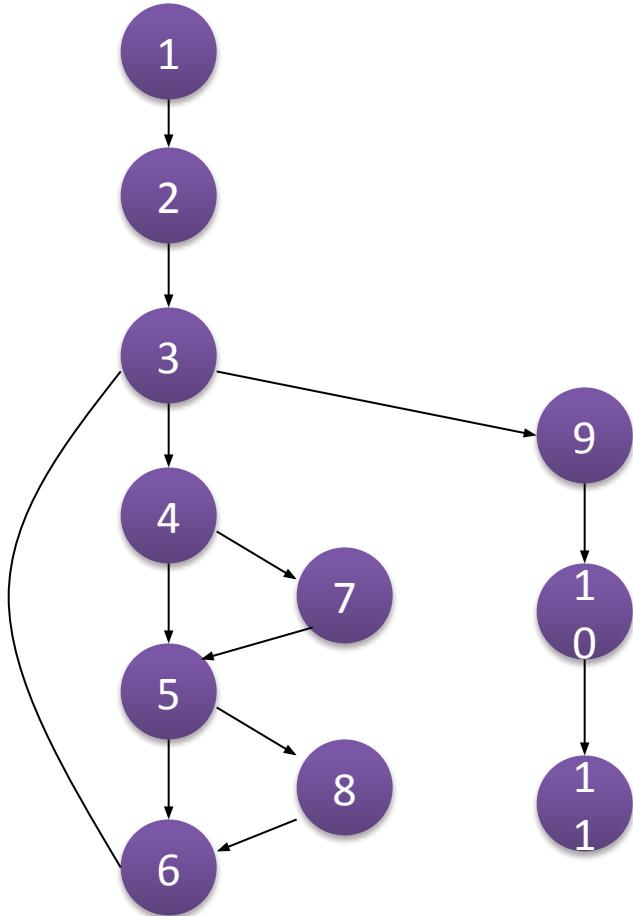
## Topic 5.2: Path Testing

# Control Flow Graph

---

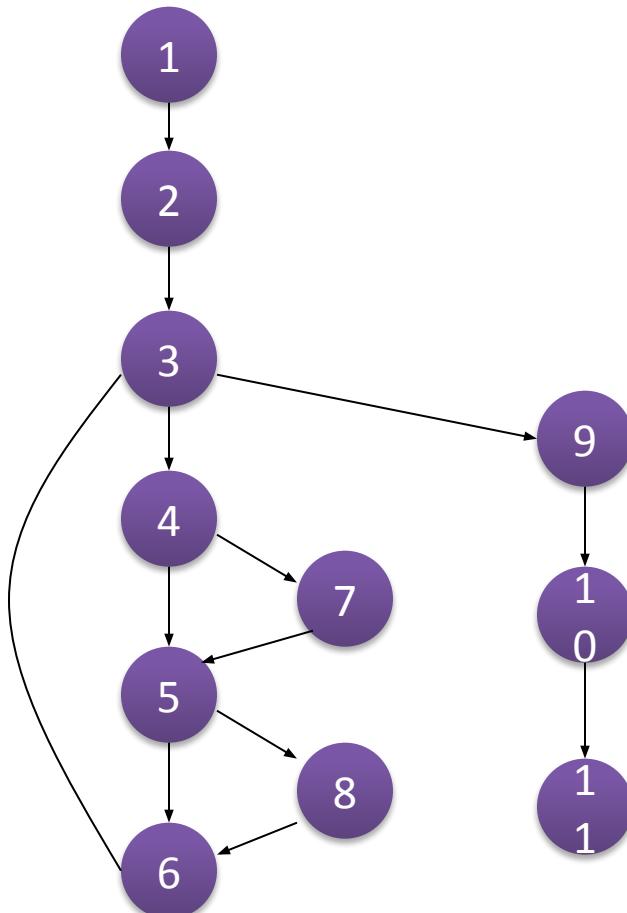
- A control Flow Graph consists of Nodes and Edges. Edges are between nodes and are directed
- A Node: A statement (i.e. executable atomic entity in a program)
  - An assignment statement
  - An input/output statement
  - Predicate of a condition
- An Edge: An edge represents a flow of control between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

# Control Flow Graph



- A path in a control flow graph of a program is a sequence of nodes (statements) in a control flow graph
- A path represents a possible execution of the program

# Loop Testing



## Simple Loop

- Test #1: Skip the loop
- Test #2: Iterate the loop once
- Test #3: Iterate the loop several times (normal Case)
- Test #4: Iterate max number of times
- Test #5: Iterate the loop (max-1) number of times

```

mathstable (x, y)
int i, j;
For (i=x; i<=x; i++) {
    print(j);
    j=j+j;
}
  
```

Work out the above example as per the loop testing concept

# Path Testing

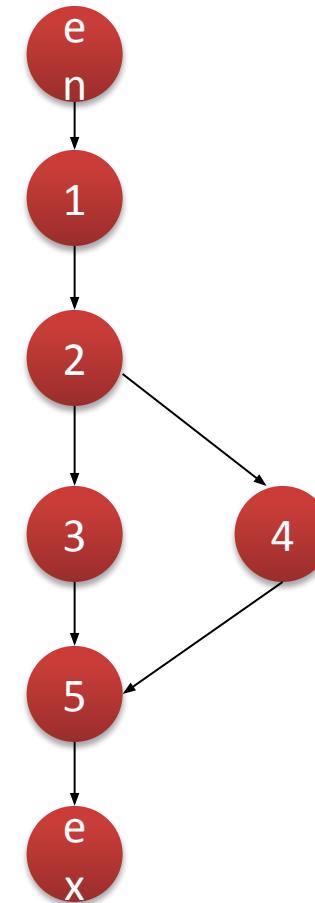
## Basic Concept

- To design a test suite (a set of test cases) for which every possible path is executed at least once

```

1      input (x)
2, 3 if (x<10) y=0;
4      else y=1;
5  output (y)
  
```

- Path#1: en, 1, 2, 3, 5, ex
- Path#2: en, 1, 2, 4, 5, ex
- Test #1: 5
- Test #2: 15
- All paths may not be executable



# Path Testing - Example

```
input (x)
if (x<10) then y=0
else y=1
if (x<30) then z=1
else z=2
output (y, z)
```

Branch testing : 2 branches

Test

Path#1: 1, 2, 3, 5, 6, 8 T1 : x=5

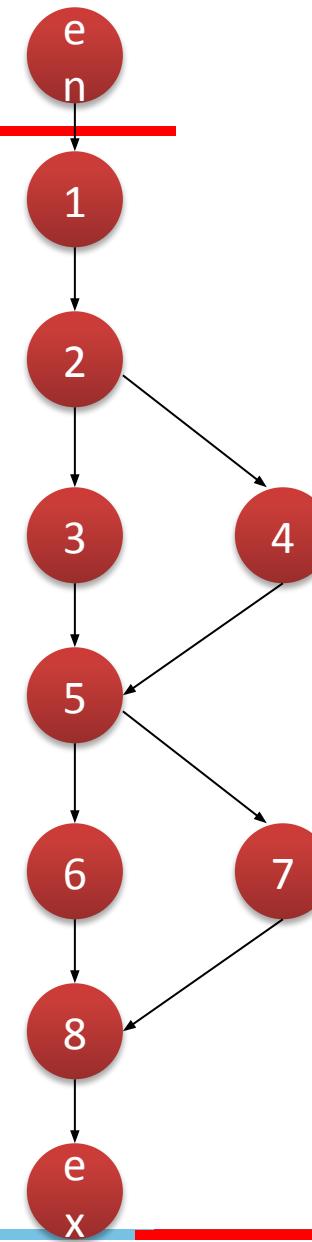
Path#2: 1, 2, 3, 5, 7, 8 T2 : x=?

Path#3: 1, 2, 4, 5, 6, 8 T3 : x=15

Path#4: 1, 2, 4, 5, 7, 8 T4 : x=35

(x<10) and x>30 is not possible

Therefore, Path#2 is not possible



# Path Testing

# of branches = 1

Paths

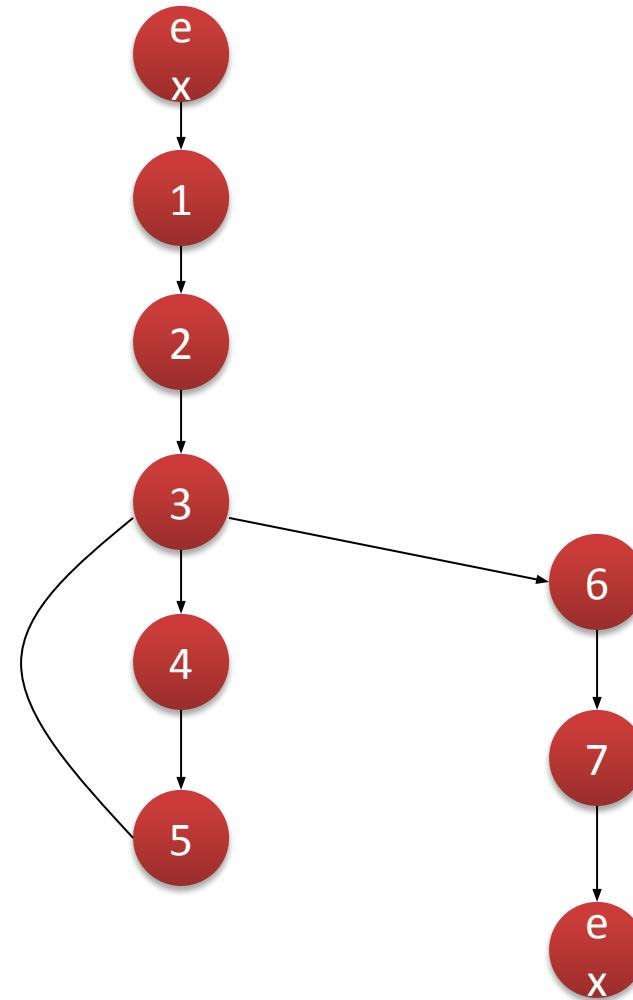
P1: 1 2 3 6 7

P2: 1 2 3 4 5 3 6 7

P3: 1 2 3 4 5 3 4 5 3 6 7

Such we can have infinite paths

Such situations use loop testing



# McCabe Path Testing

Complexity, Effort, # of tests....



- Complexity  
 $\#1 > \#2$
- Effort in testing  
 $\#1 > \#2$
- Number of Tests  
 $\#1 > \#2$

# Cyclomatic Number

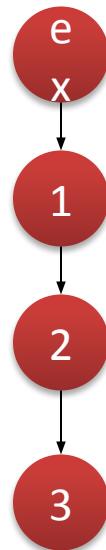
McCabe's cyclomatic number (end 70's)

$$v = e - n + 2$$

e: # of edges in a control graph

n: # of nodes in a control graph

Examples:



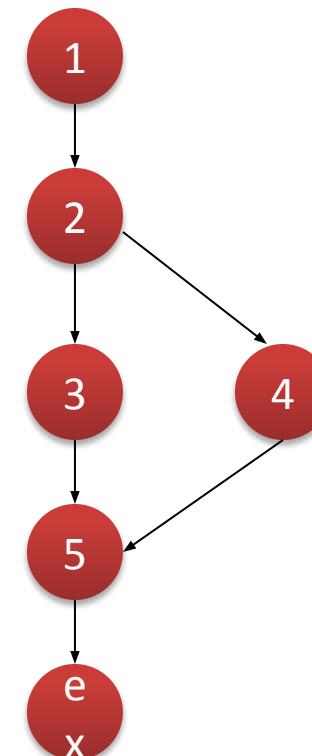
$$e = 3$$

$$n = 4$$

$$v = e - n + 2$$

$$= 3 - 4 + 2$$

$$= 1$$



$$e = 6$$

$$n = 6$$

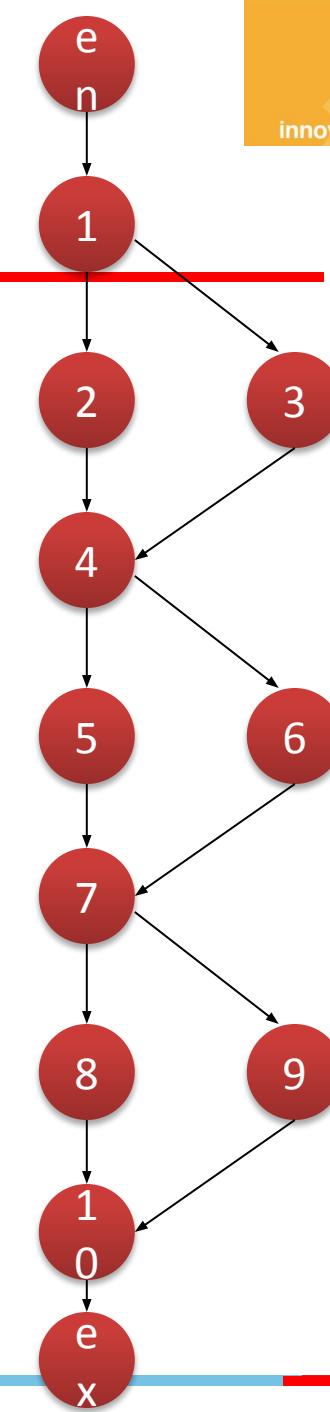
$$v = e - n + 2$$

$$= 6 - 6 + 2$$

$$= 2$$

# McCabe Path

Number of Paths?



# McCabe Path

$$e = 14$$

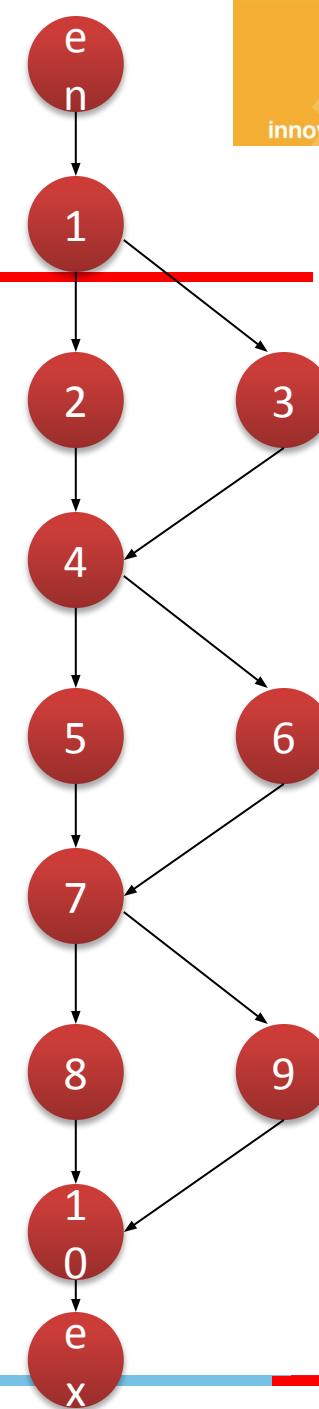
$$n = 12$$

$$v = e - n + 2$$

$$= 14 - 12 + 2$$

$$= 4$$

Please observe carefully the  
four distinct paths



# McCabe – Testing Criteria

---

## Testing Criteria

- Every branch must be executed at least once
- At least “v” distinct paths must be executed
  - $v = e - n + 2$
- # of test cases is a function of program complexity



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

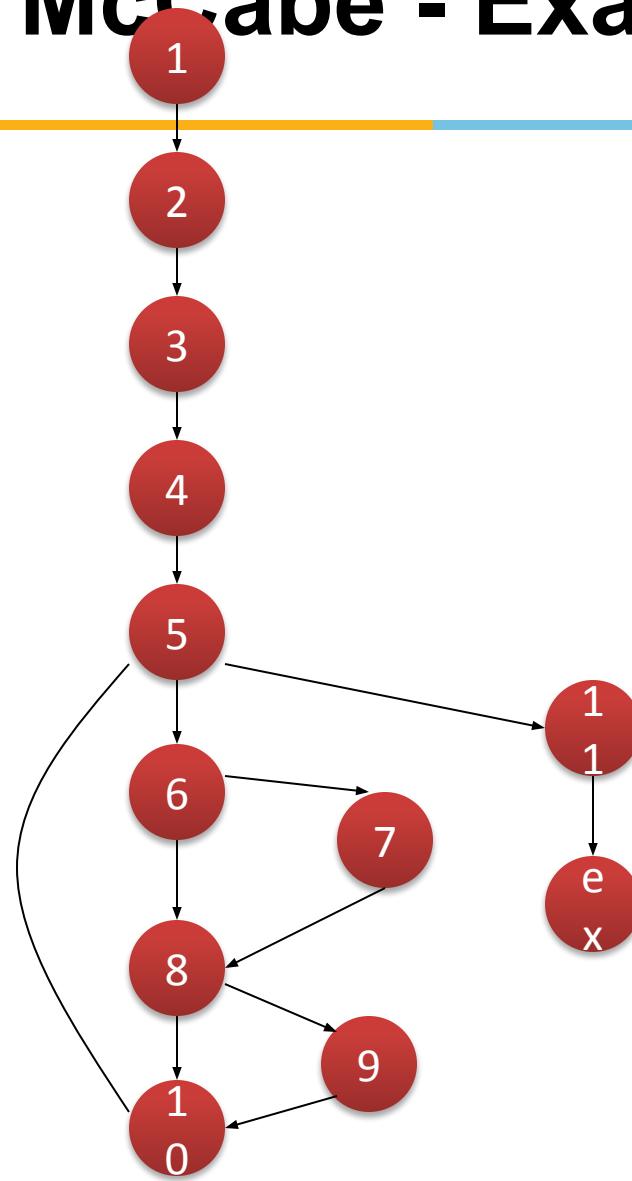


## Topic 5.3: Examples

# McCabe - Example

```
1      input (n, a)
2      max = a[1]
3      min = a[1]
4      i = 2
5      While I <=n do
6, 7      if max < a[i] then max = a[i]
8, 9      if min > a[i] then min = a[i]
10     i = i + 1
11     endwhile
12     output (max, min)
```

# McCabe - Example



$$e = 15$$

$$n = 13$$

$$\begin{aligned} V &= 15 - 13 + 2 \\ &= 4 \end{aligned}$$

4 distinct test cases (minimum)

1. 1 2 3 4 5 11 [n=1 a=5]
2. 1 2 3 4 5 6 8 10 5 11 [n=2 a=5 5]
3. 1 2 3 4 5 6 7 8 10 5 11 [n=2 a= 2 4]
4. 1 2 3 4 5 6 8 9 10 5 11 [n=2 a= 4 2]

1 2 3 4 5 6 7 8 9 10 5 11

$n = ?$

This path is not executable

# McCabe Path Testing

---

- Complexity number gives bound for number of test cases
- Number of test cases is a function of complexity
- Complexity can be used for design as well
- Number of paths can be computed from  
 $v = \text{number of regions} + 1$  as well.



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



## Topic 5.4: Case Study

# Contacts Application

- Create a Contact
- Retrieve a Contact
- Update a Contact
- Delete a Contact
- Share Contact
  - Bluetooth
  - Email
  - WhatsApp
- Fields in a contact



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# Module 6: Agenda

---

## Module 6: Code Based Testing (2/2)

Topic 6.1

Data Flow Testing

Topic 6.2

Path Based Testing - Metric

Topic 6.3

Examples



# Topic 6.1: Data Flow Testing

# 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/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 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  $\text{PATHS}(P)$

# Definitions

# Definitions

# Definitions

*Outdegree: The outdegree of a node in a directed graph is the number of distinct edges that the node as a starting point*

# Definitions

# Definitions

# Data Flow Testing

Concept: Use of Data Flow information for design of Test Cases

- Definition-Use Pair (du pair)

Definition: A definition is a statement that assigns a value to a variable

- $v=x+4$ ; a definition of  $v$
- $\text{input}(v)$ ; read and assign a value to  $v$

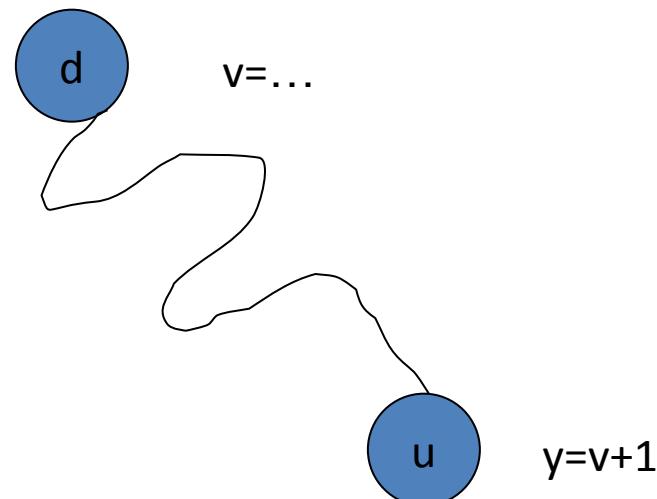
Use: An use is a statement that uses (references) a value of a variable

- $z=v+1$ ; use of  $v$
- $\text{if } (v>0)$ ; use
- $\text{printf}(v)$ ; use
- $v=v+1$ ;

# Definition Use Pair

A definition Use Pair: There exists a definition-use pair between two statements ( d and u) if

- d is a definition of a variable
- u is an use of variable
- There exists a control path in the program from d to u along which the variable is not modified



# Data Flow – Concept Example

```
1 input (x, y)
2 z=x+1;
3 v=x+y;
4 x=0;
5 w=z+x;
```

**No Dataflow  
between 1 to 5 as x  
gets modified @4**

Variable x  
1□2  
1□3  
4□5

Variable y  
1□3

Variable z  
2□5

# Steps for Data Flow Testing

---

- Identify all data flows (all definition-use) pairs in the program
- Design a set of test cases such that each data flow (definition-use pair) is “executed” at least once

# 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



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# **Topic 6.2: Path Based Testing - Metric**

# 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.

H.James Harrington

- 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



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



## Topic 6.3: Examples

# Example

```
1 input(a, n)
2 max=a[1];
3 min=a[1];
4 i=2;
5 while i<n do
6, 7 if max<a[i] then max=a[i]
8, 9 if min>a[i] then min=a[i]
10 i=i+1;
11 output(max, min)
```

# Example: d-u paths

Variable max

<2, 6>  
<2, 11>  
<7,6>  
<7,11>

Variable min

<3,8>  
<3,11>  
<9,8>  
<9,11>

Variable i

<4,5>  
<4,6>  
<4,7>  
<4,8>  
<4,9>  
<4,10>  
<10,5>  
<10,6>  
<10,7>  
<10,8>  
<10,9>  
<10,10>

```

1 input(a, n)
2 max=a[1];
3 min=a[1];
4 i=2;
5 while i<n do
6, 7 if max<a[i] then max=a[i]
8, 9 if min>a[i] then min=a[i]
10 i=i+1;
11 output(max, min)
```

# Example: Solution

---

Test #1:  $n=2$ ,  $a=(2,4)$

Path: 1 2 3 4 5 6 7 8 10 5 11

Test #2:  $n=1$ ,  $a=(5) \quad 2 \square 11$

Test #3:  $n=3$ ,  $a=(2, 4, 3) \quad 7 \square 6$

Test #4:  $n=3$ ,  $a=(5, 1, 2) \quad 9 \square 8$

Test #5:  $n=2$ ,  $a=(5,1) \quad 9 \square 11$

Complete other du pairs



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

### **Brief:**

1. Data flow testing is related to path testing.
2. Node can be either a defined node or a used node.
3. DU path is the path.

### **Use Node:**

Use node has been further categorized as:

1. P-use: Predicate use i.e a node which deals with conditions, e.g if-else etc.
2. C-use: Called as computation use, e.g sum = sum + 2.
3. O-use: Output use, e.g print i.
4. L-use: Location use, e.g a[i] as used in an array.
5. I – Use: Iteration use, e.g i = i + 1.

i = i + 1 is an example of both define and use. It is getting defined at left side and is being used at right.

### **DU Testing Process:**

1. Select the set of paths and test them to find bugs, if any. Bugs can be in any of the following forms: a variable has been defined but is not used; or a variable has been used but not defined.
2. To ensure that all the paths, edges and nodes have been tested.
3. Further bugs can be any of following: variable is defined multiple times before being used; deallocating a variable before it is being used.

### **Example (for practice):**

1. int x, y, z
2. if (x > y)
3. a = x + 2
4. print a  
    else
5. a = y + 4

6. print m

**Program Slices:**

1. If a program is large or complex, it can be further split into various sub-programs.
2. Slicing can be achieved upto any point in the program, but the impact of control statement, if any must be taken in consideration, e.g effect on line number which lies between a loop must be taken care in slice.



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi

# Module 7: Agenda

## Module 7: Model Based Testing (1/2)

Topic 7.1

Model Based Testing – Introduction & Overview

Topic 7.2

Finite State Machines & Fault Model

Topic 7.3

Examples

Topic 7.4

Case Study



# **Topic 7.1: Model Based Testing – Introduction & Overview**

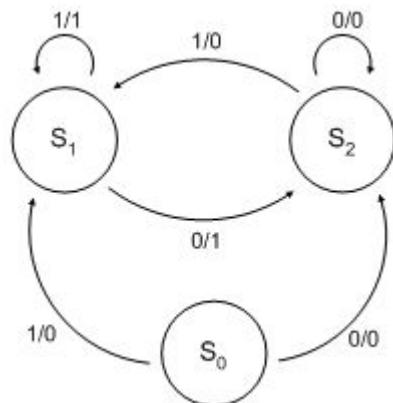
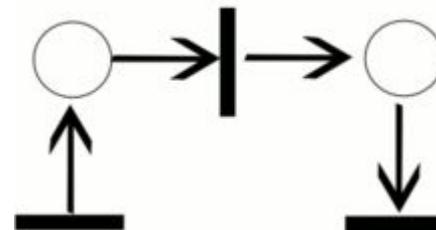
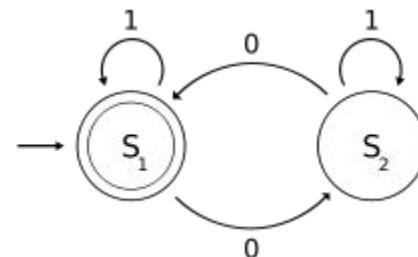
# Model Based Testing

---

- Process of creating a **Model** results in deeper insights and understanding of the system
- Adequacy of MBT – depends on accuracy of the Model
- 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

# Executable Models

- Finite State Machines
- Petri Nets
- StateCharts



# What System Is?

---

- The Components
- Their Functionality
- Interfaces
- All that emphasizes structure

# What System Does?

---

- Decision tables
- State Charts
- Petri nets/EDPN
- FSM/EFSM
  
- All these describe System Behaviour
- Look for expressive capabilities of the system

# Modelling

---

- What the system is
  - Emphasize structure
  - Components, their functionality and interfaces
  - DFD, Entity/Relation models, hierarchy charts, classes diagrams and class diagrams
- What the system does
  - Emphasize behavior
  - Decision Tables, FSM, State Charts & Petri Nets

Refer: Page 225 and 226 of T1

# Model Based Testing Tools

---

- Modelling the system provides ways to generate test cases automatically
- Example: <http://graphwalker.org/index>

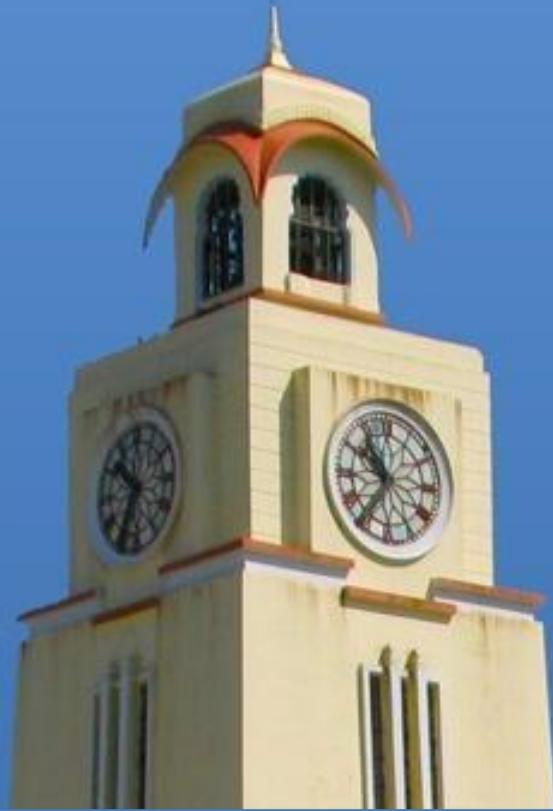


# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



# **Topic 7.2: Finite State Machine & Fault Model**

# Finite State Machines

---

- Method of expression of a design
- Simple way to model state-based behavior
- State Charts – a rich extension of FSM
- Petri nets – useful formalism to express concurrency and timing

# State Based Testing

---

- State “Behaviour” exhibited
- Example: Stack
- Operation pop

```
s.push(5);  
y=s.pop();  
print(y);
```

Y=5

```
s.push(5);  
s.push(7);  
y=s.pop();  
print(y);
```

Y=7

# State Based Testing

---

- Testing state based components
- State-full
- State-less
- Testing only individual methods or functions for state-full components is not sufficient

# State-full Component

---

- A set of States
- Transition between states

# State-full Component

---

- Objects/Classes
- Control Systems
- Embedded Systems
- Communication Systems
- ...

# State Based Component

States	Values (of some data)
Empty State	top=0
Full State	top=10
Partial State	$1 \leq top \leq 9$

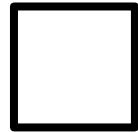
# State Based Modeling Techniques

---



- State transition diagrams
- Extended Finite State Machines

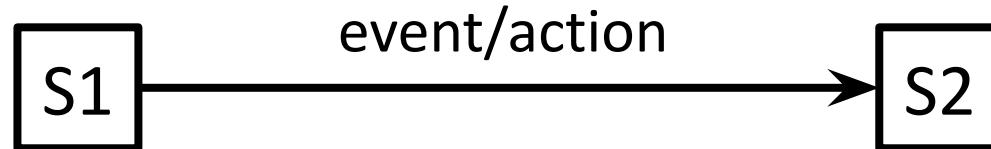
# State Transition Diagram



A State



A transition



# The Fault Model

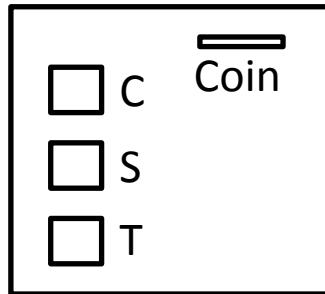
- Process of Design
- Conforming of the implemented system to the Requirements
- Fault Model defines a set of small set of possible fault types that can occur
- Our focus here is FSM or EFSM (*Lets talk modelling later!*)

# Fault Categories

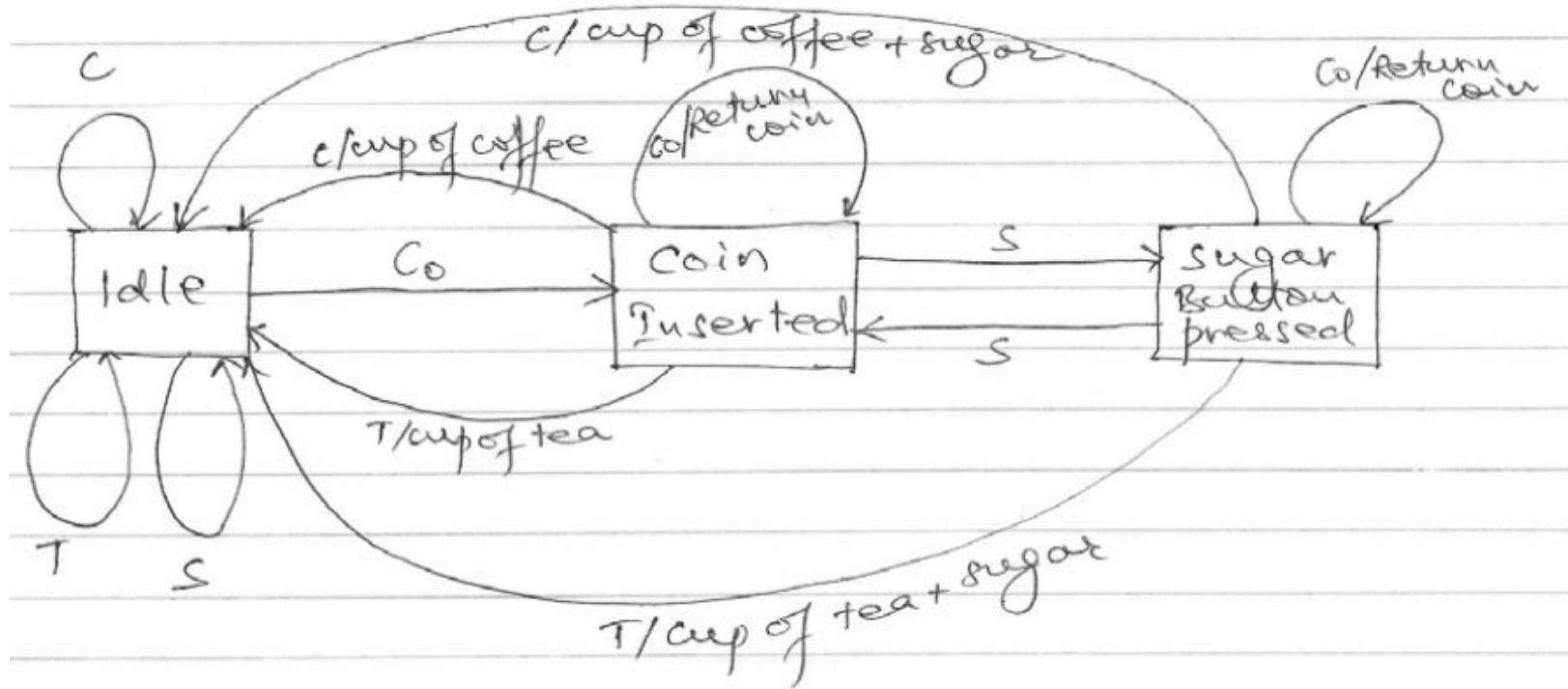
---

- Operation Error
  - Error generated upon transition
  - Incorrect output function
- Transfer Error
  - Incorrect state transition
- Extra State Error
- Missing State Error

# Vending Machine



C: Coffee Button pressed  
 T: Tea Button pressed  
 S: Sugar Button pressed  
 Co : Coin inserted



# Some examples to discuss

---

- Garage Door
- Building Lighting Control System
- Lift/Elevator Control System (One or Multiple)
- A MMI (Man-machine interface) Interface of an instrument



# Software Testing Methodologies



**BITS** Pilani

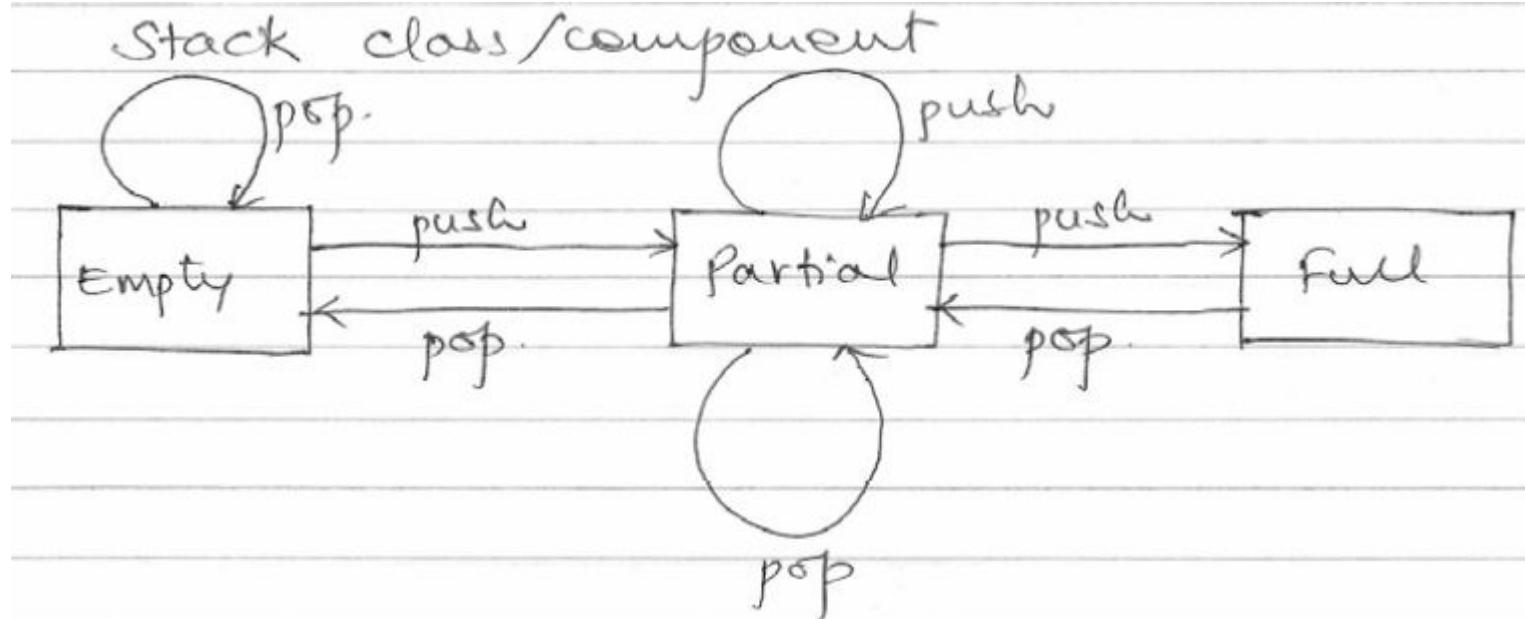
Prashant Joshi



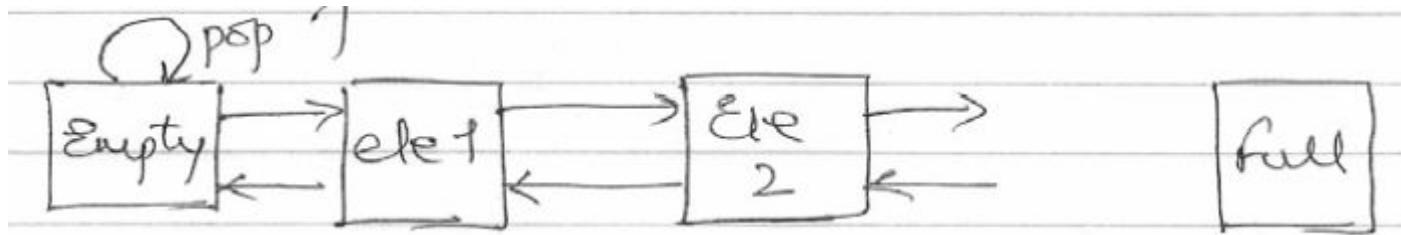
## Topic 7.3: Examples

# Stack

- Simple stack
- Operations (push and pop)



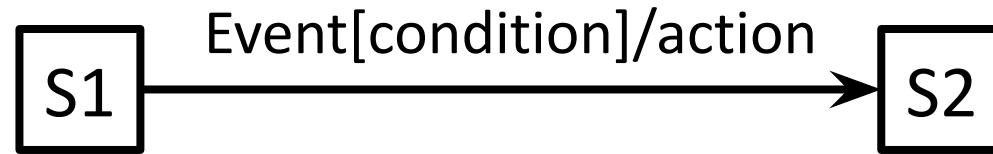
# Notion of State Explosion



- Too many states
- State Explosion problem!

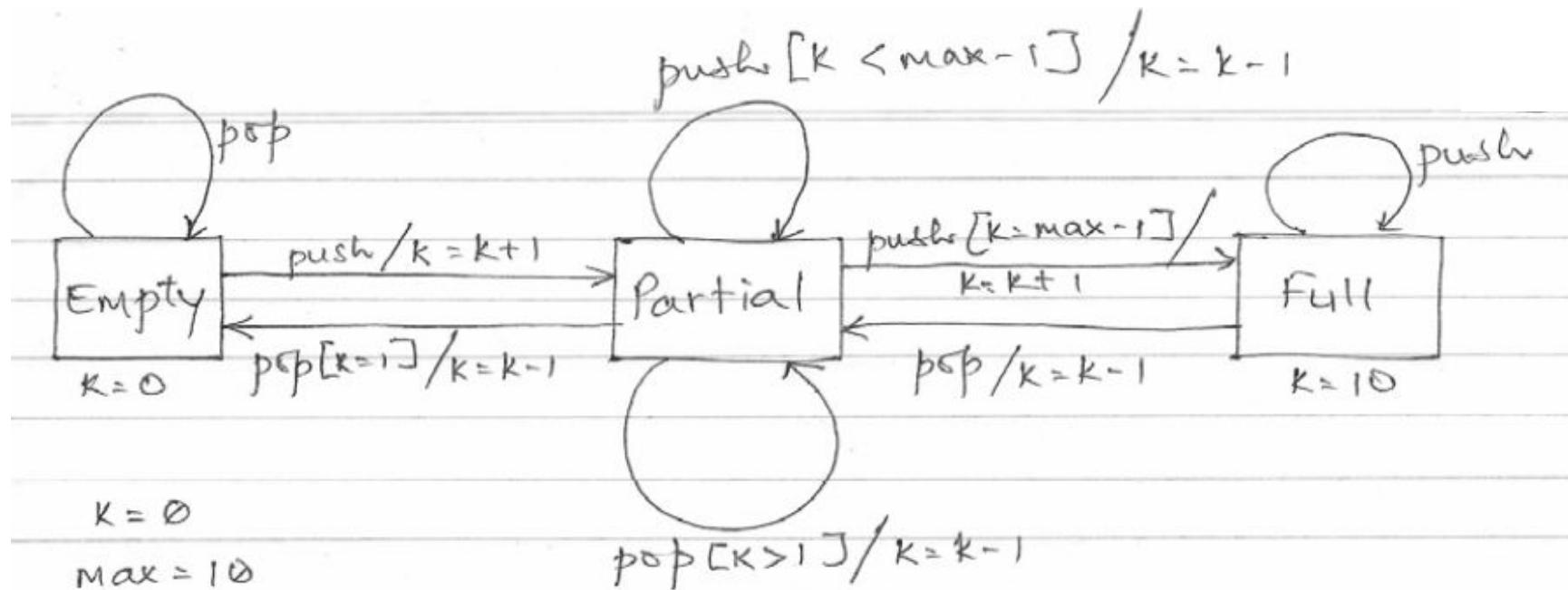
# Extended FSM

- Extended Finite State Machine
- Extension of the state transition diagram by introducing
  - Variables
  - Conditions



1. The system is in S1
2. Event occurs
3. Condition evaluates to true
4. Transition from S1 to S2 takes place
5. Action is performed

# Stack



$K = \emptyset$

$\max = 10$

# Testing Stack Component

---

- Operations/methods
  - Push
  - Pop
- State based testing

# Testing with Criteria

---

- State Testing
  - Every state in the model should be visited at least once
- Transition Testing
  - Every transition in the model is “traversed” at least once
- Path Testing
  - Traverse every path in the model at least once

# State Coverage

Test #1: s.push(5) //partial state

Test #2: s.push(5)  
s.push(7)      10 push operations

s.push(20) //full state

- State Coverage Satisfied

# Transition Coverage

Test #3: y.pop()

Test #4: s.push(5)

y.pop()

Test #5: s.push(5)

s.push(7)

s.push(20)

s.push(12)



11<sup>th</sup> push

# Transition Coverage

Test #6: s.push(5)

s.push(7)

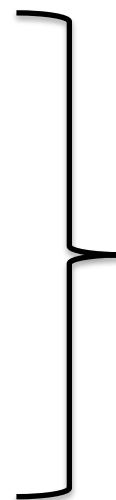
y=s.pop()

Test #7: s.push(5)

s.push(7)

s.push(17)

y=s.pop()



10 push

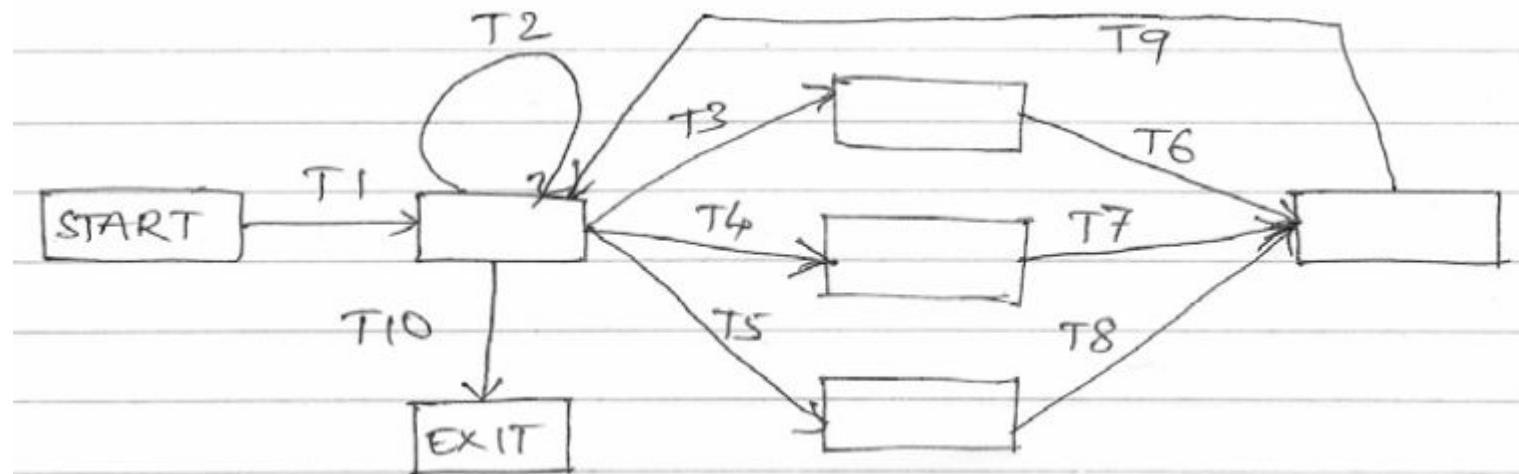
# Constrained Path Testing

---

- Modified Path Testing
- Traverse every path in the model under the constraint that any transition in the path is traversed at most N times

# Constrained Path Testing

Use of n=1 (Say repeat only once)



# Constrained Path Testing

T1: T1, T10

T2: T1, T2, T10

T3: T1, T3, T6, T9, T10

T4: T1, T2, T3, T6, T9, T10

T5: T1, T3, T6, T9, T2, T10

T6, T1, T4, T7, T9, T10

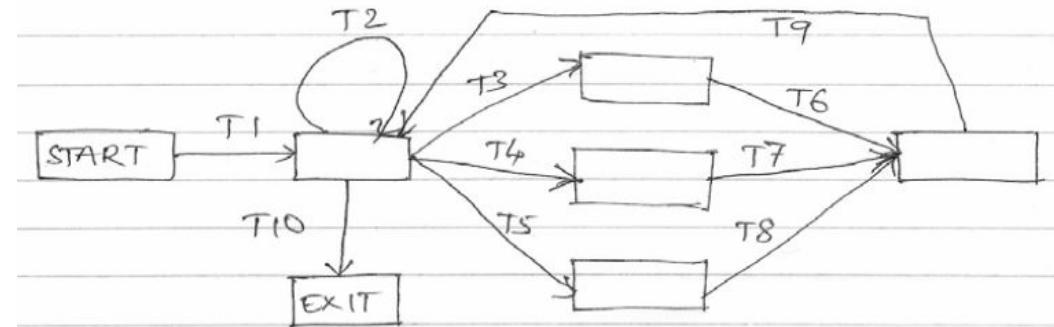
T7, T1, T2, T4, T7, T9, T10

T8: T1, T4, T7, T9, T2, T10

T9: T1, T5, T8, T9, T10

T10: T1, T2, T5, T8, T9, T10

T11: T1, T5, T8, T9, T2, T10



# State Based Testing

---

- We use state model to design test cases using different strategies
  - State Testing
  - Transition Testing
  - Path/Constraint path testing
- Non-executable elements e.g comments in a code.



# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi



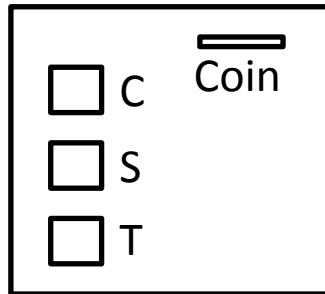
## Topic 7.4: Case Study

# Simple Vending Machine

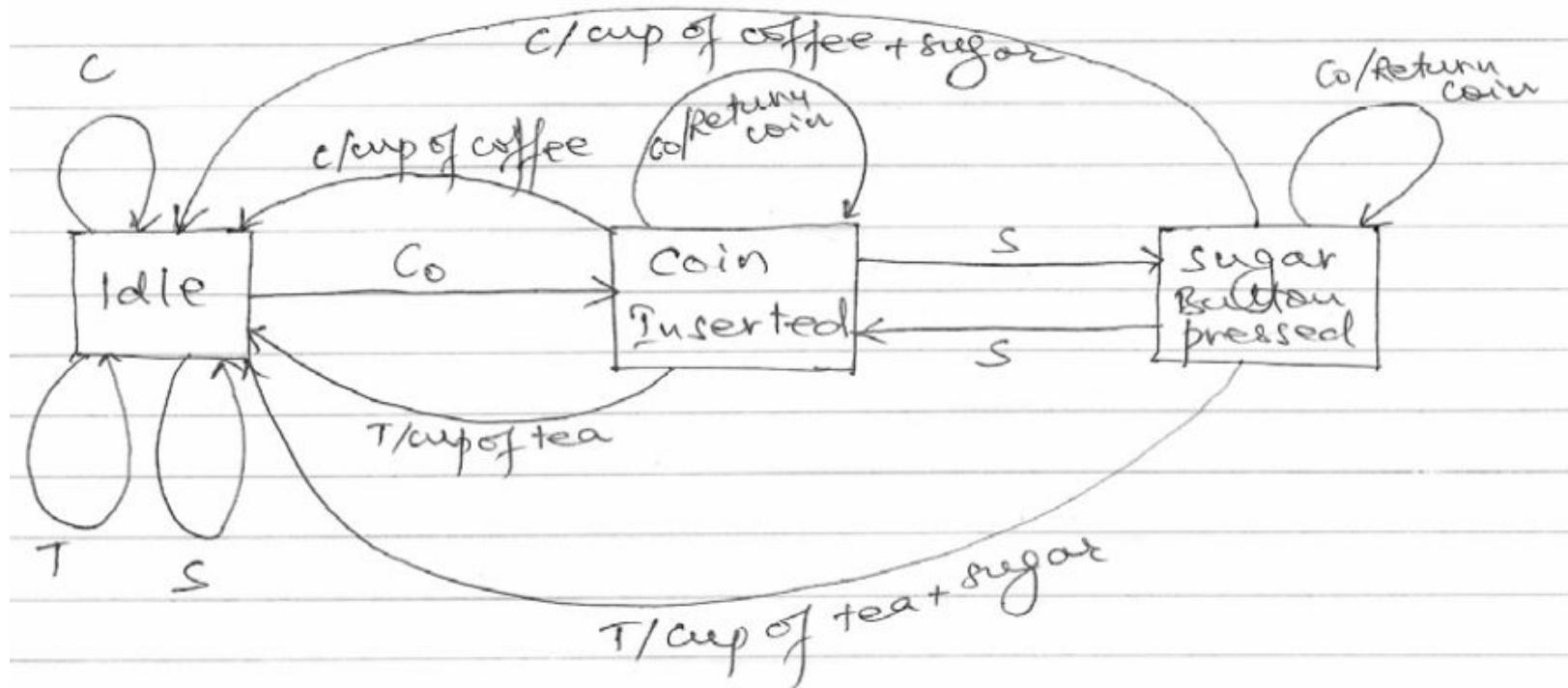


- Tea/Coffee vending Machine
- Options
  - Accepts token/coin
  - Sugar

# Vending Machine



C: Coffee Button pressed  
 T: Tea Button pressed  
 S: Sugar Button pressed  
 Co : Coin inserted





# Software Testing Methodologies



**BITS** Pilani

Prashant Joshi