



BITS Pilani

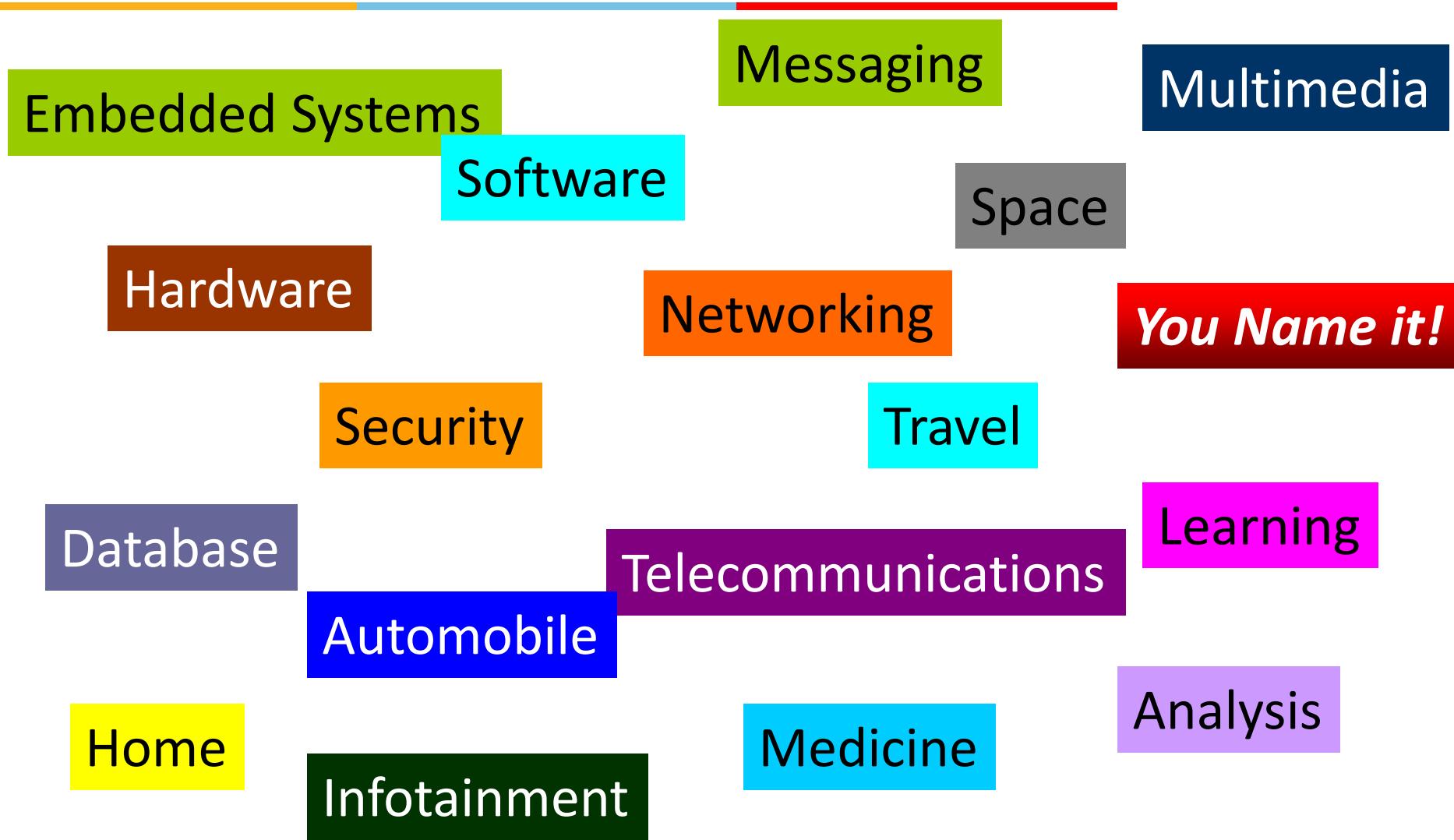
Software Testing Methodologies

Prashant Joshi

A definition

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

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

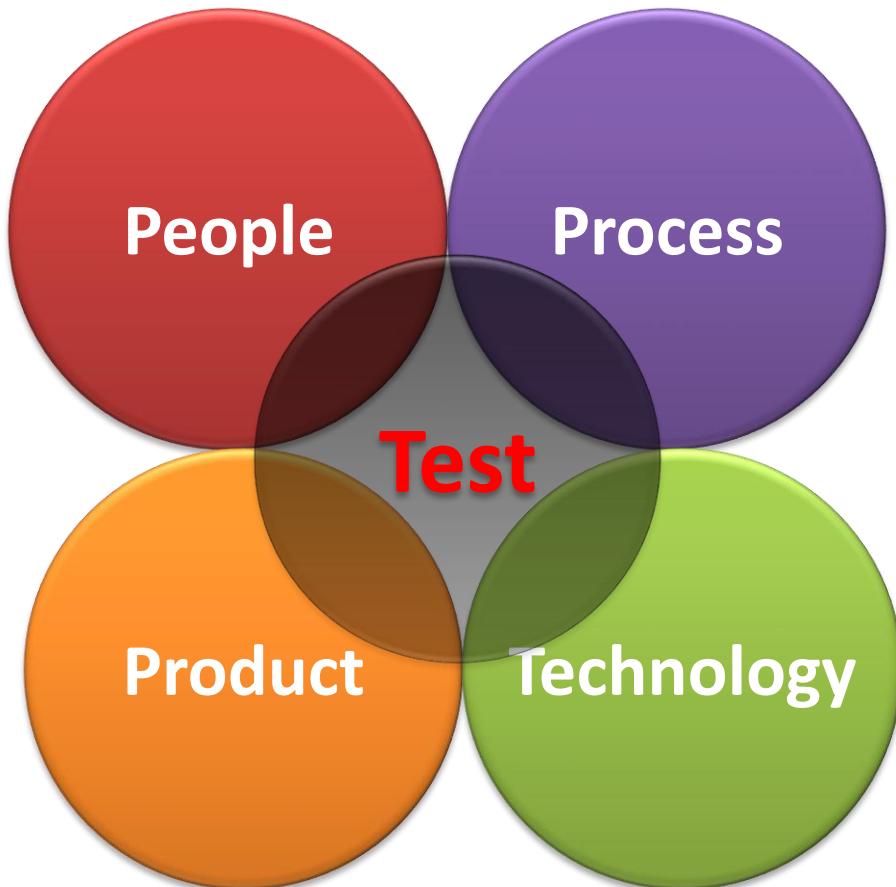
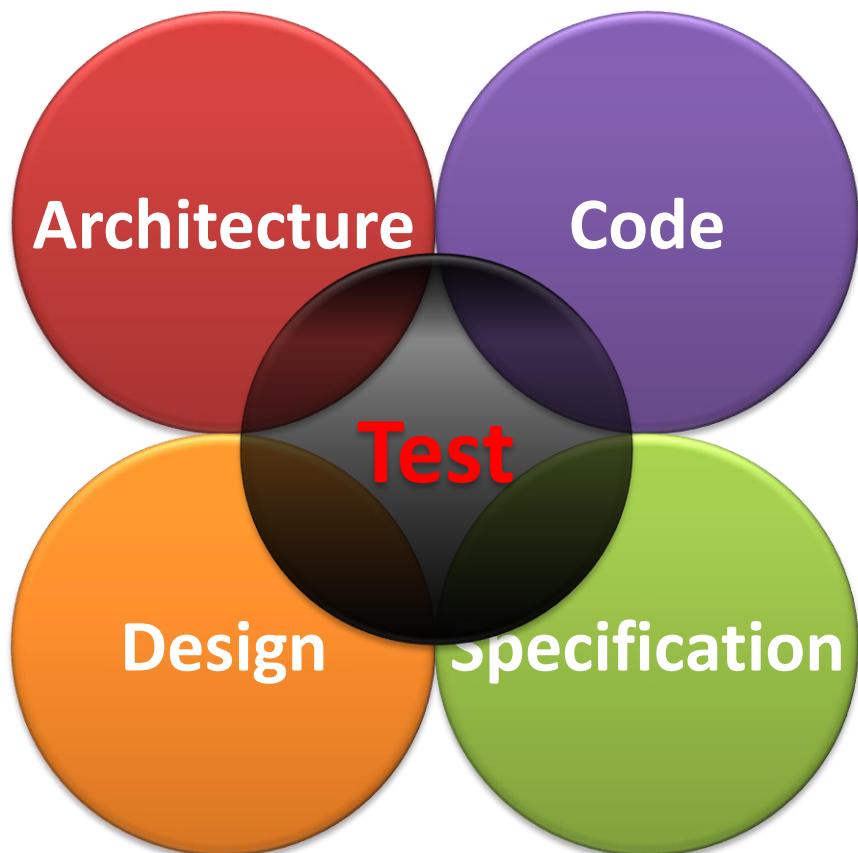
Home

Infotainment

Medicine

Analysis

Building Blocks



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?**
-



Goals of Software Engineering

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

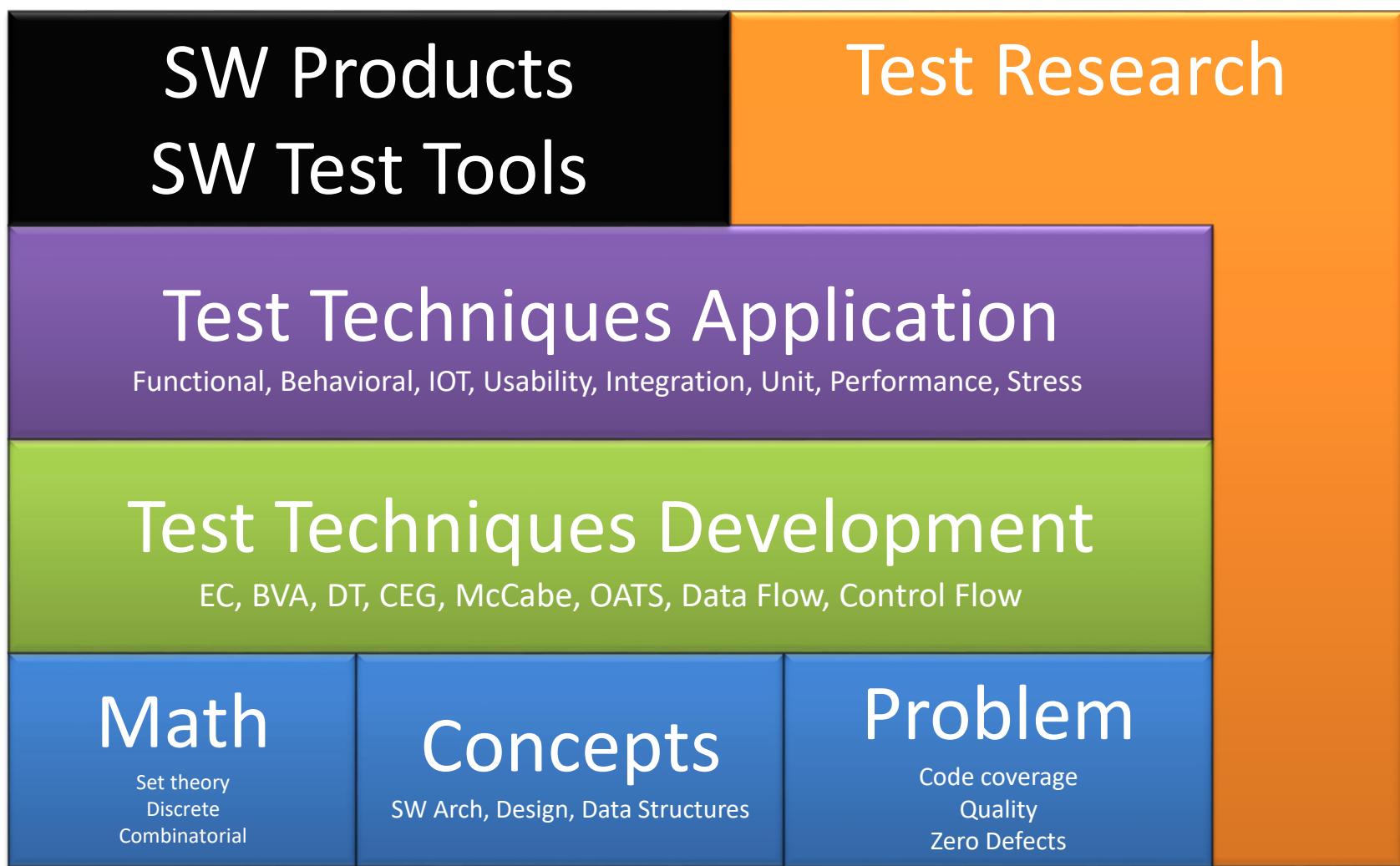
And many more...

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

Building Blocks – A View



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

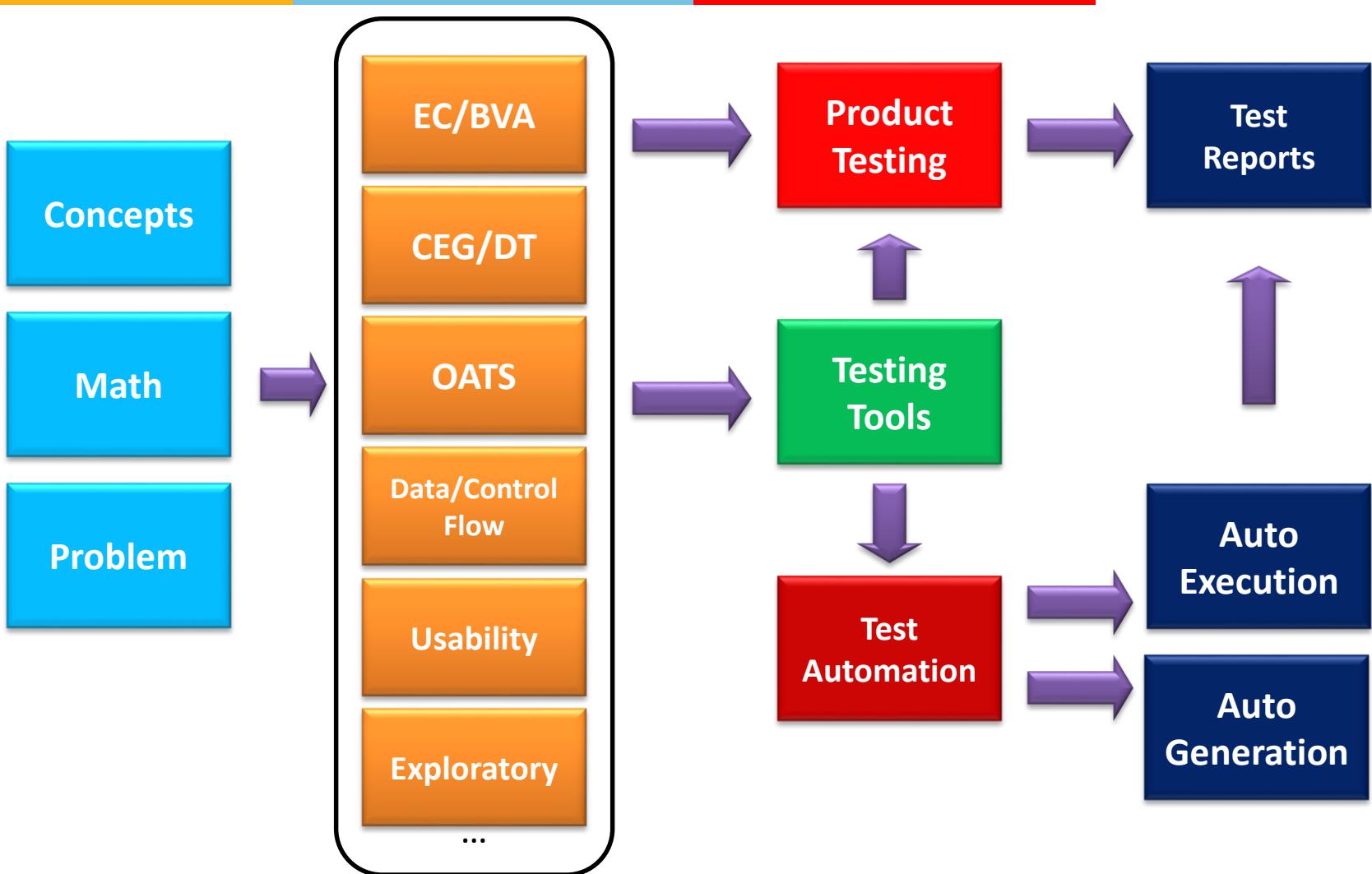
Techniques based on nature of application

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

Usage based testing

- Operational Profile
- Reliability Engineered Testing

Progression



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

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?

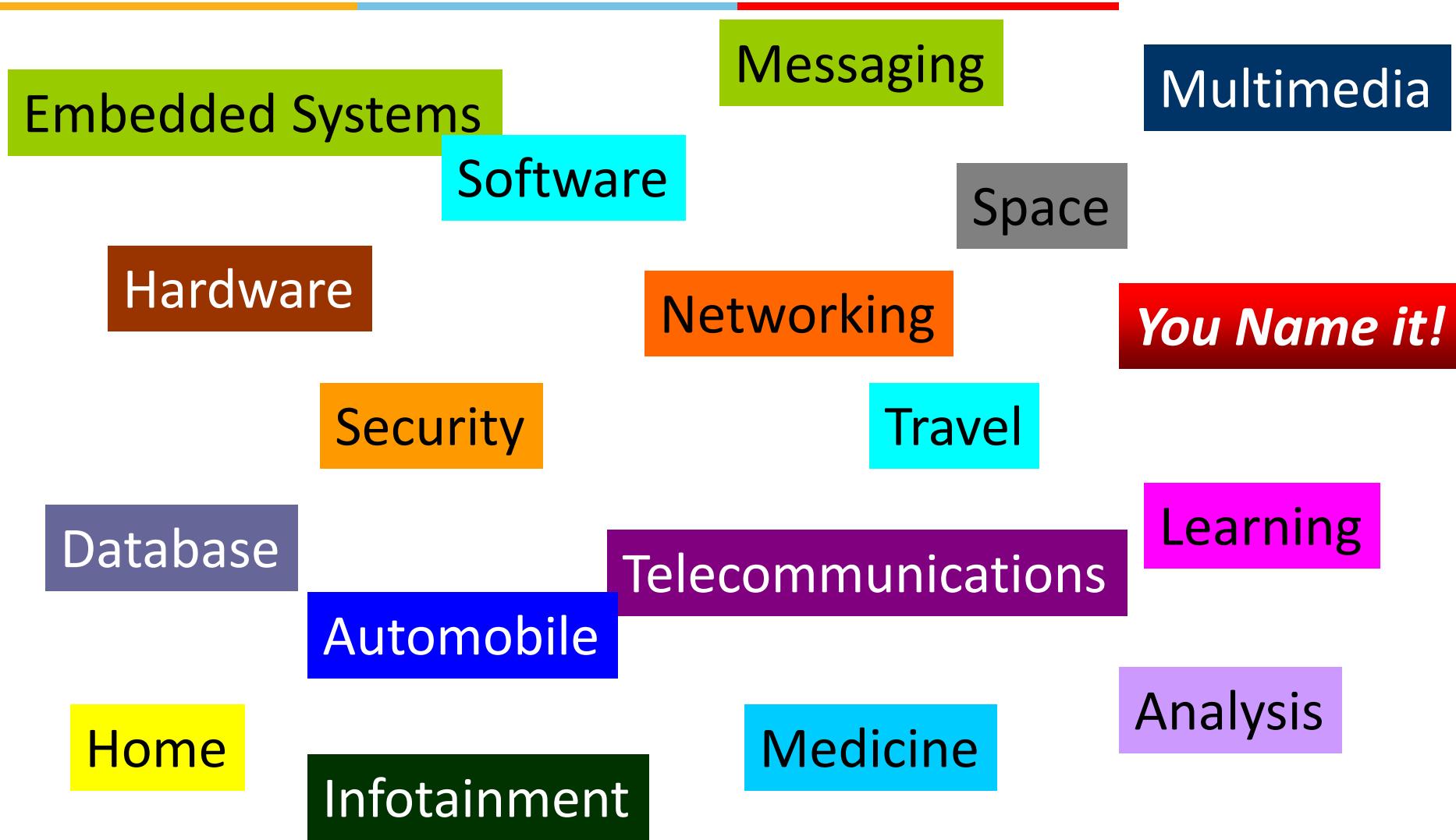
A definition

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

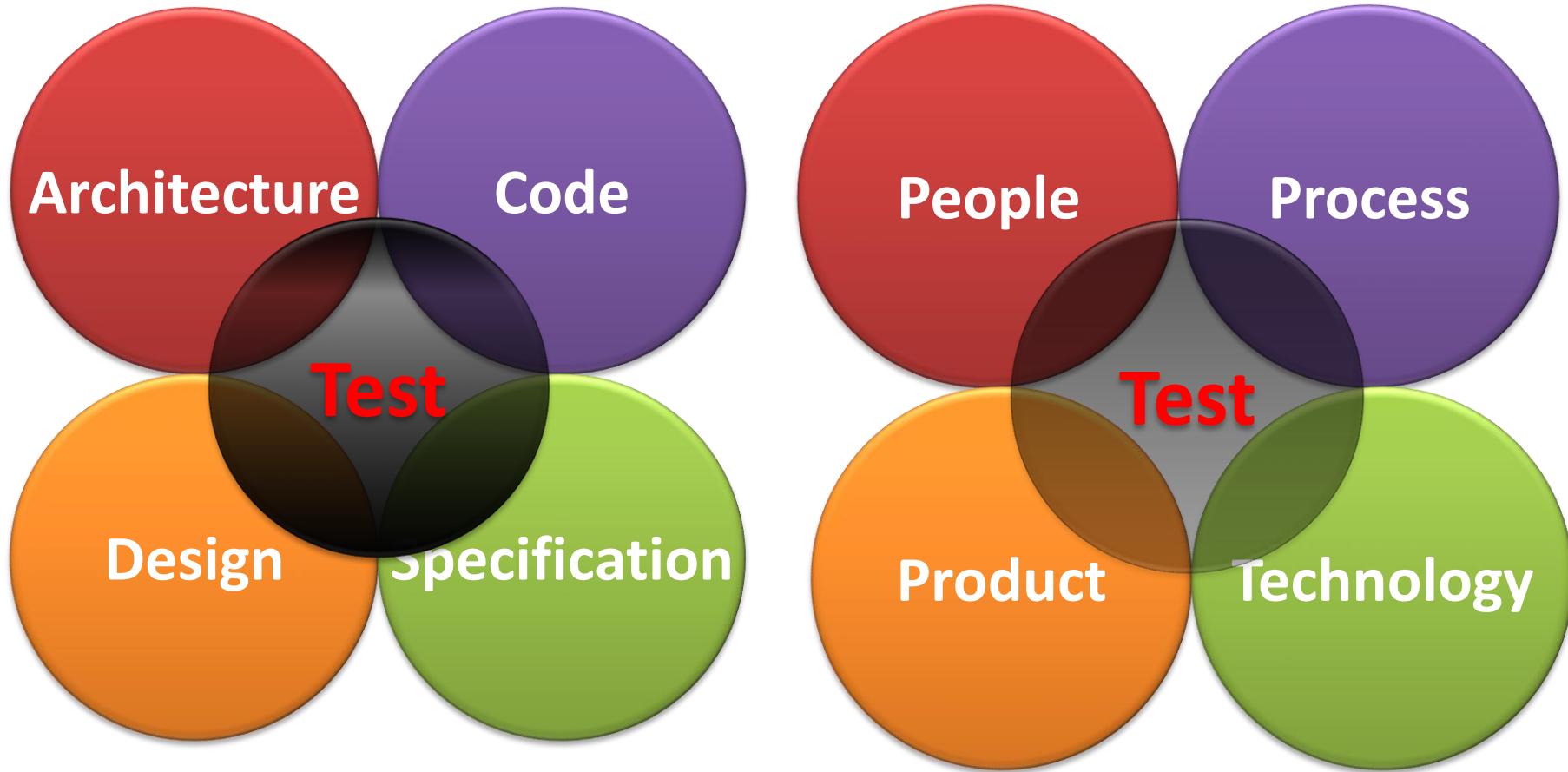
Home

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



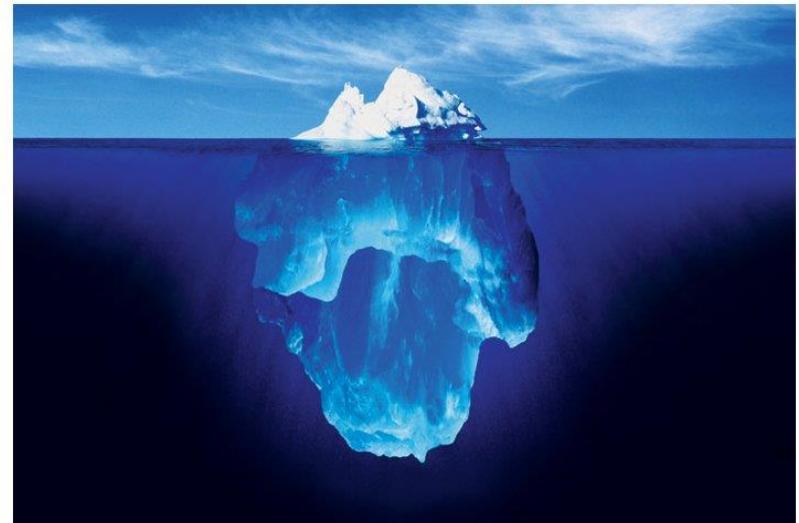
Technology

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



Topic 1.2: Overview of the Course

Learning Principles

Make everything as simple as possible, but not simpler

- Albert Einstein

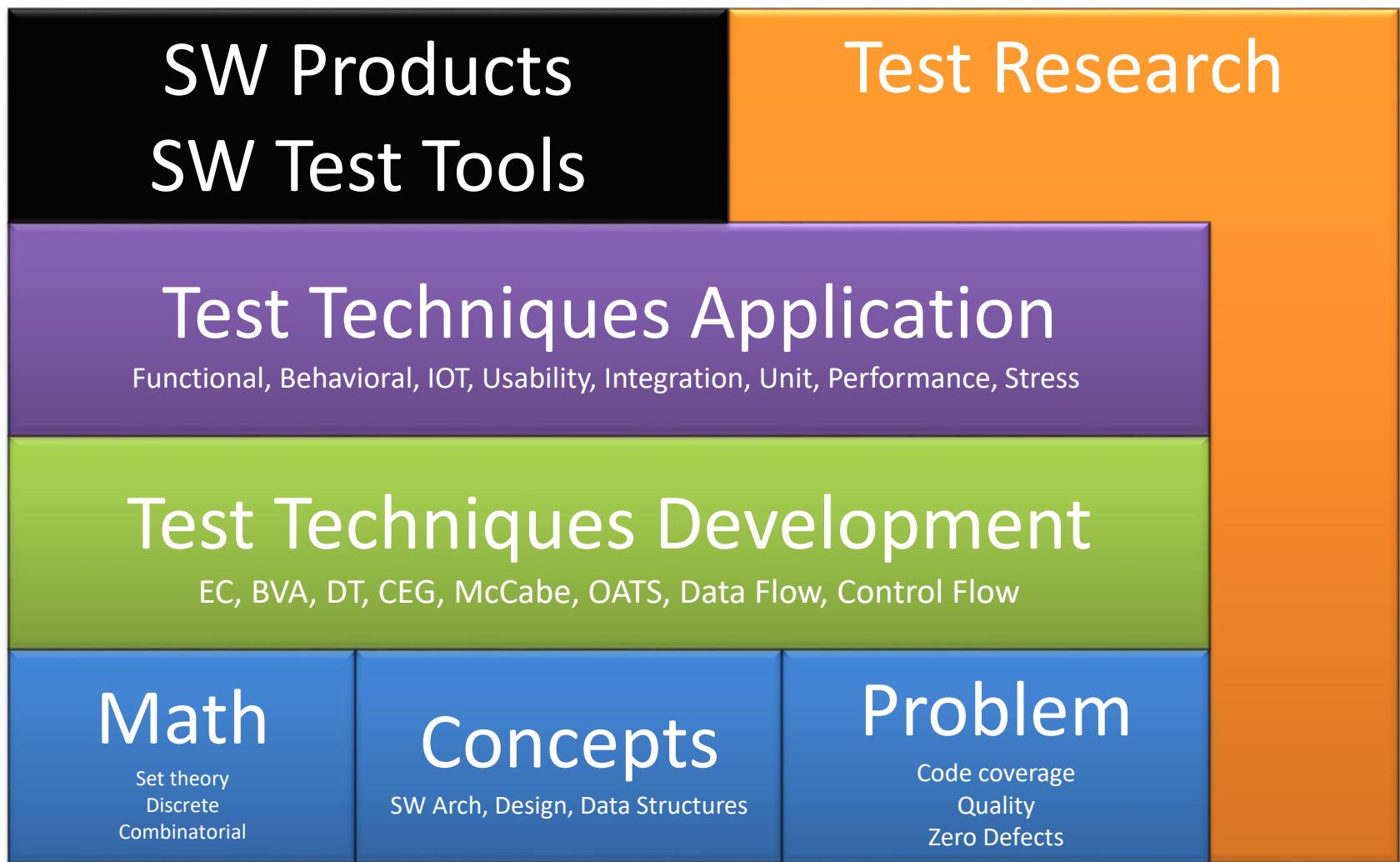
I hear and I forget, I see and I remember, I do and I understand

- 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

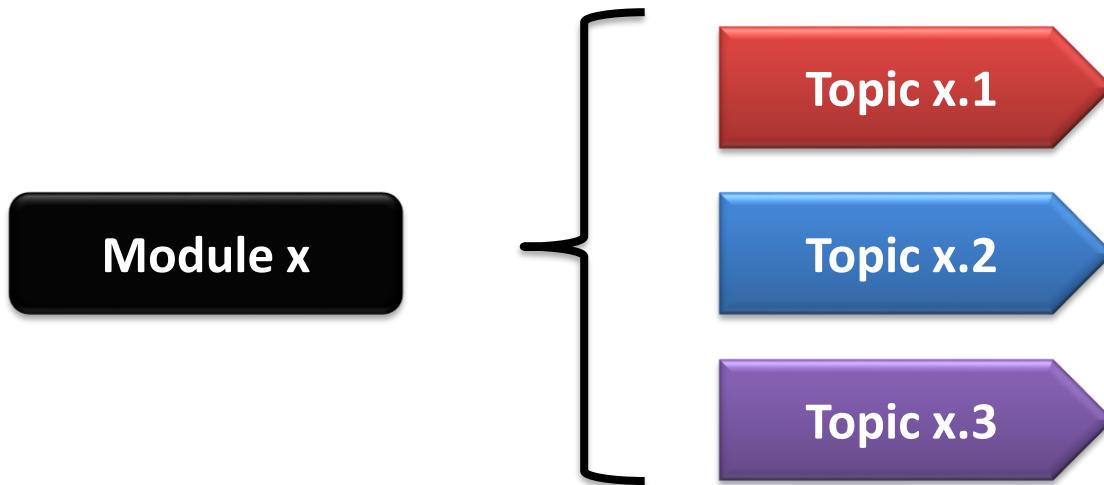


Ask the Questions!



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

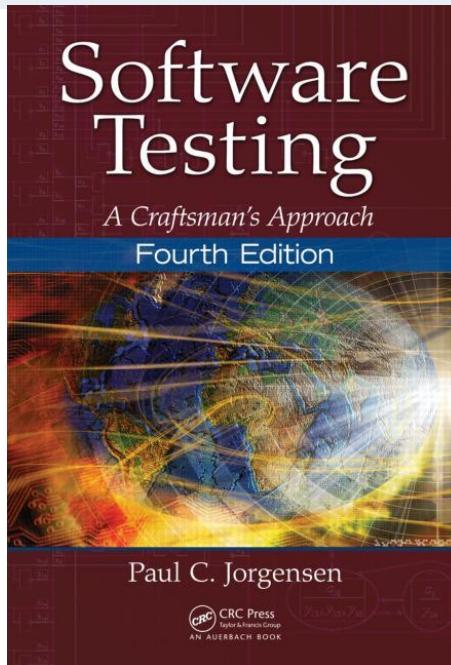
Module No	Module Title	Objectives
1	Introduction to Software Testing & Techniques	Introduce the course and course handout. Bring a perspective of need and motivation for this course. Provide an overview of the course, quality attributes, levels and types of Testing
2	Mathematics and Formal Methods	Provide 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.

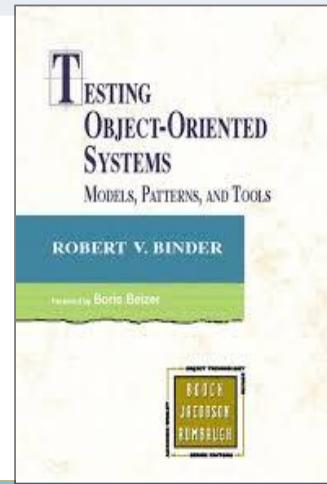
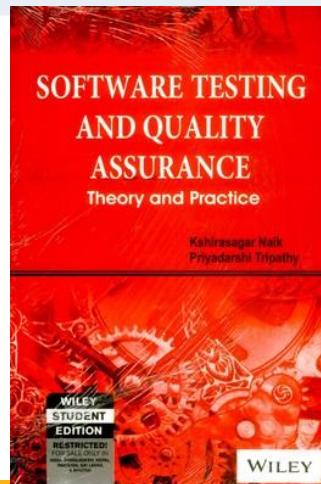
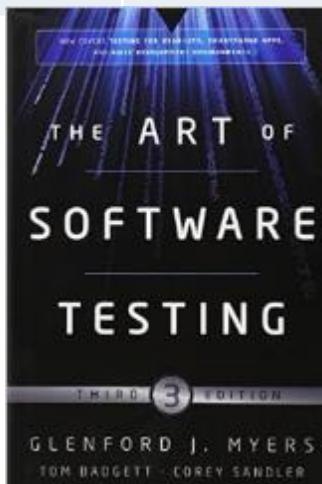
Text Books

- | | |
|----|---|
| 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 |





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

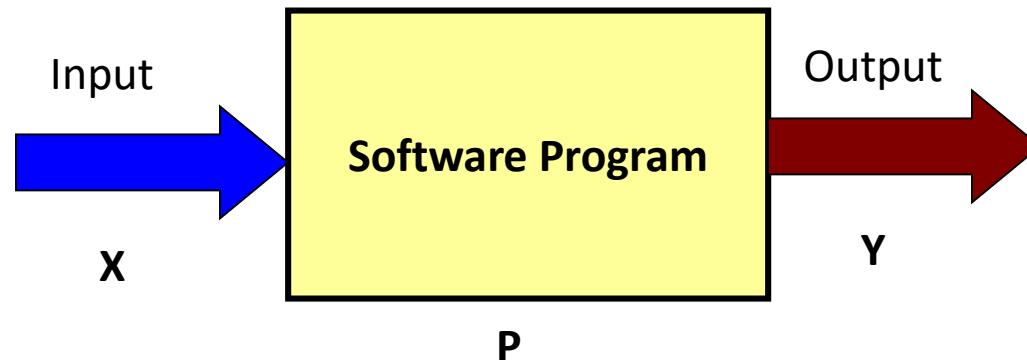
Techniques based on nature of application

- 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;

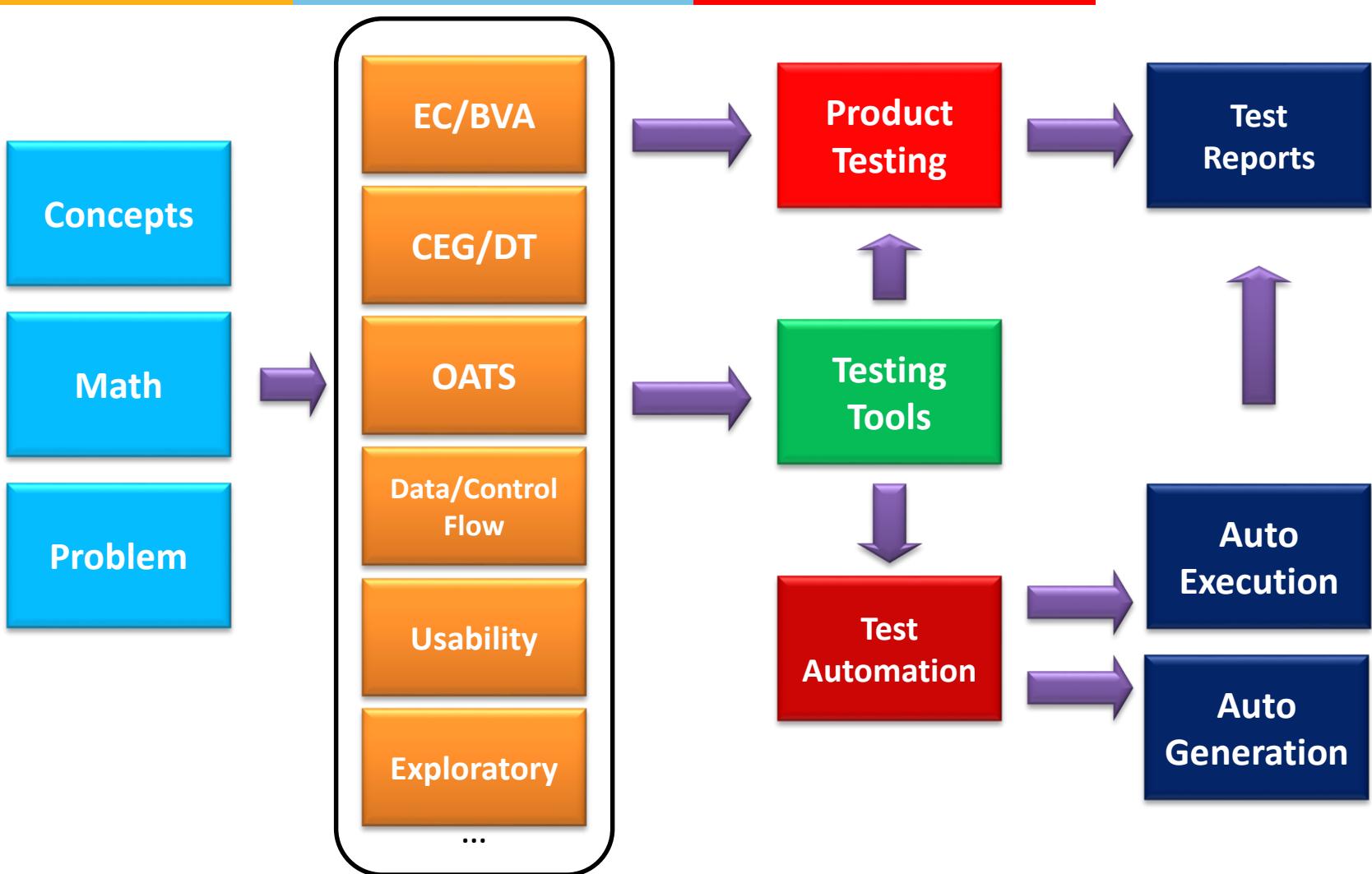
    //Logic to find the max. Compare with other two.
    if(a<b) {
        max = b;
    }
    if(max<c) {
        max=c;
    }

    // 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





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

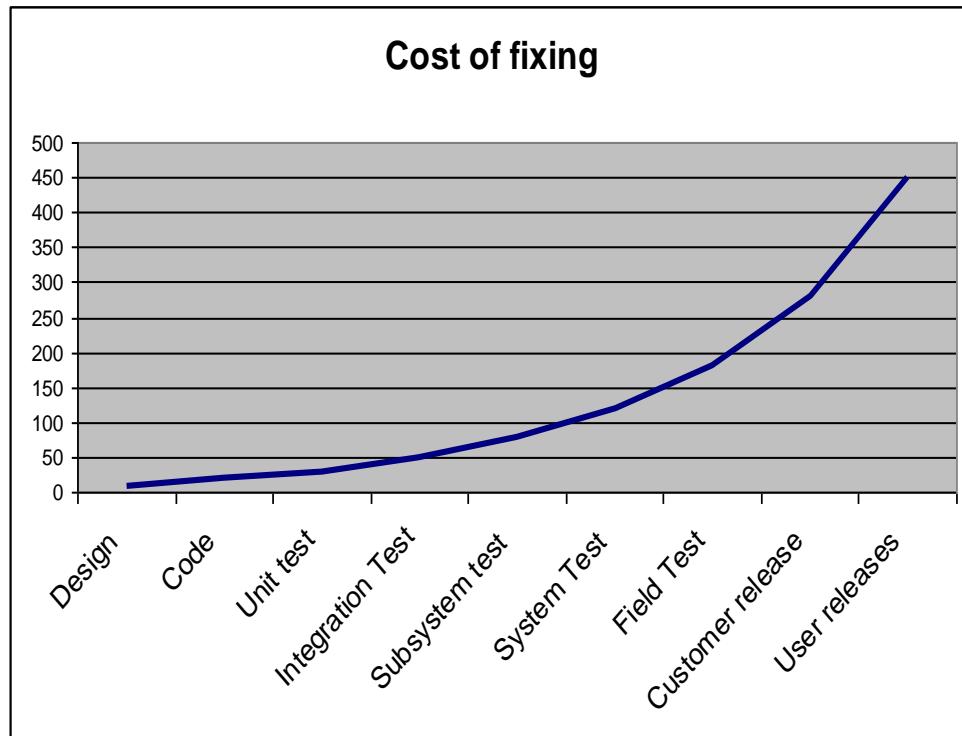
A Test Case

- **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

Module 1 Self Study

- SS1.1 Make a list of day-to-day products that you use. Create a prioritised list of quality attributes applicable to the SW from users perspective and from engineers perspective
- SS1.2 List the software test techniques that you use on a day to day basis



SS1.1 Products around you and ones you use

1.1 Self Study

To explore:

- Various products around us
- Quality attributes for these products

Study Work:

- You are required to explore your surroundings at home, office, transit, cafes etc. and make a list of products you see/use. For each of these devices make a note of (a) specific use (b) think on the design and development of the product
- Create a list of quality attributes from engineers and users perspective
- Discuss your findings with your fellow students



SS1.2 List the software test techniques that you use on a day to day basis

1.2 Self Study

To explore:

- List of software test techniques that you use on regular basis

Study Work:

- Make a list of all the software techniques you use on a daily basis
- Create notes of the type of techniques, the basis of the technique and it's effectiveness

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

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

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

Propositional Logic

p	q	$p \wedge q$ (AND)	$p \vee q$ (OR)	$\sim p$ (NOT)
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

p	q	$p+q$ (EX-OR)	$p \rightarrow q$ (IF-THEN)
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T

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

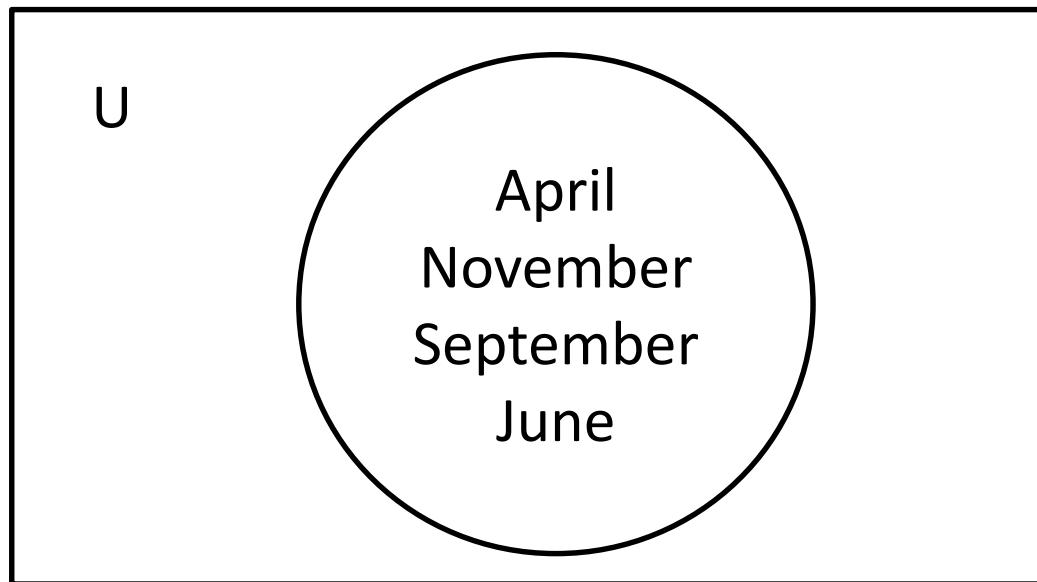
$$Y = \{year: 1800 \leq year \leq 2014\}$$

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

Graph

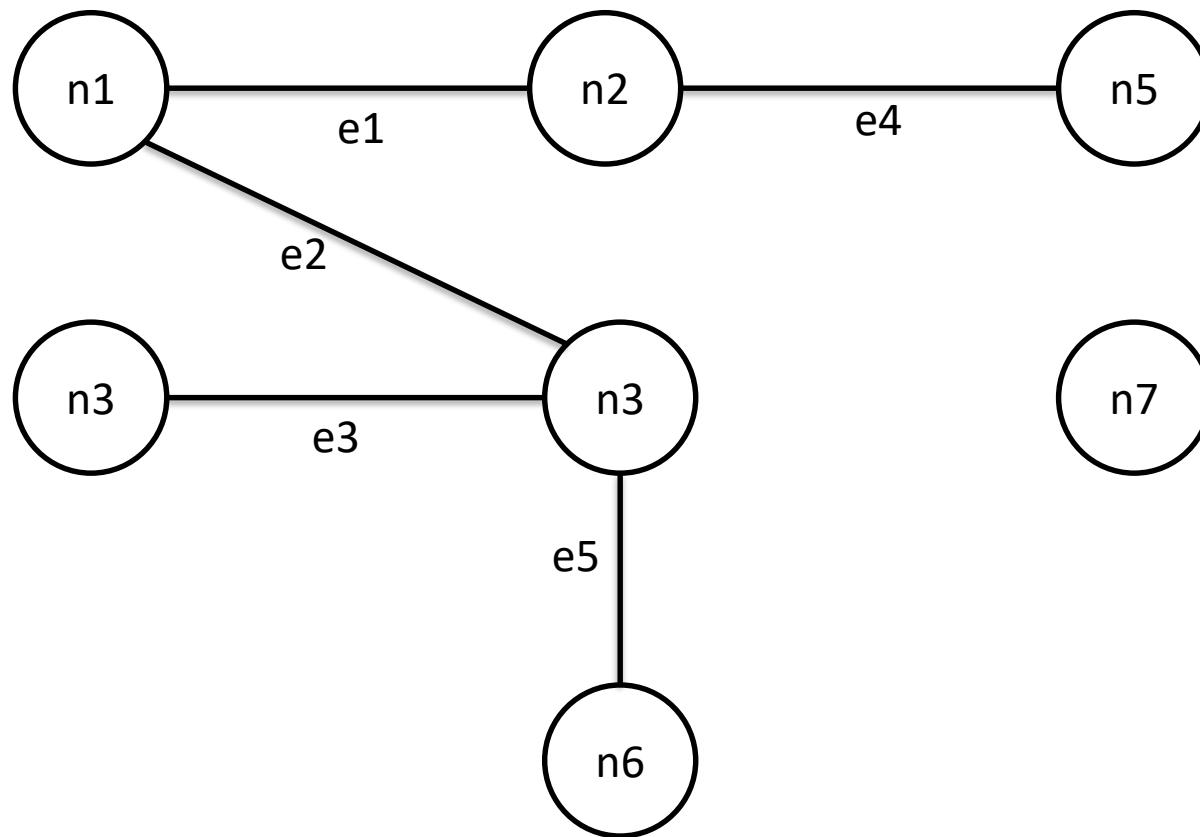
- A graph (also known as linear Graph) is an abstract mathematical structure defined from two sets – set of nodes and set of edges that form connections between nodes
- Example: Computer Network
- Definition
- *A Graph $G = (V,E)$ is composed of a finite (and nonempty) set V of nodes and a set of E of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

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

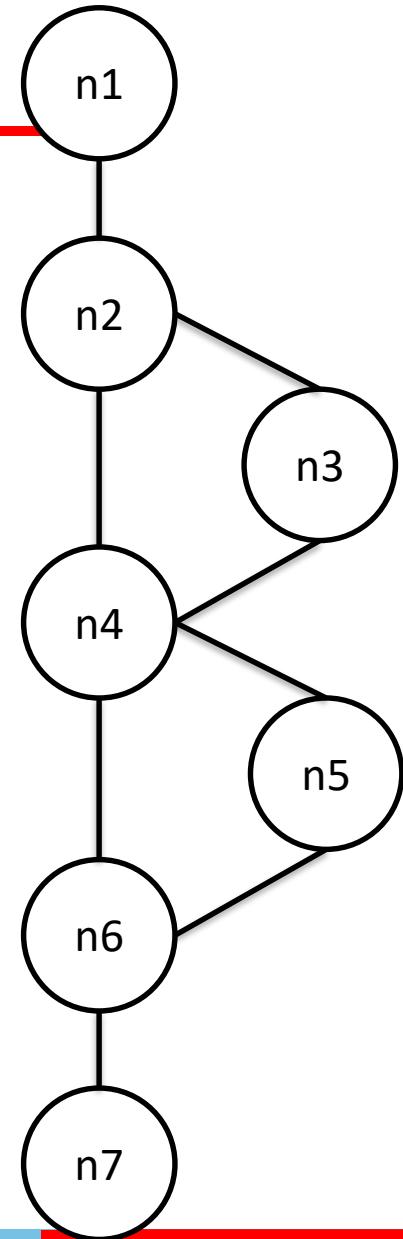
Paths

- 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?



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

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



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

p	q	$p \wedge q$ (AND)	$p \vee q$ (OR)	$\sim p$ (NOT)
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

p	q	$p+q$ (EX-OR)	$p \rightarrow q$ (IF-THEN)
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T



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

p	q	$p \wedge q$ (AND)	$p \vee q$ (OR)	$\sim p$ (NOT)
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

p	q	$p+q$ (EX-OR)	$p \rightarrow q$ (IF-THEN)
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

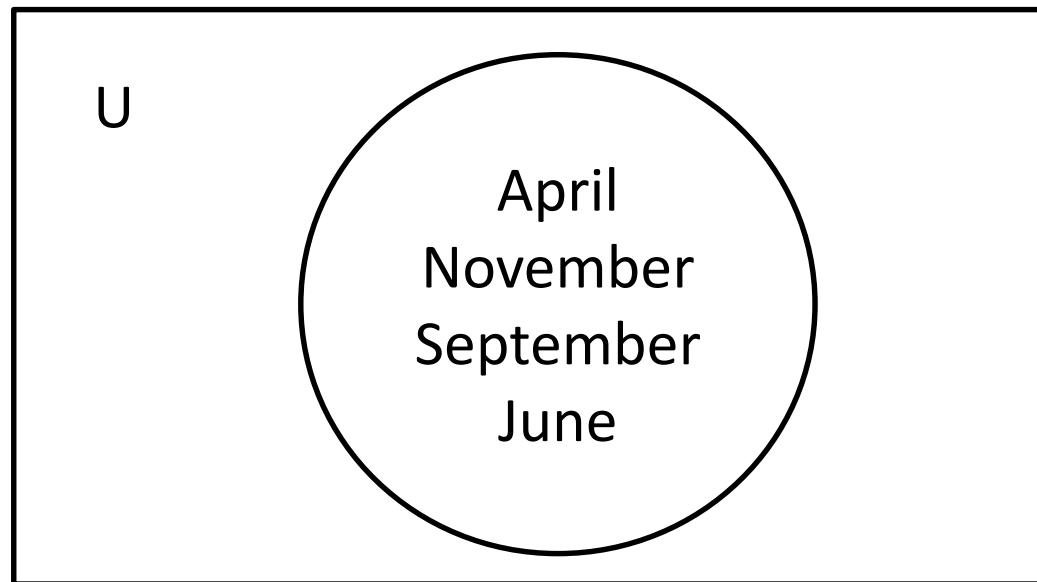
$$Y = \{year: 1800 \leq year \leq 2014\}$$

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

- Particularly important for test engineers
- Divide the whole in parts
- Given a set B , and a set of subsets $A_1, A_2 \dots, A_n$ of B , the subsets are a partition of B iff

$$A_1 \cup A_2 \cup \dots \cup A_n = B, \text{ and}$$
$$i \neq j \Rightarrow A_i \cap A_j = \emptyset$$



Topic 2.4: Graph Theory

Graph

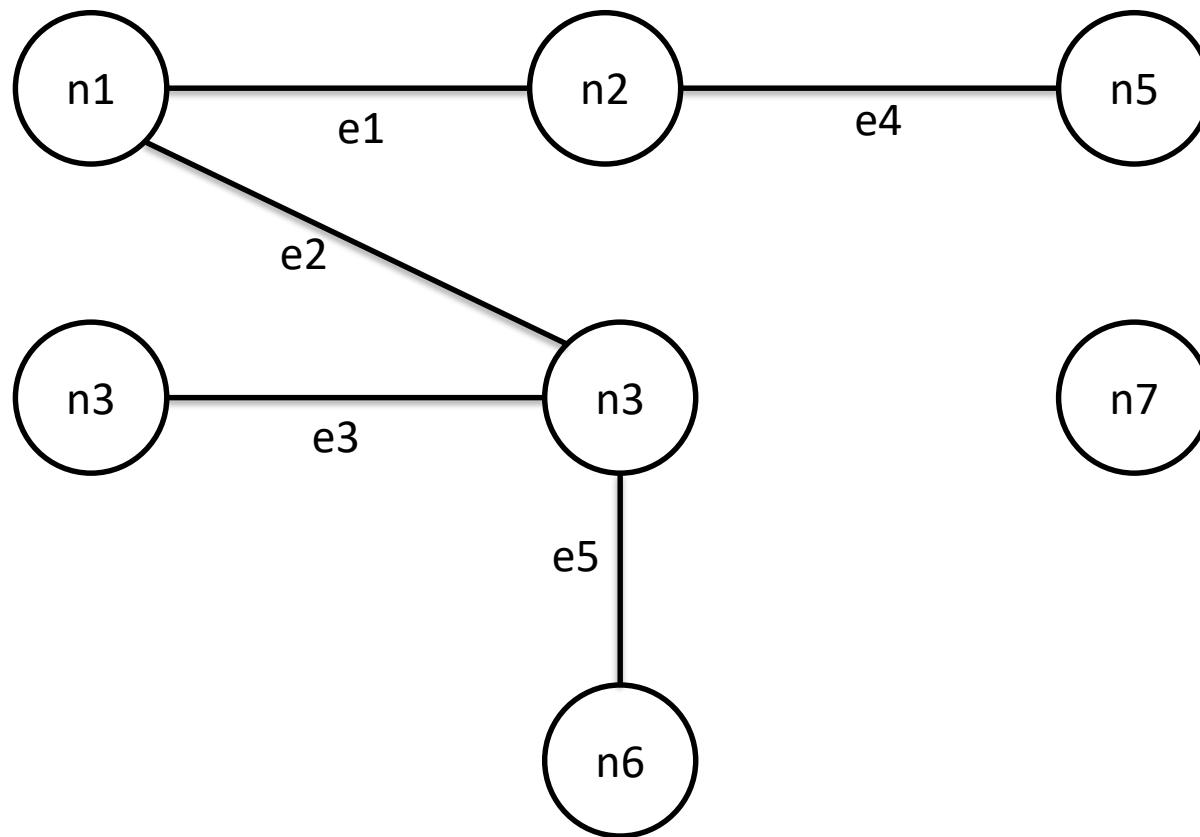
- A graph (also known as linear Graph) is an abstract mathematical structure defined from two sets – set of nodes and set of edges that form connections between nodes
- Example: Computer Network
- Definition
- *A Graph $G = (V,E)$ is composed of a finite (and nonempty) set V of nodes and a set of E of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

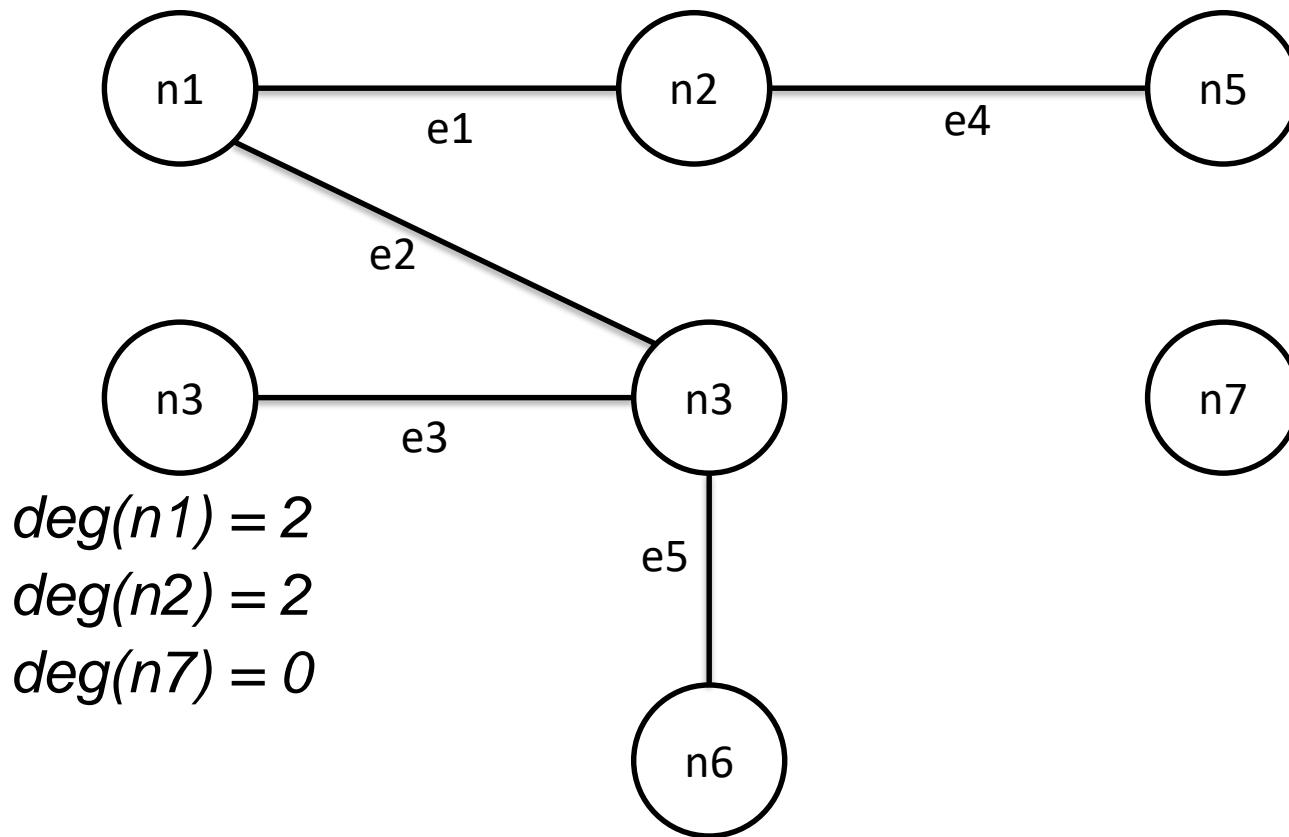


Use of representation

- 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

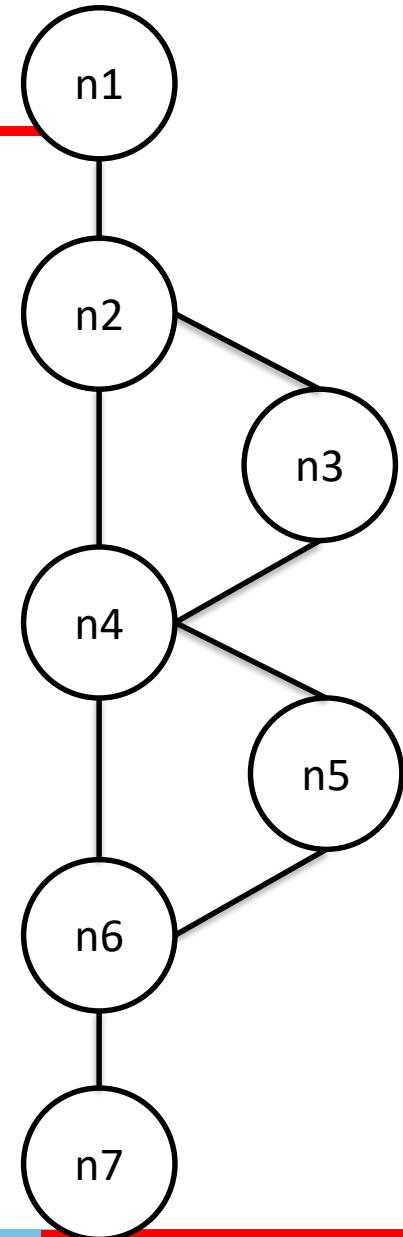
Paths

- 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?



Module 2 Self Study

- SS2.1 Map the techniques with the math concept
- SS2.2 Study the chapter 3 & 4 from T1



SS 2.1 Math Concepts \diamond Test Techniques

2.1 Self Study

To explore:

- Mathematics concepts and their direct use in test techniques

Study Work:

- Create a list of all the test techniques you know or can find from various sources
- Map all the known test techniques with their corresponding mathematics concepts on which they are based



SS2.2 Study Math for Test Engineers

2.2 Self Study

To explore:

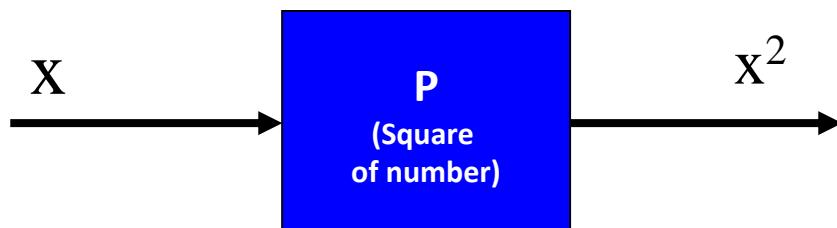
- Study math for test engineers

Study Work:

- Study the chapters 3 and 4
- Solve math problems so as to ensure understanding of the concepts

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 ($x \cdot x$)
- B. successive addition ($x + x \dots x$ times)

Perspectives

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

Equivalence Class

- **Equivalence Classes (EC)**
- 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

Recommendations for Identifications of EC

- Equivalence class for invalid inputs
- Looks for Range in numbers
- Look for membership in a group
- Analyze 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

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. $10 < x < 100$

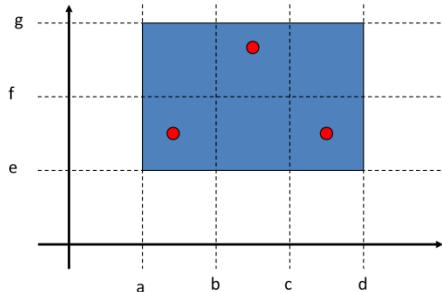
Generalising BVA

- **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

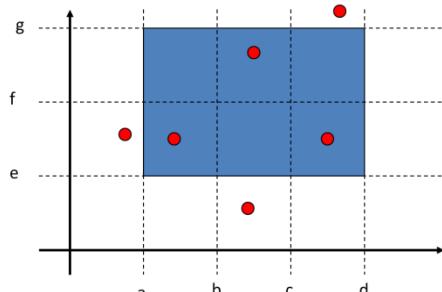
BVA Limitations

- 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

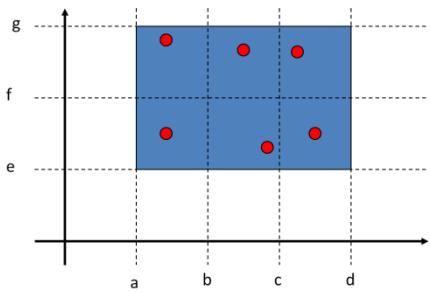
EC & BVA



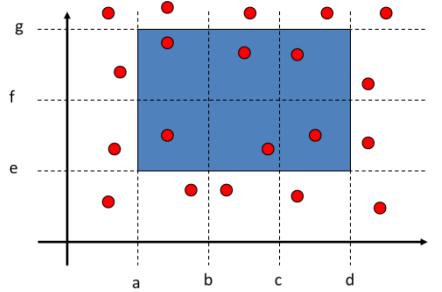
Weak Normal



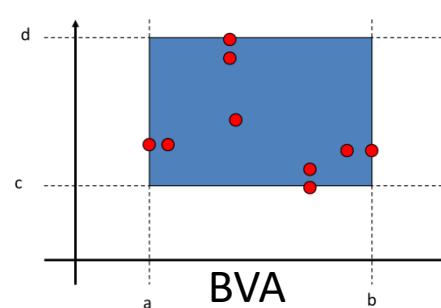
Weak Robust



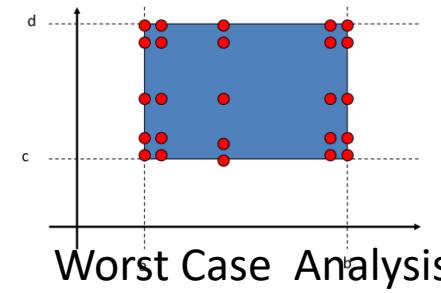
Strong Normal



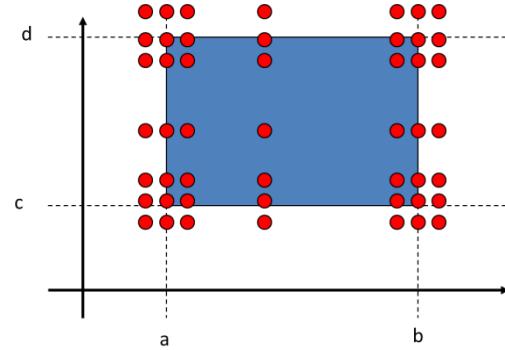
Strong Robust



BVA



Worst Case Analysis



a b

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 required for the problem

Example – Max of 3 numbers

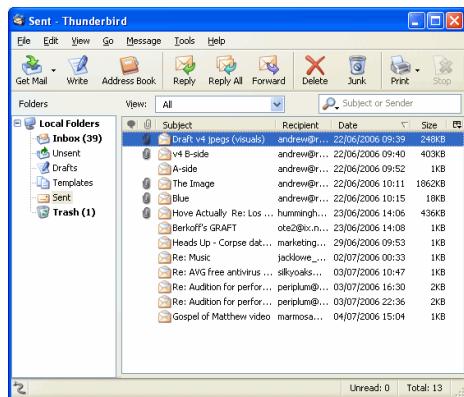
#	Input Condition	Valid Sub-domain		Invalid Sub-domain	
1	Number a	$0 \leq a < 100$	(1)	$a < 0$ $a > 100$	(2) (3)
2	Number b	$0 \leq b < 100$	(4)	$b < 0$ $b > 100$	(5) (6)
3	Number c	$0 \leq c < 100$	(7)	$c < 0$ $c > 100$	(8) (9)
4	Max	a b c	(10) (11) (12)	?	
5	Two equal	a & b b & c c & a	(13) (14) (15)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

Examples

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



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

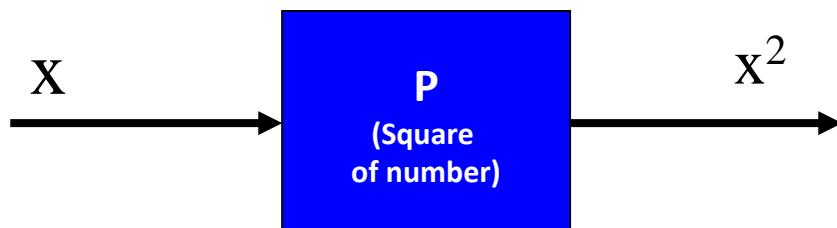
Examples & Case Study



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 ($x*x$)
- B. successive addition ($x+x\dots x$ times)

Approaches & “View”

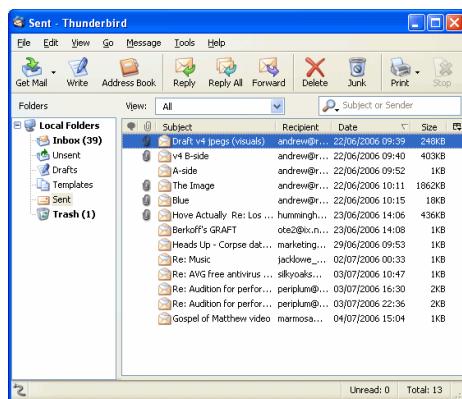
- Purpose is to uncover defects
- Demonstrate the system works (Treat this as a by-product!)
- Validate that it functions per specifications
- Works as specified – always!

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
- ...





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

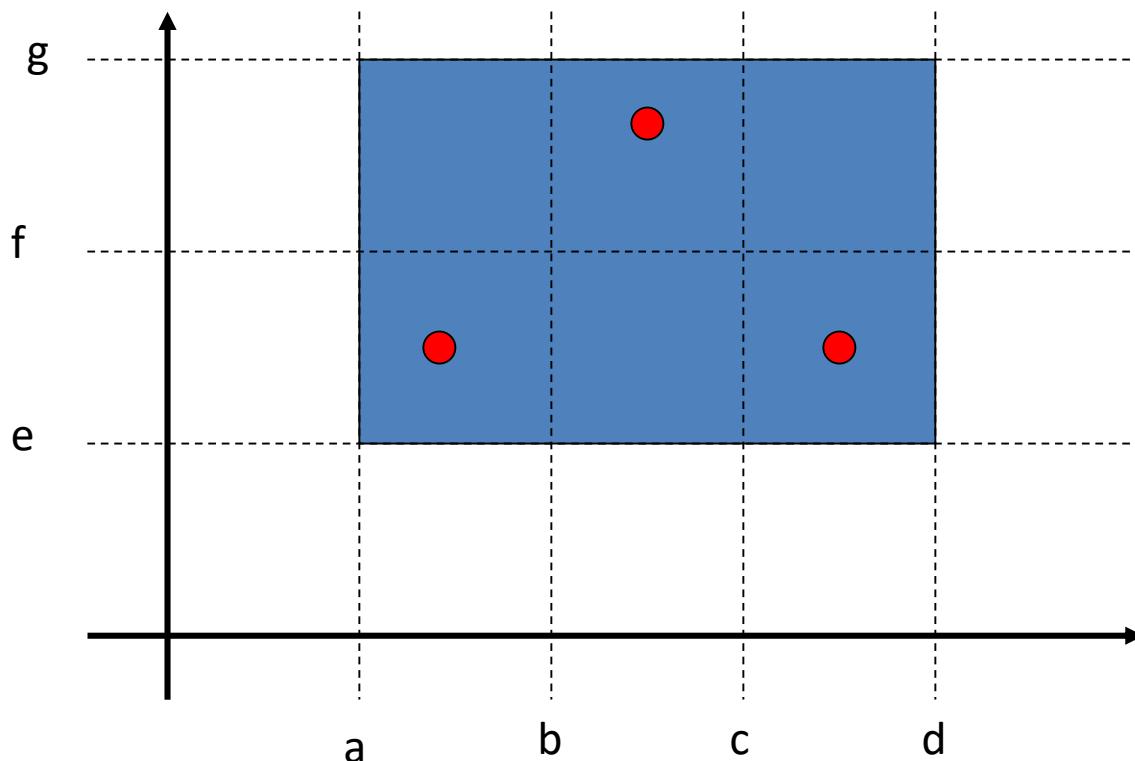
EC - Example

- 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
 - $[a, b), [b, c), [c, d] \leftarrow x_1$
 - $[e, f), [f, g] \leftarrow x_2$
- $[\rightarrow$ closed interval endpoint
- $(\rightarrow$ open interval endpoint
- $< >$ \rightarrow Ordered pair
- $() \rightarrow$ 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

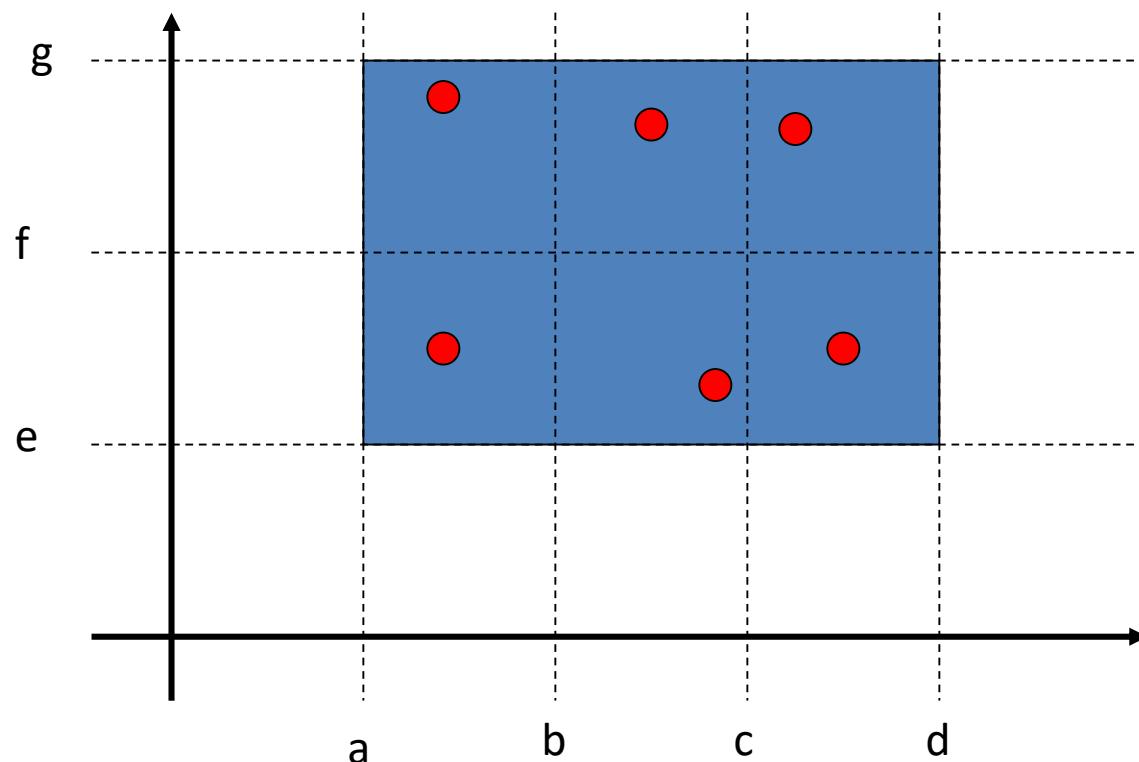
EC – Weak Normal



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

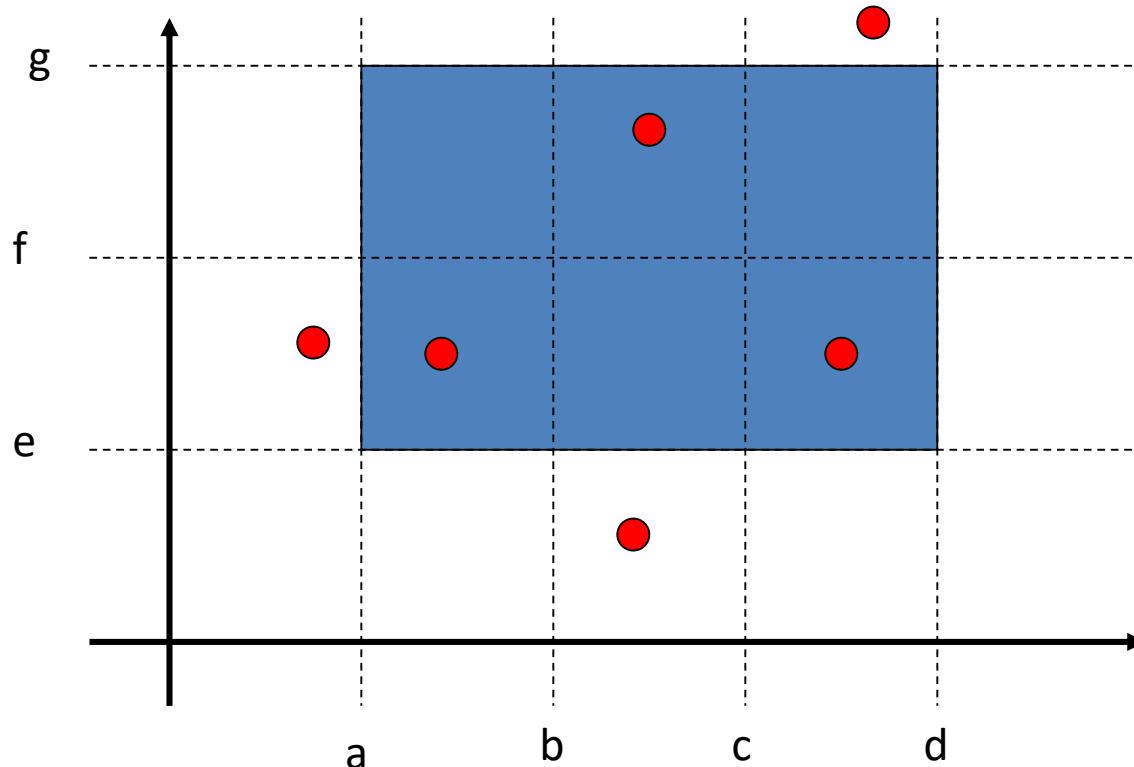
EC – Strong Normal



EC – Weak Robust

- Weak<>Robust is counterintuitive.
- 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

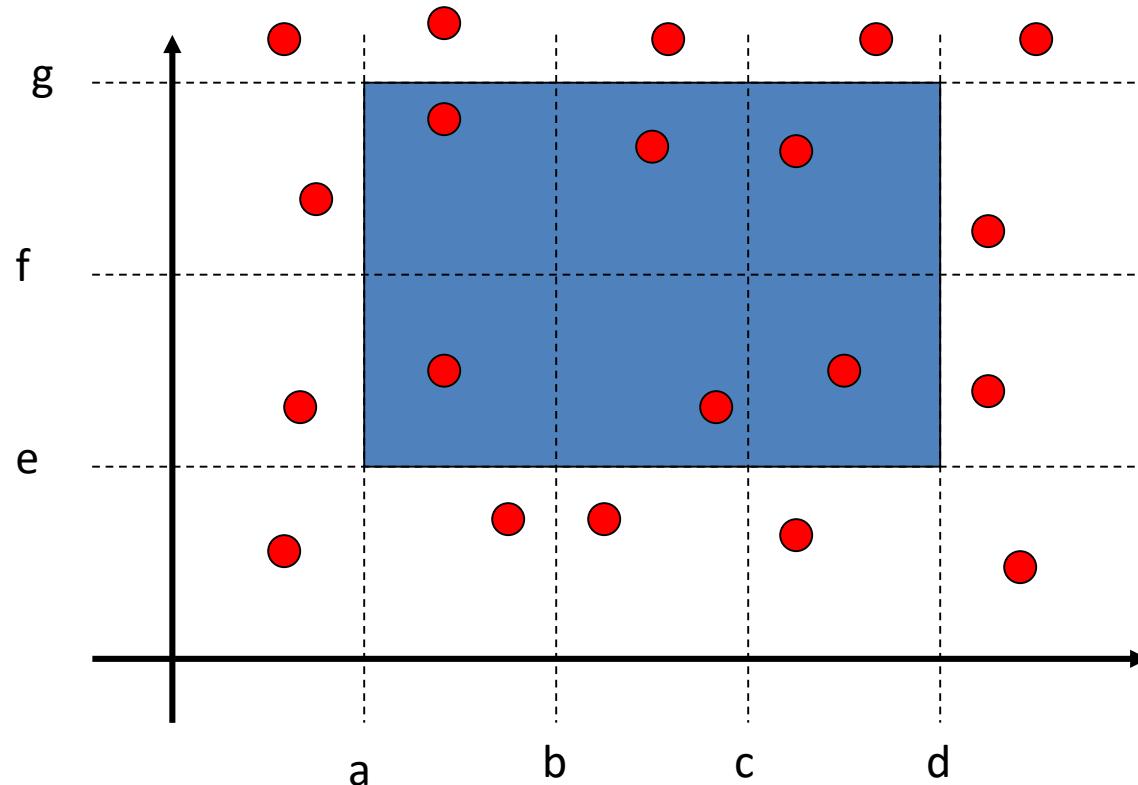
EC – Weak Robust



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



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

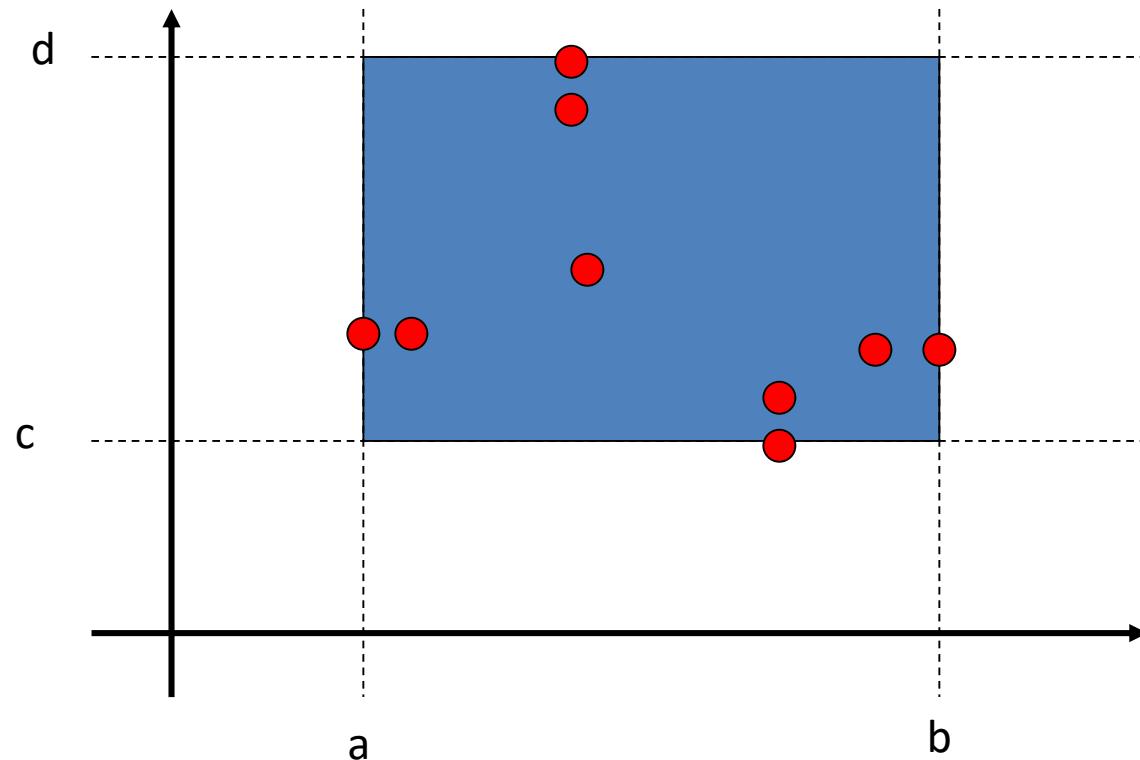
BVA – Explore the types

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
 - $[a, b] \leftarrow x_1$
 - $[c, d] \leftarrow x_2$

BVA

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



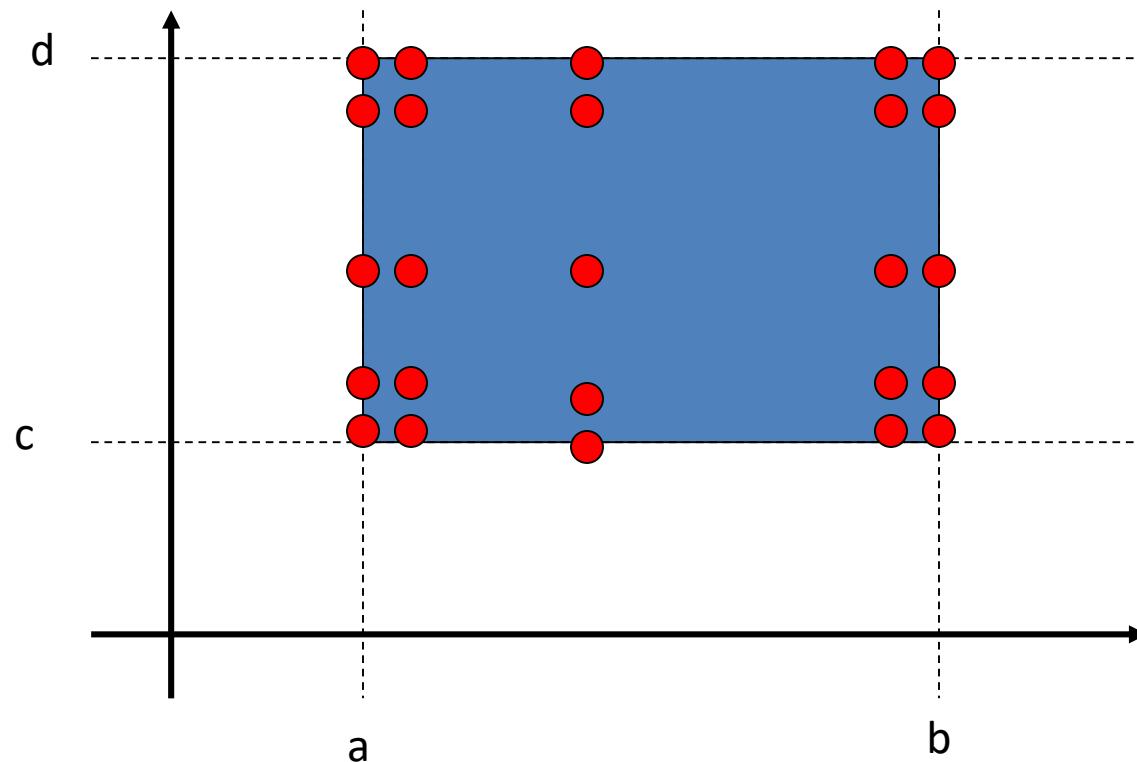
Generalising BVA

- **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

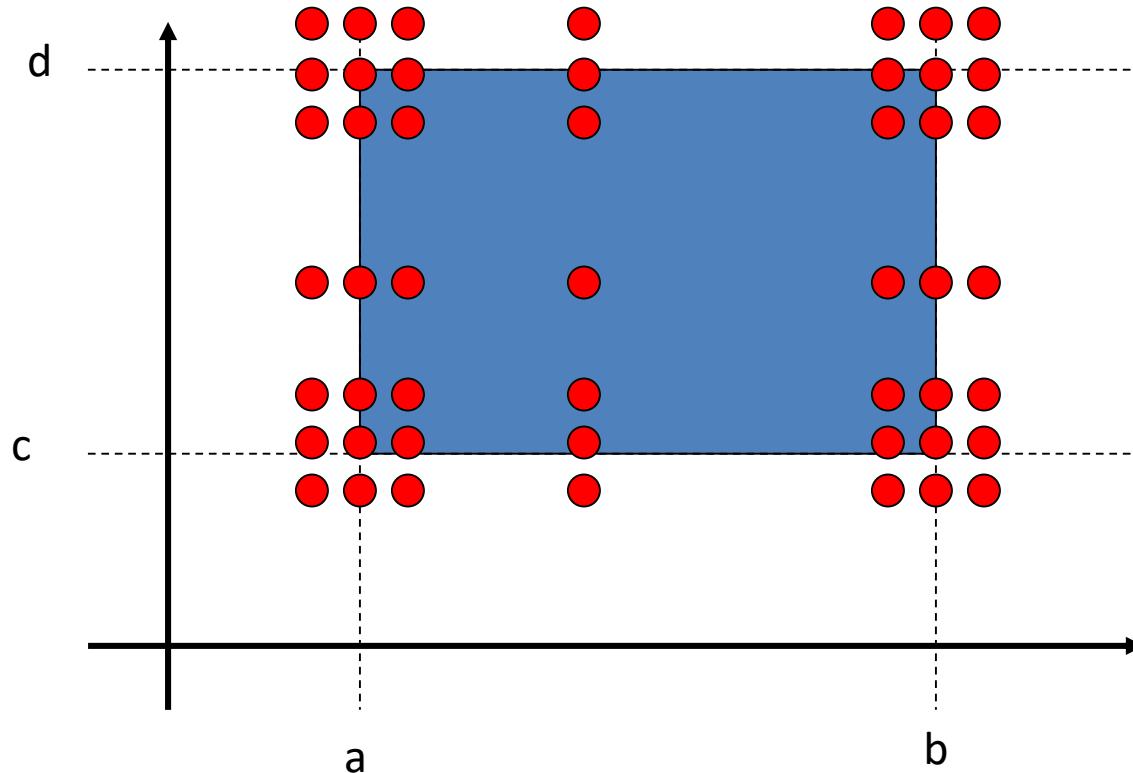
BVA Limitations

- 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

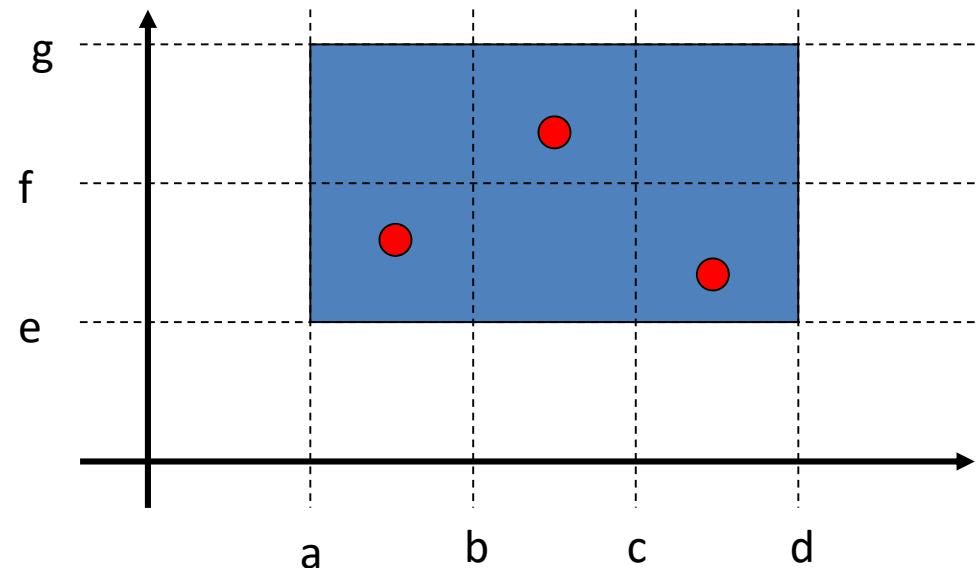


BVA – Special Value Testing

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

Edge Testing

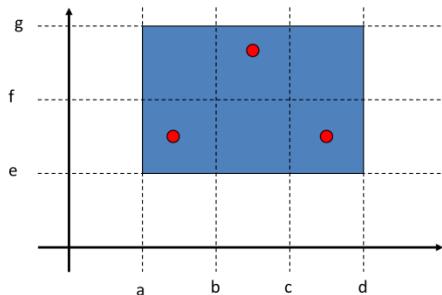
- ISTQB Advanced Level Syllabus (ISTQB, 2012) describes a hybrid of BVA and EC
- Edge Testing
- Faults near the boundaries of the classes
- Normal & Robust for Edge Testing



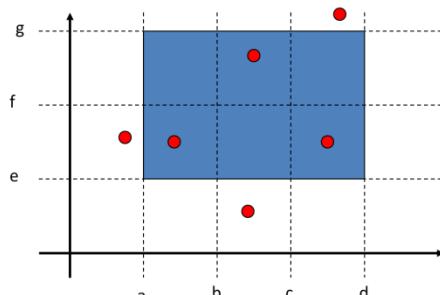


Topic 3.4: Examples & Case Study

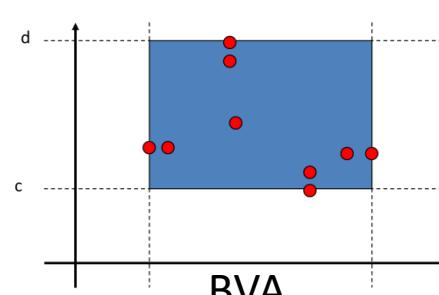
EC & BVA



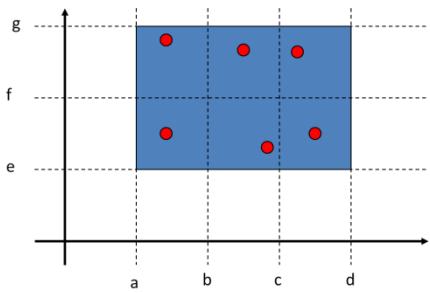
Weak Normal



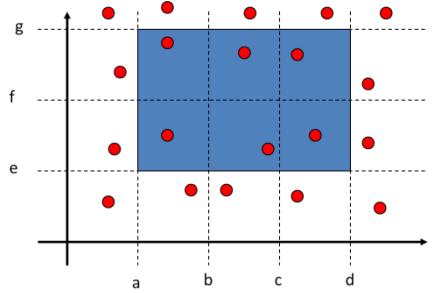
Weak Robust



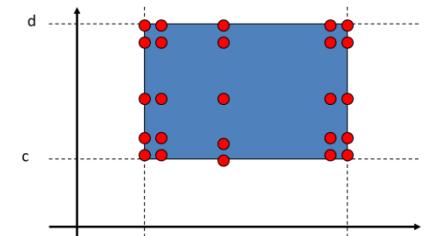
BVA



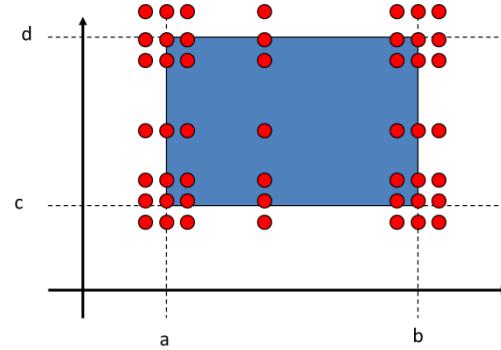
Strong Normal



Strong Robust



Worst Case Analysis



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

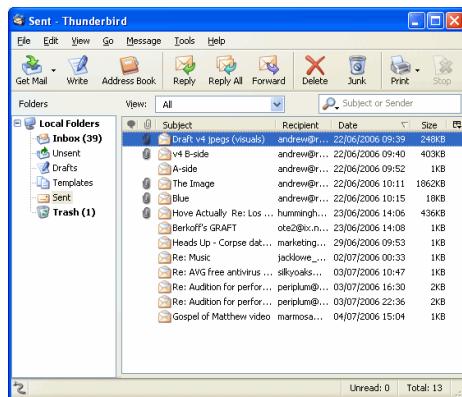
Problem Statement

- 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
- ...



Module 3 Self Study

- SS3.1 Write a program in your chosen language (C/C++/Java) to solve the triangle problem (along with variants). Make use of the EC and BVA techniques and test your program
- SS3.2 Take at least two systems or products of daily use. Write down the top functions of it. Analyse and reflect on use of EC and BVA for it. Compare and contrast the effectiveness of the techniques to test the System



SS 3.1 Triangle Program

3.1 Self Study

To explore:

- Use of EC and BVA for a program
- Analyse the results of test run and comment on the effectiveness of the techniques

Study Work:

- You are required to write the program in your chosen language along with the variants (Which side is hypotenuse and area of the triangle). Compile and run the program
- Create the test cases for the program using EC and BVA
- Test your program using the test cases
- Analyse the results of the test run and comment on the effectiveness of the techniques



SS3.2 Analyse effectiveness of EC and BVA for systems or products

1.2 Self Study

To explore:

- Effectiveness of use of EC and BVA for two or more systems
- List the limitations that you come across

Study Work:

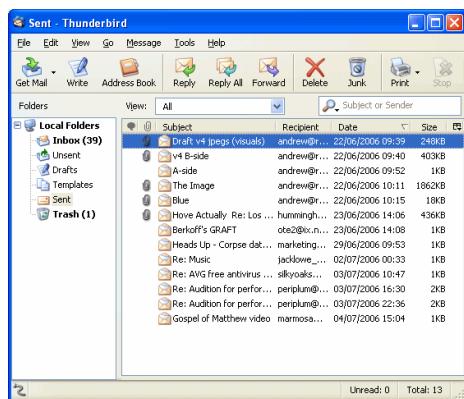
- Take at least two systems or products of daily use. Write down the top functions of it
- Analyse and reflect on use of EC and BVA for it
- Compare and contrast the effectiveness of the techniques to test the System

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!

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
-

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

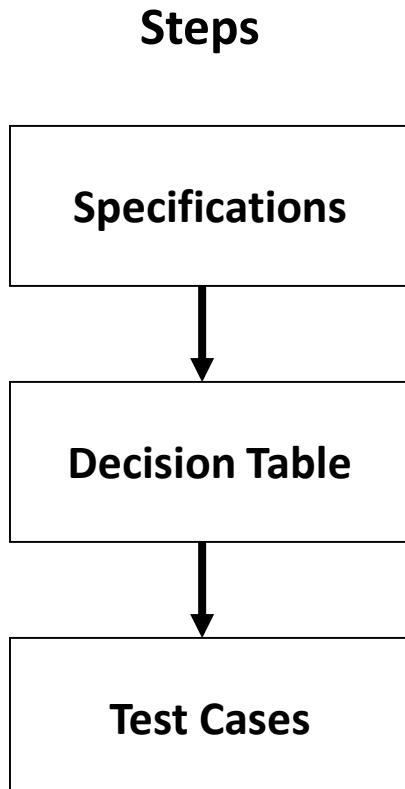
- Program P
- Input variables: X and Y
- X can take one value from {a, b, c}
- Y can take one value from {d, e, f}
- Leads to:
- 9 factor combinations (3^2)

- For large number of variables and values of each variable these combinations can be very large
- If we image 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

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 → 2^N rules
- Actions can be defined by user

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

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

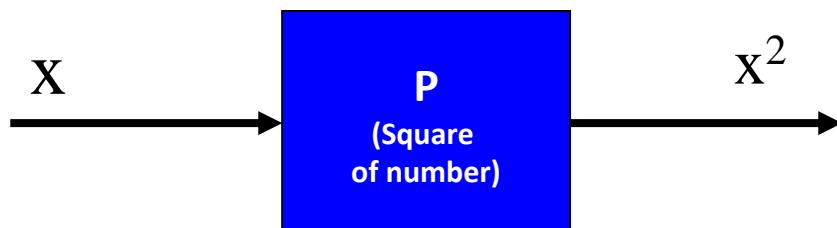
Examples & Case Study



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 ($x*x$)
- 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



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 image one test case per combination; we will have a huge number of test cases

Model the input

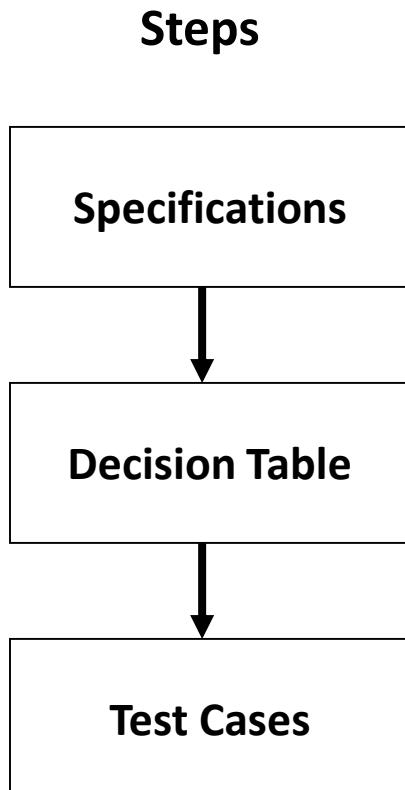
- Fault Model (Interaction Faults)
 - Two or more variables play a role
 - One or more specific values play a role
- Unique combinations
 - Latin Squares
 - Pairwise Testing
 - Orthogonal Array

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



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 → 2^N rules
- Actions can be defined by user

DT - Example

Component Specification: Input of 2 characters such that

1. The 1st character must be A or B.
2. The 2nd character must be a digit.
3. If the 1st character is A or B and the 2nd character is a digit the file is updated.
4. If the 1st character is incorrect, message X12 is displayed.
5. If the 2nd 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.



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

Module 4 Self Study

- SS4.1 Elements of being systematic
- SS4.2 Take at least two systems or products of daily use. Write down the significant functions of it. Analyse and reflect on use of Combinatorial & Decision Table Test Techniques for it. Compare and contrast the effectiveness of the techniques to test the System



SS 4.1 Elements of a test engineer

4.1 Self Study

To explore:

- Elements of being systematic – recipe for success for a test engineer

Study Work:

- Work with your colleagues and fellow students to find out the elements which will make an engineer systematic!
- Review with fellow students what you came up with
- Compare and contrast with what exists today



SS4.2 Analyse effectiveness of Combinatorial & DT for systems or products

4.2 Self Study

To explore:

- Effectiveness of use of Combinatorial & DT for two or more systems
- List the limitations that you come across

Study Work:

- Take at least two systems or products of daily use. Write down the top functions of it
- Analyse and reflect on use of Combinatorial & DT for it
- Compare and contrast the effectiveness of the techniques to test the System

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

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

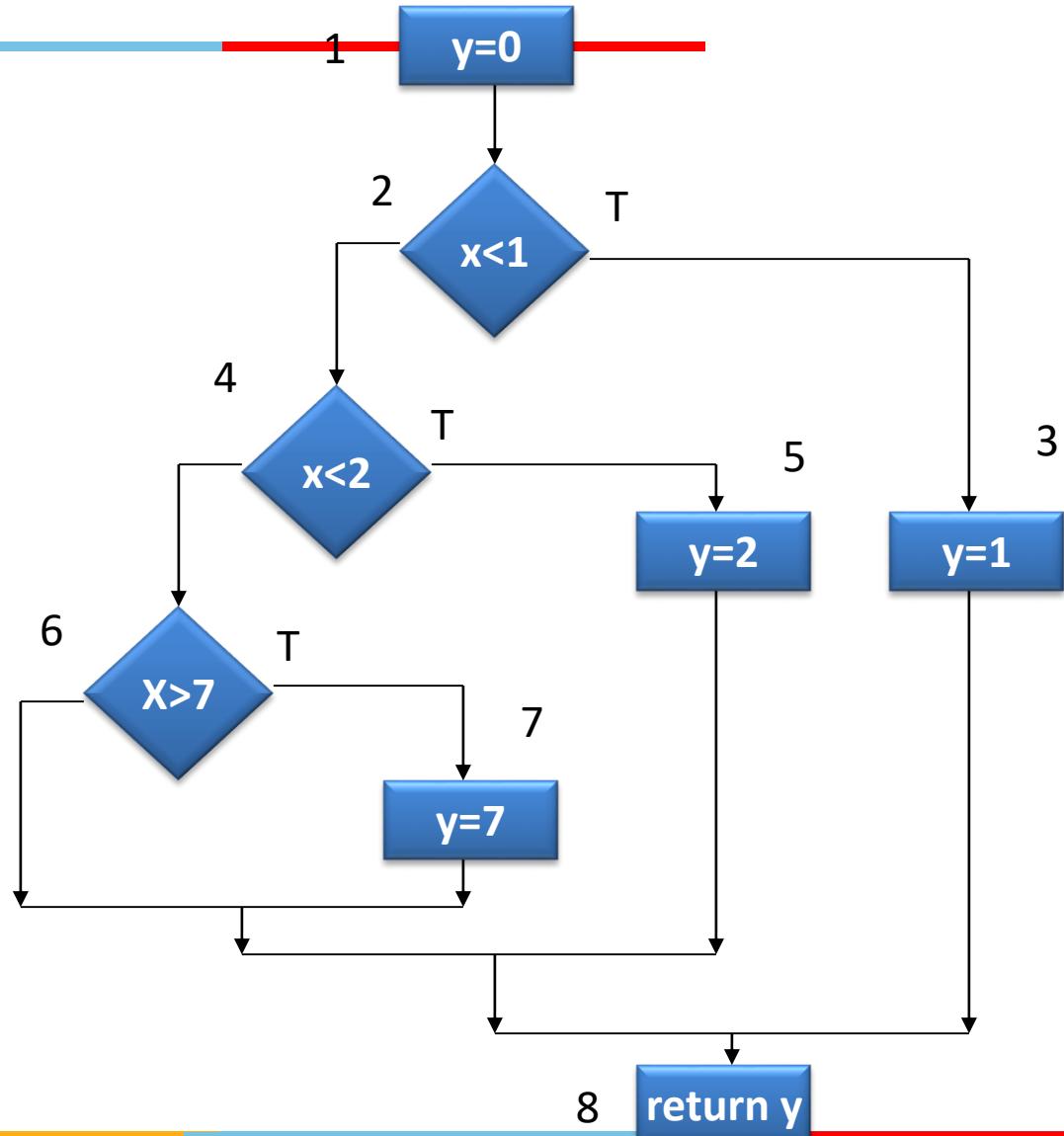
1, 2, 3, 8

Test #2:x=1

1, 2, 4, 5, 8

Test #3:x=10

1, 2, 4, 6, 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

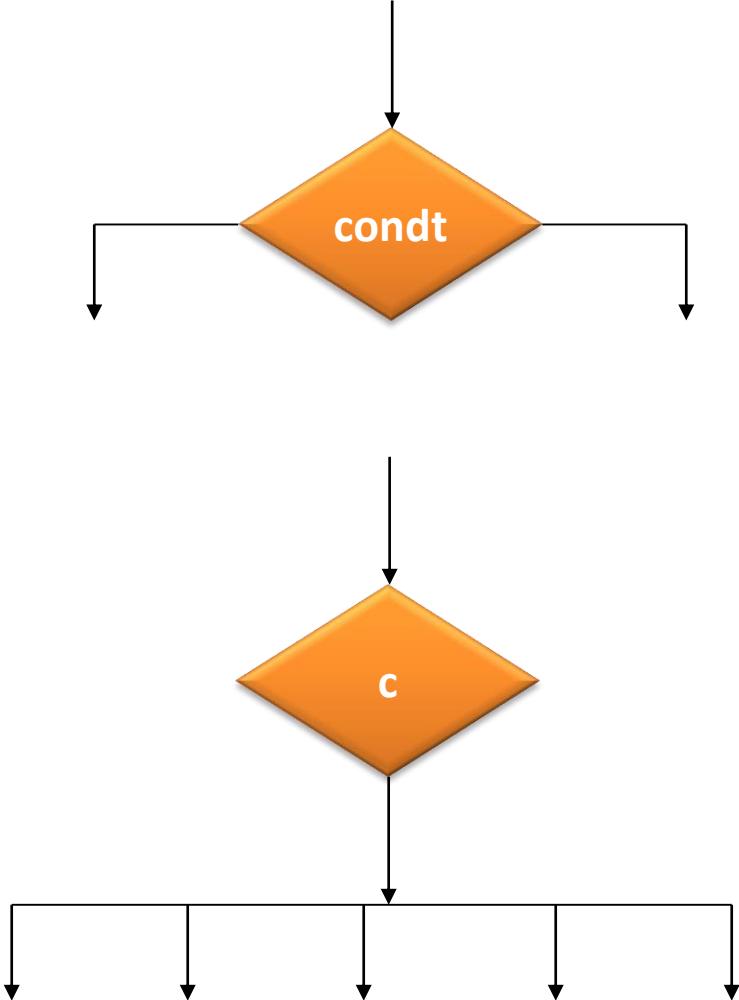
SWITCH

 SWITCH-CASE

Salient Features

 More demanding

 When branch testing is satisfied
 the statement testing is also
 satisfied



Multiple Condition Testing

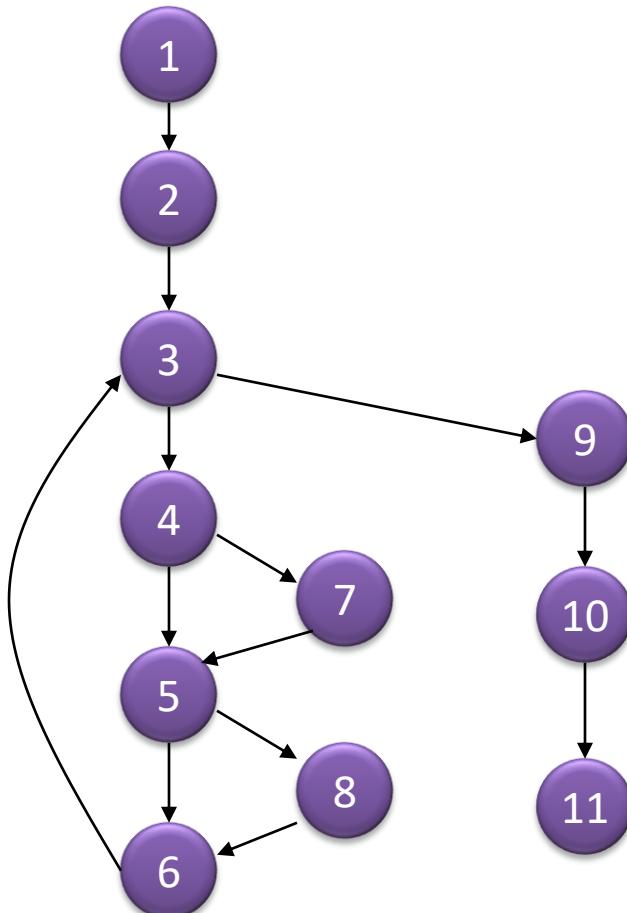
- This is testing of condition with complex predicates (OR, NOT and AND)
- IF C1 THEN, ELSE → Branch testing ~ multiple condition
- IF (C1 AND C2 AND C3) THEN, ELSE → Multiple condition

- 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”

Control Flow Graph

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

Loop Testing



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

```

mathtable (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

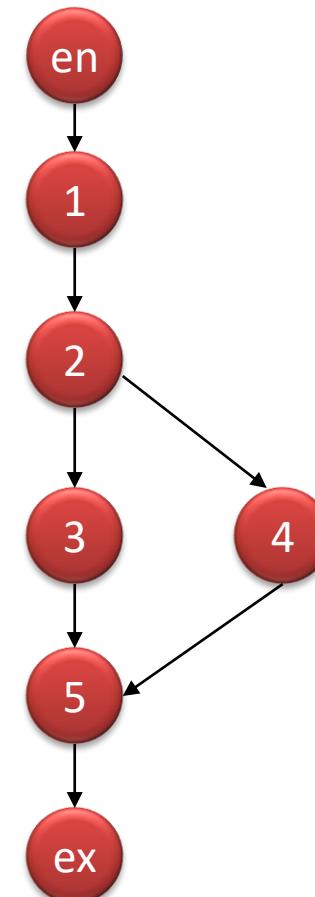
Basic Concept

- 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

of branches = 1

Paths

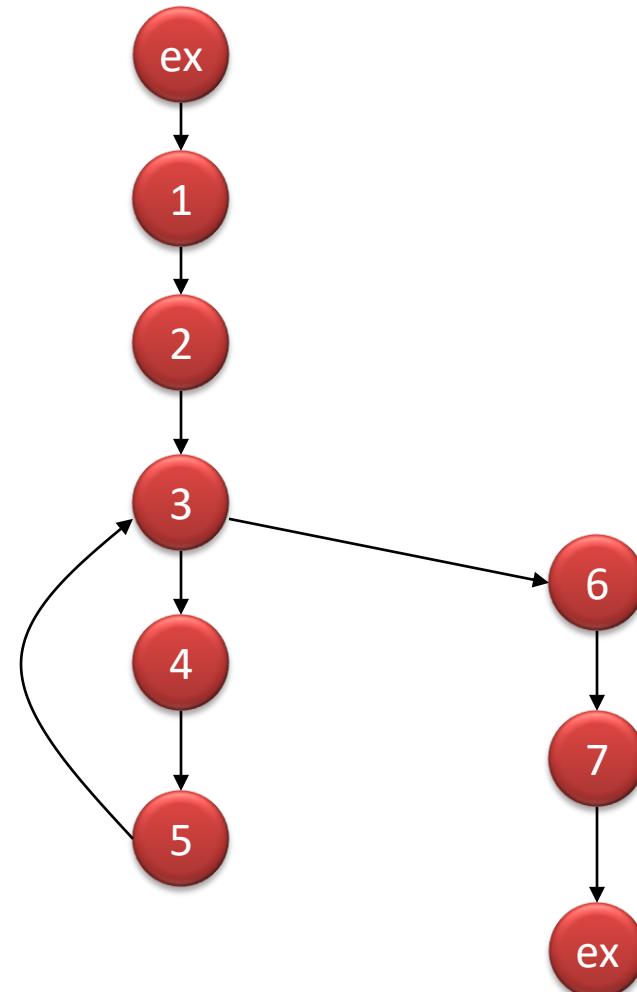
P1: 1 2 3 6 7

P2: 1 2 3 4 5 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

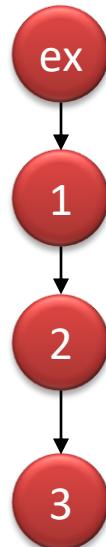
McCabe's cyclomatic number (end 70's)

$$v = e - n + 2$$

e: # of edges in a control graph

n: # of nodes in a control graph

Examples:



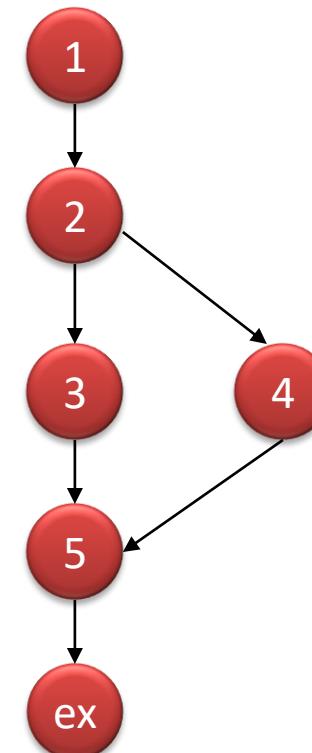
$$e = 3$$

$$n = 4$$

$$v = e - n + 2$$

$$= 3 - 4 + 2$$

$$= 1$$



$$e = 6$$

$$n = 6$$

$$v = e - n + 2$$

$$= 6 - 6 + 2$$

$$= 2$$

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

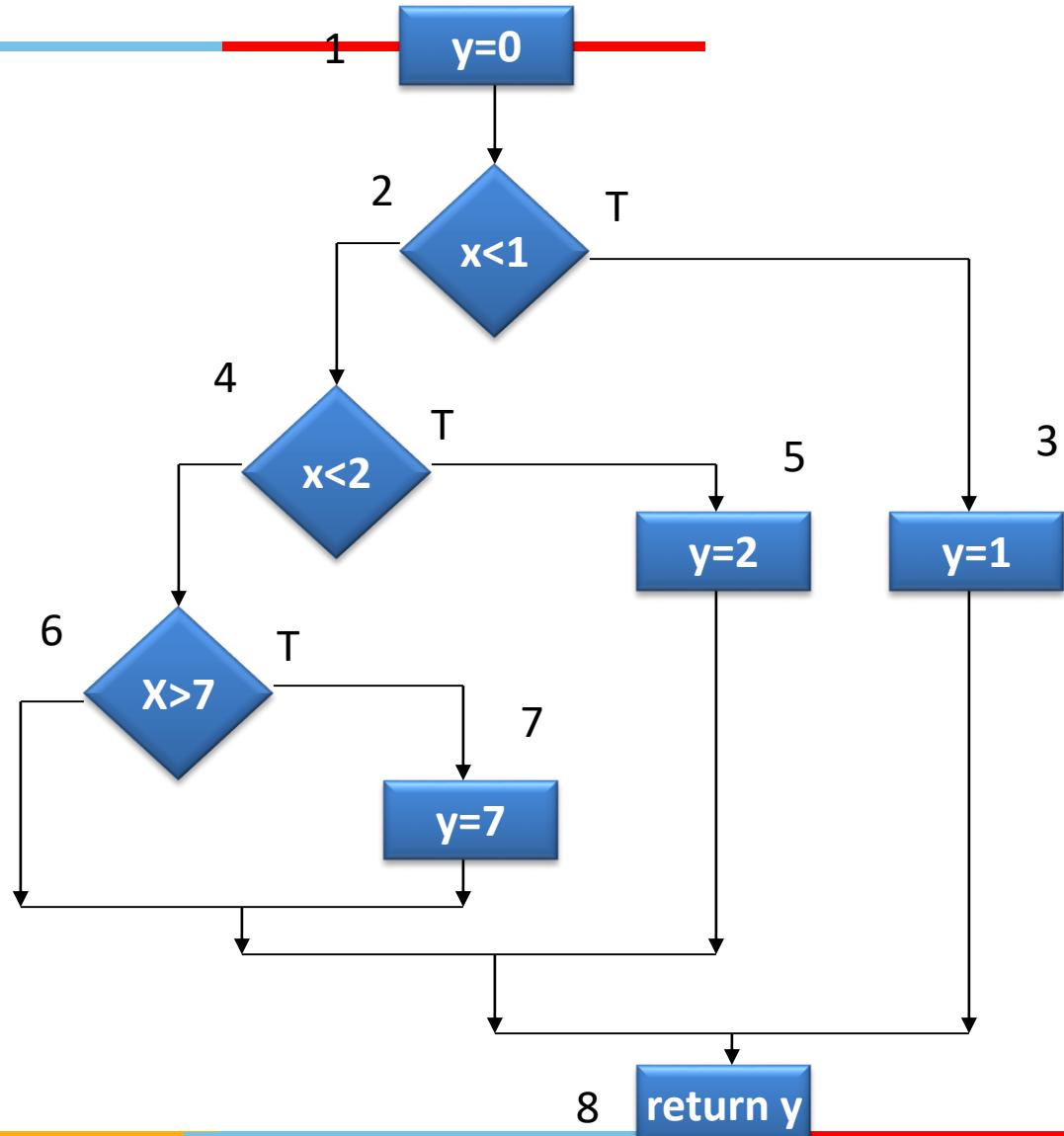
1, 2, 3, 8

Test #2:x=1

1, 2, 4, 5, 8

Test #3:x=10

1, 2, 4, 6, 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

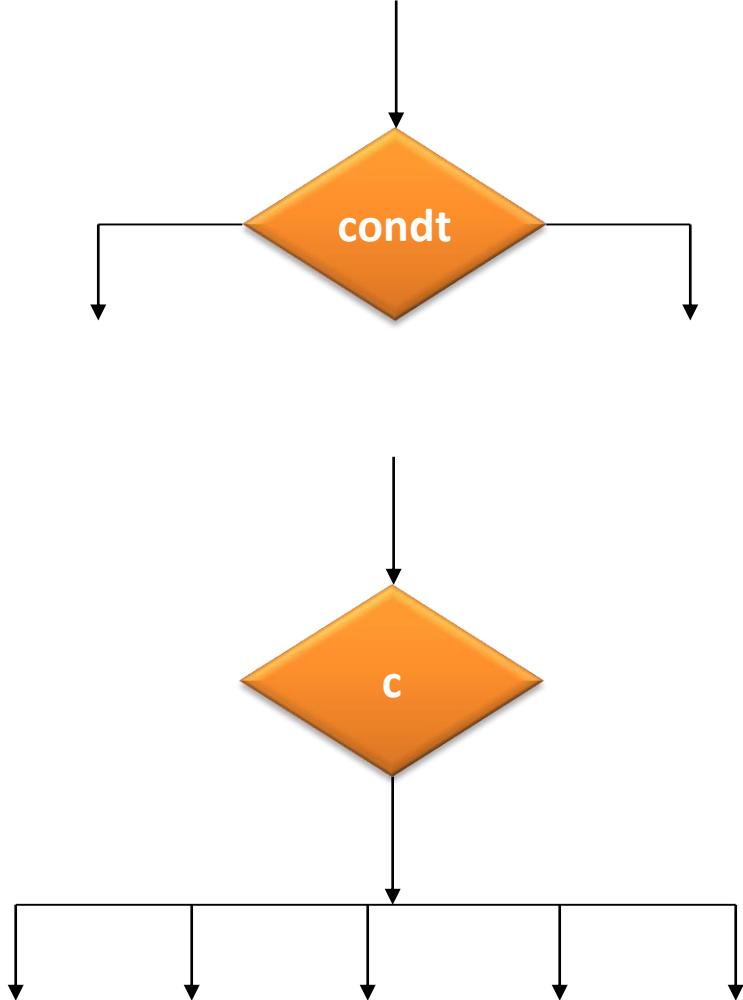
SWITCH

 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

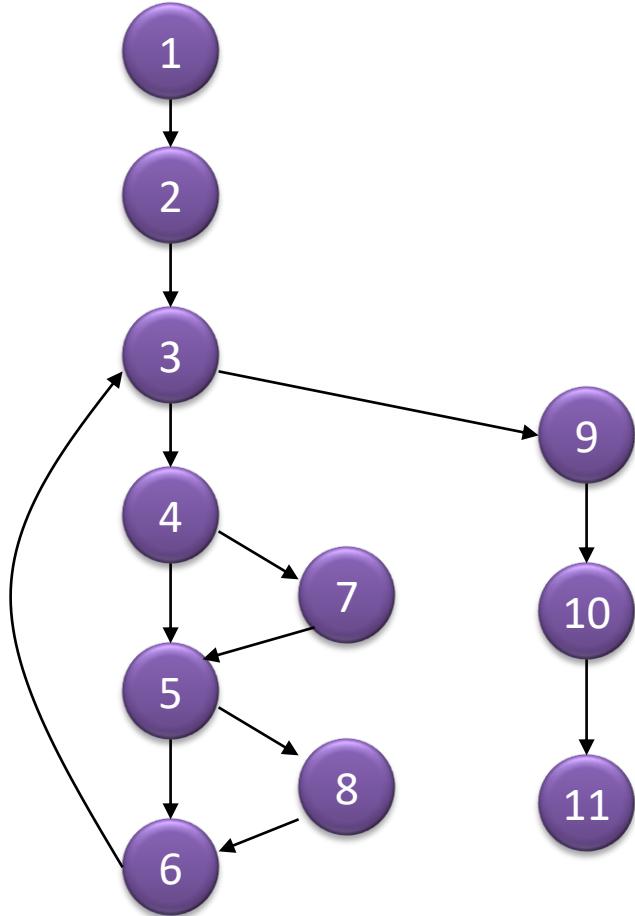


Topic 5.2: Path Testing

Control Flow Graph

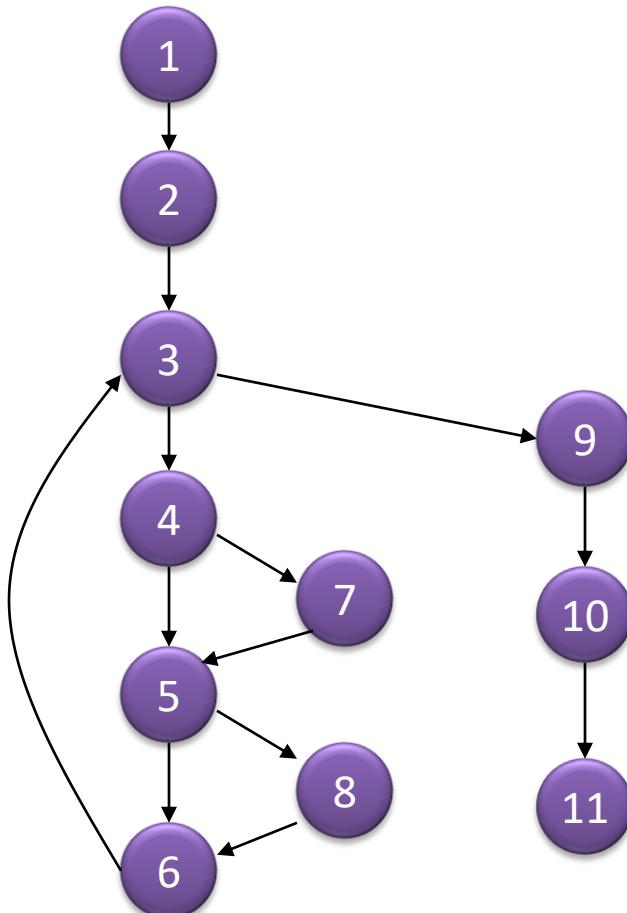
- A control Flow Graph consists of Nodes and Edges. Edges are between nodes and are directed
- A Node: A statement (i.e. executable atomic entity in a program)
 - An assignment statement
 - An input/output statement
 - Predicate of a condition
- An Edge: An edge represents a flow of control between two 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

```

mathtable (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

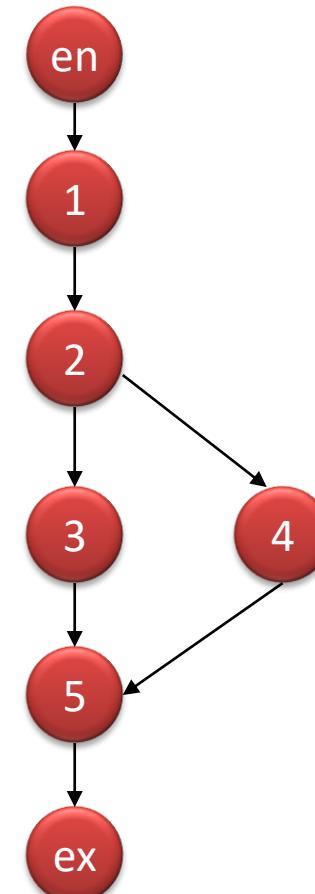
Basic Concept

- 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

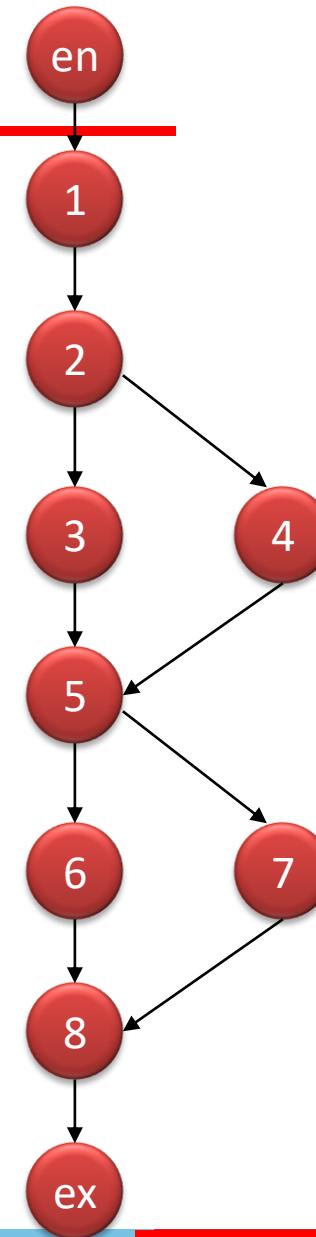
```
input (x)
if (x<10) then y=0
else y=1
if (x<30) then z=1
else z=2
output (y, z)
```

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

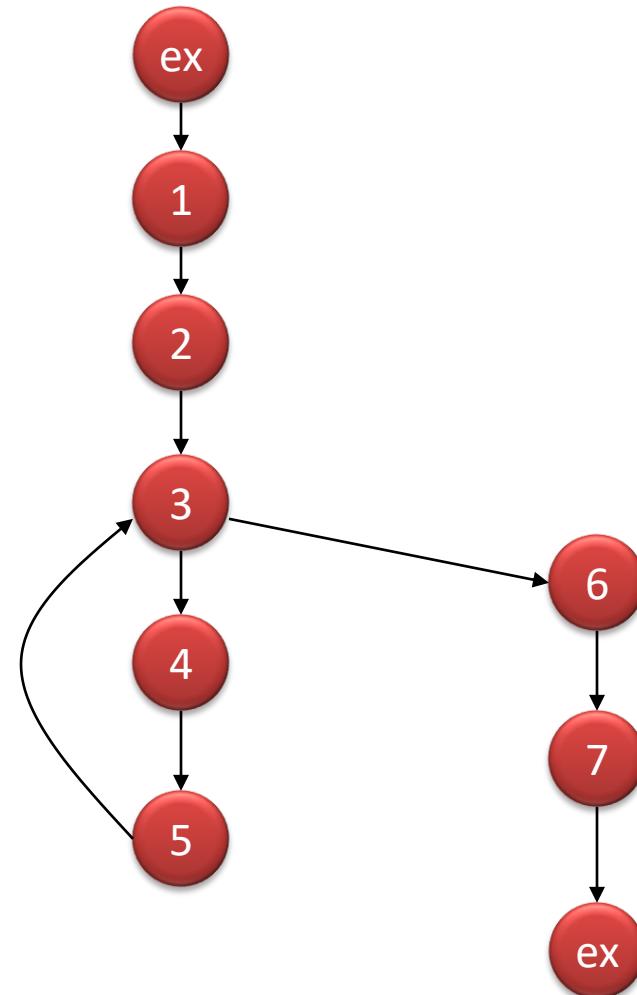
P1: 1 2 3 6 7

P2: 1 2 3 4 5 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

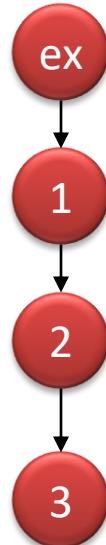
McCabe's cyclomatic number (end 70's)

$$v = e - n + 2$$

e: # of edges in a control graph

n: # of nodes in a control graph

Examples:



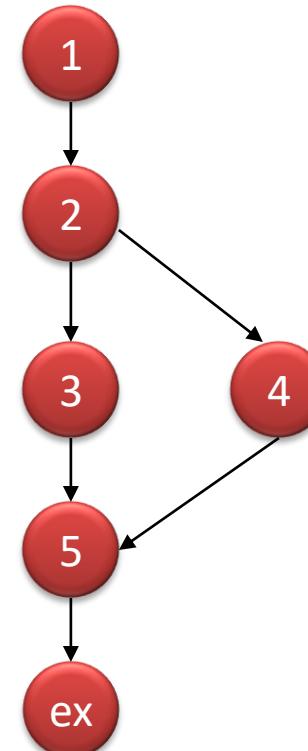
$$e = 3$$

$$n = 4$$

$$v = e - n + 2$$

$$= 3 - 4 + 2$$

$$= 1$$



$$e = 6$$

$$n = 6$$

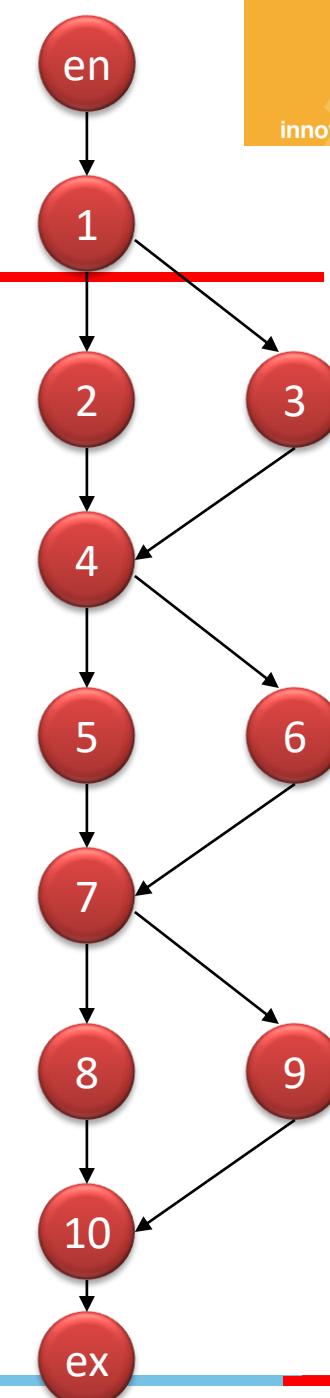
$$v = e - n + 2$$

$$= 6 - 6 + 2$$

$$= 2$$

McCabe Path

Number of Paths?



McCabe Path

$$e = 14$$

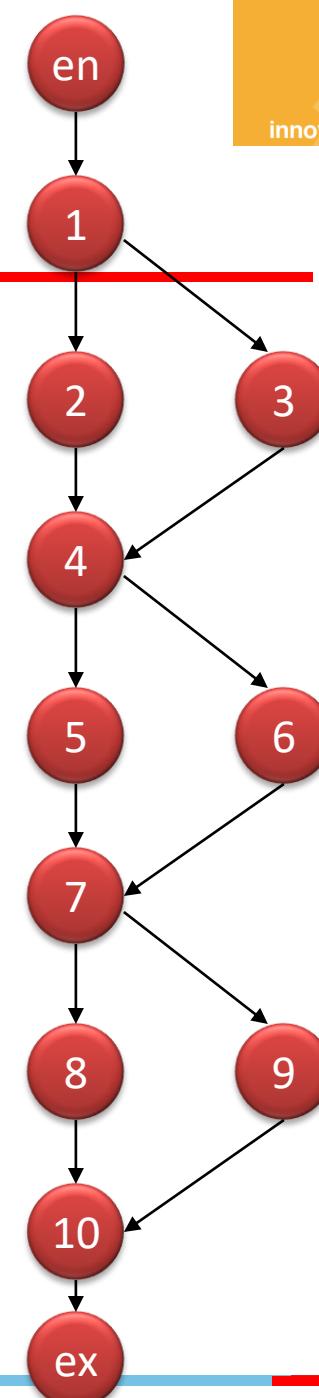
$$n = 12$$

$$v = e - n + 2$$

$$= 14 - 12 + 2$$

$$= 4$$

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

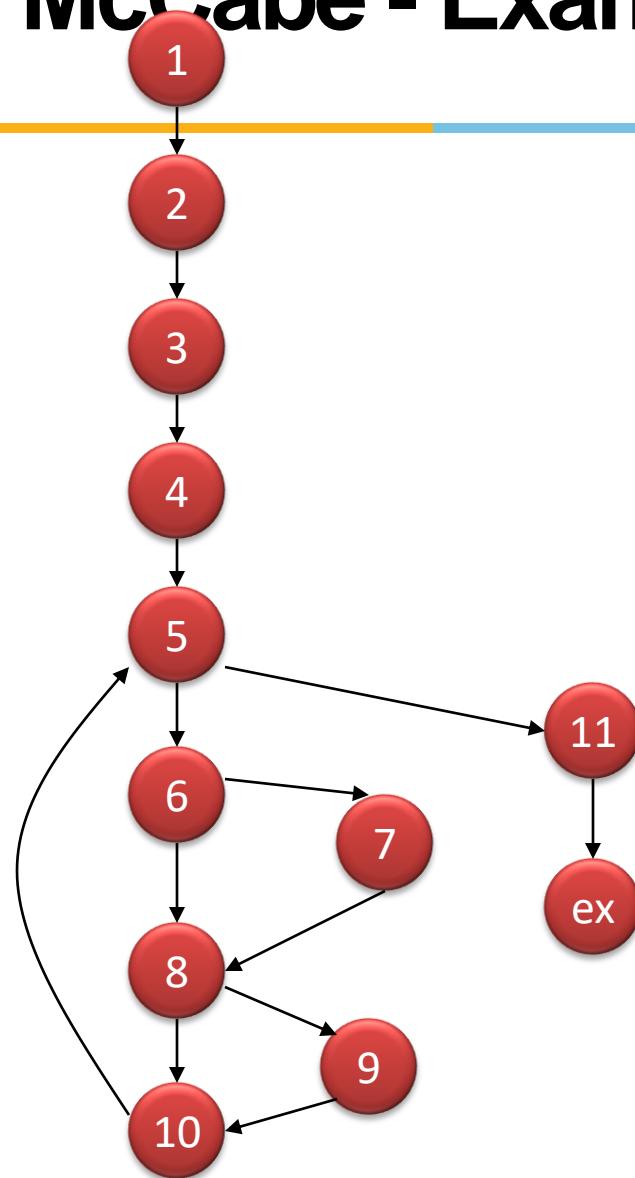


Topic 5.3: Examples

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



$$e = 15$$

$$n = 13$$

$$V = 15 - 13 + 2$$

$$= 4$$

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

$n = 2$ $a = ?$

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



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

Module 5 Self Study

- SS5.1 Choose a program or a function (about 25 lines) and design unit (code based) test cases for it. Discuss and review the effectiveness of the techniques
- SS5.2 Search on the internet for the tools which are useful in unit/code based testing and measurements (metrics like complexity)



SS 5.1 Code Based Test Cases

9.1 Self Study

To explore:

- Test techniques for Code Based Testing
- Apply techniques for Code Based Testing

Study Work:

- You are required to choose (or write) a program in your chosen language (C/C++/Java)
- Create the code based test cases for the program using the techniques learnt
- Test your program using the test cases
- Analyse the results of the test run and comment on the effectiveness of the techniques



SS 5.2 Code Based Testing tools and metrics

5.2 Self Study

To explore:

- Software tools for unit/code based testing
- List the limitations that you come across

Study Work:

- Search and research for tools and frameworks for unit/code based testing
- Compare and contrast the effectiveness of the techniques to test

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 $\text{PATHS}(P)$

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
- $v=v+1$;

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

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

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

- Node $n \in G(P)$ is a defining node of the variable $v \in V$ written as $DEF(v, n)$, iff the value of the variable v is defined at the statement fragment corresponding to node n
 - Input statements
 - Assignment statements
 - Loop control statements
 - Procedure calls

Definitions

- Node $n \in G(P)$ is a use node of the variable $v \in V$ written as $USE(v, n)$, iff the value of the variable v is used at the statement fragment corresponding to node n
 - Output statements
 - Assignment statements
 - Conditional statements
 - Loop control statements
 - Procedure calls

Definitions

- A usage node $USE(v, n)$ is a predicate use (denoted as P-use), iff the statement n is a predicate statement; otherwise, $USE(v, n)$ is a computation use, (denoted C-use) value of the variable v is used at the statement fragment corresponding to node n
- Predicate use $outdegree \geq 2$
- Computation use $outdegree \leq 1$

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

Definitions

- A definition use path WRT a variable v (denoted du-path) is a path in $\text{PATHS}(P)$ such that, for some $v \in V$, there are defined usage nodes $\text{DEF}(v, m)$ and $\text{USE}(v, n)$ such that m and n are the initial and final nodes in the path

Definitions

- A definition-clear path WRT a variable v (denoted dc-path) is definition-use path in $\text{PATHS}(P)$ with initial and final nodes $\text{DEF}(v, m)$ and $\text{USE}(v, n)$ such that no other node in the path is a defining node of v

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

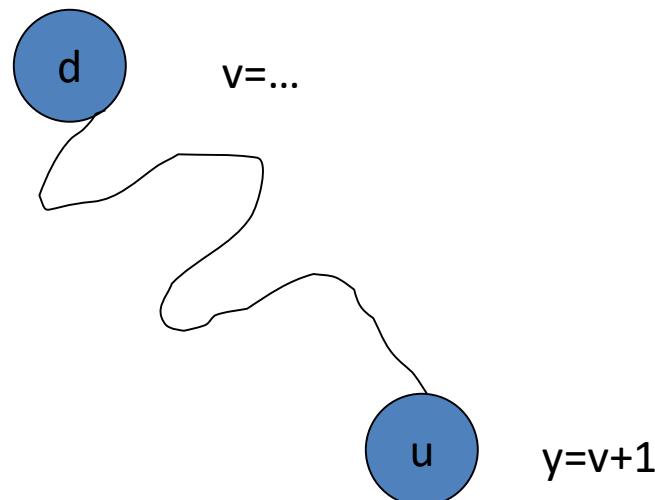
Use: An use is a statement that uses (references) a value of a variable

- $z=v+1$; 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
 $1 \rightarrow 2$
 $1 \rightarrow 3$
 $4 \rightarrow 5$

Variable y
 $1 \rightarrow 3$

Variable z
 $2 \rightarrow 5$

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



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



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>

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)

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>

Example: Solution

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

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

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

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

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

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

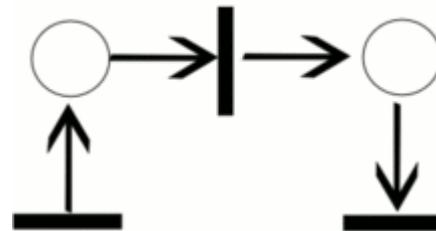
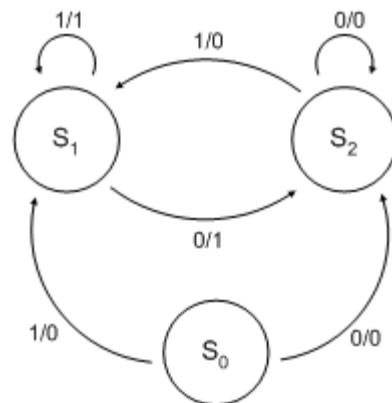
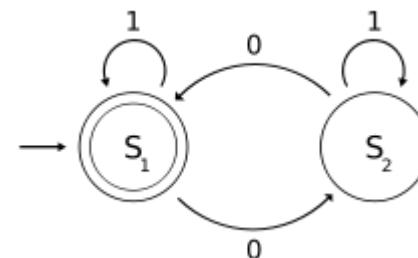
Complete other du pairs

Model Based Testing

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



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, StateCharts & 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>

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 Modelling Techniques



- State transition diagrams
- Extended Finite State Machines

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

Some examples to discuss

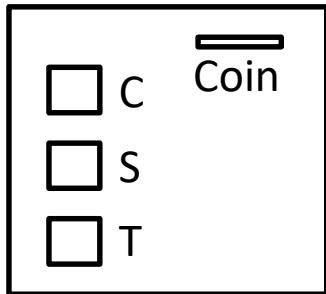
- Garage Door
- Building Lighting Control System
- Lift/Elevator Control System (One or Multiple)
- A MMI Interface of an instrument

Simple Vending Machine

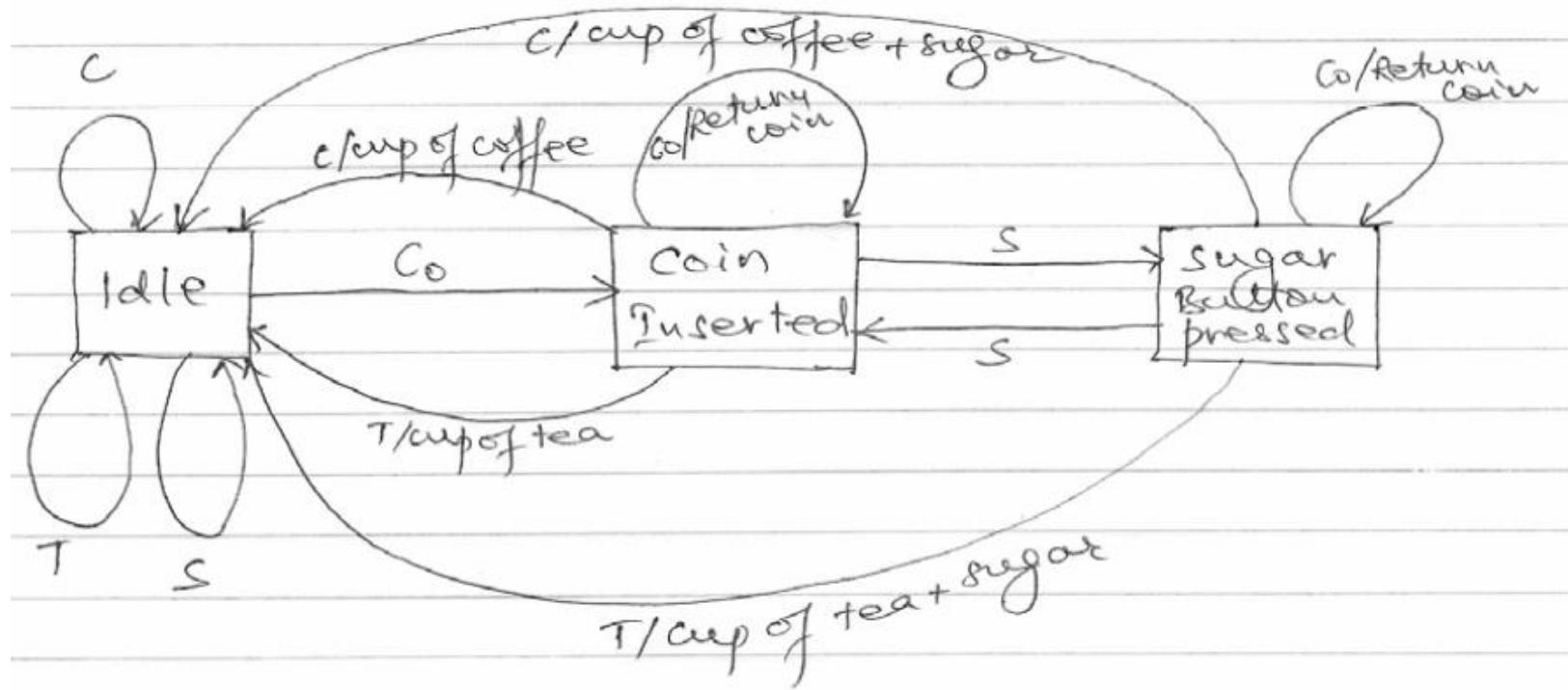


- Tea/Coffee vending Machine
- Options
 - Accepts token/coin
 - Sugar

Vending Machine



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



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



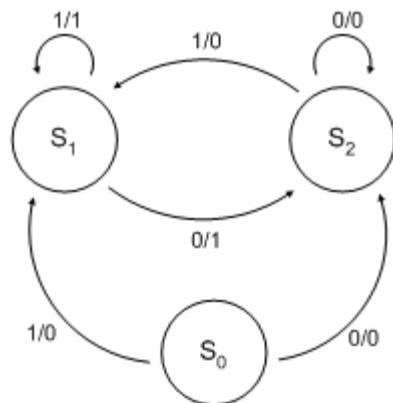
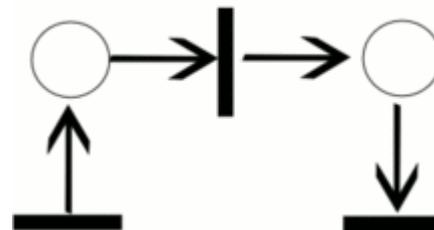
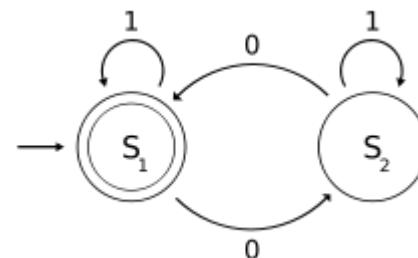
Topic 7.1: Model Based Testing – Introduction & Overview

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
- FMS/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, StateCharts & 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>



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

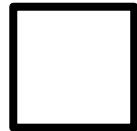
States	Values (of some data)
Empty State	top=0
Full State	top=10
Partial State	$1 \leq \text{top} \leq 9$

State Based Modeling Techniques



- State transition diagrams
- Extended Finite State Machines

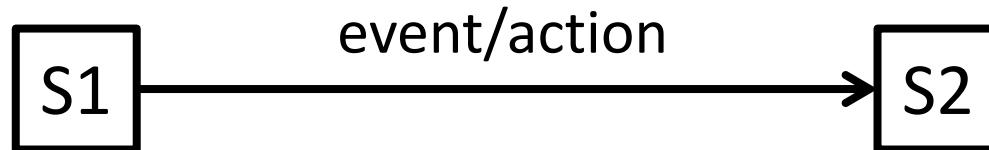
State Transition Diagram



A State



A transition



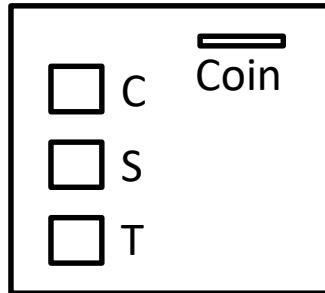
The Fault Model

- 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!*)

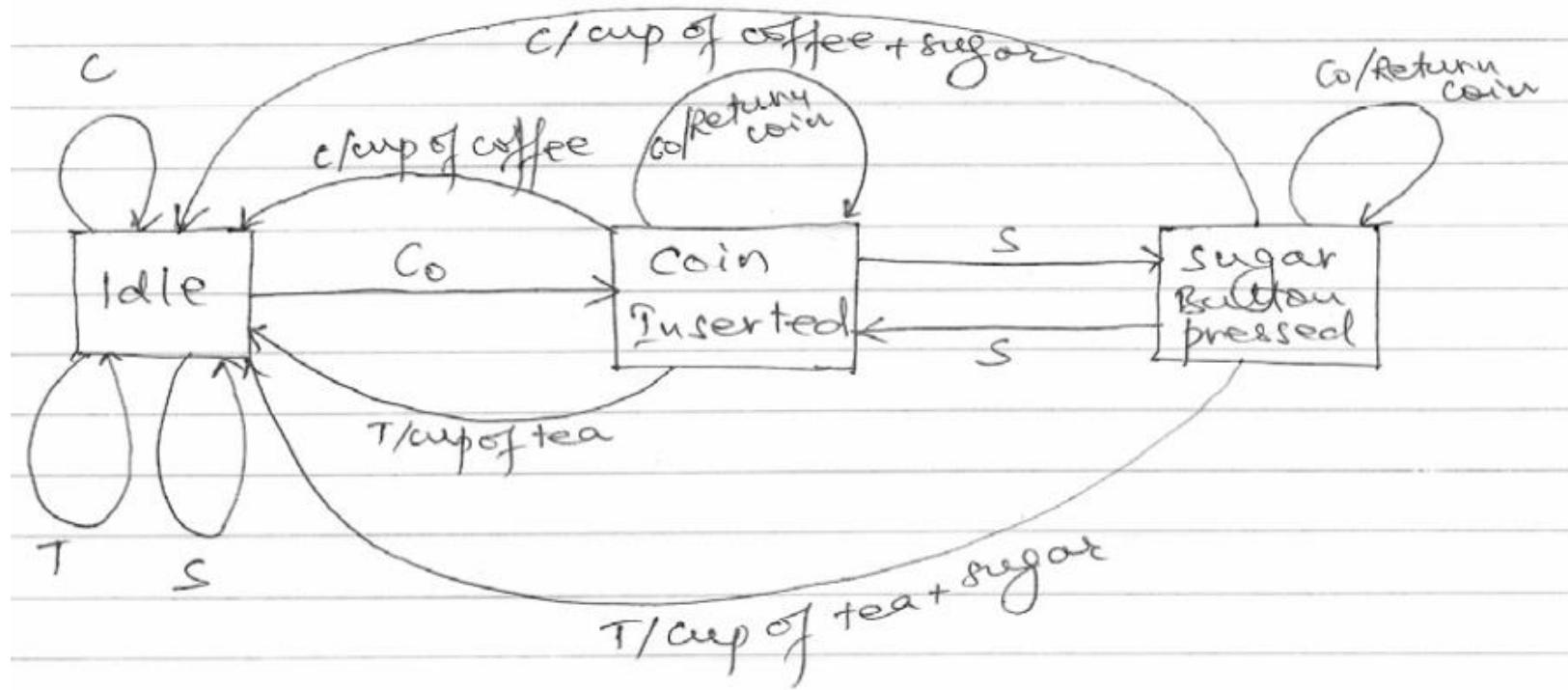
Fault Categories

- 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

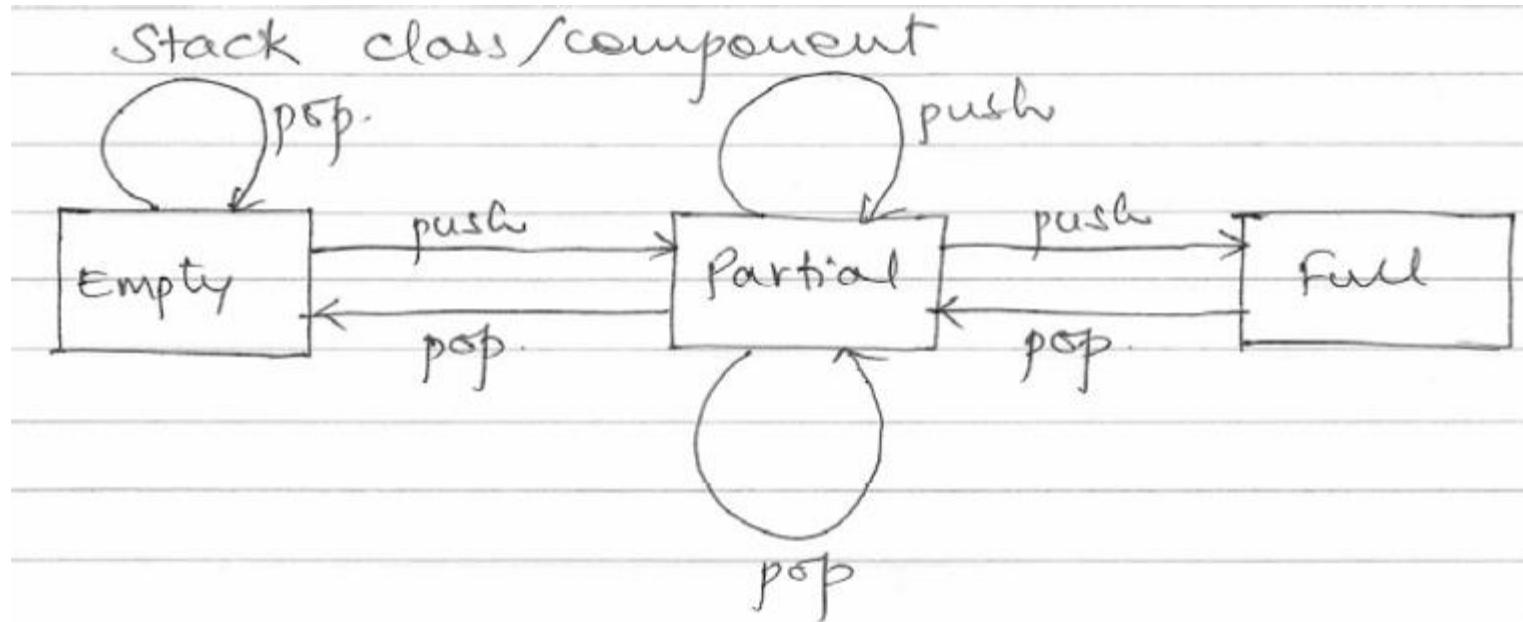
- Garage Door
- Building Lighting Control System
- Lift/Elevator Control System (One or Multiple)
- A MMI Interface of an instrument



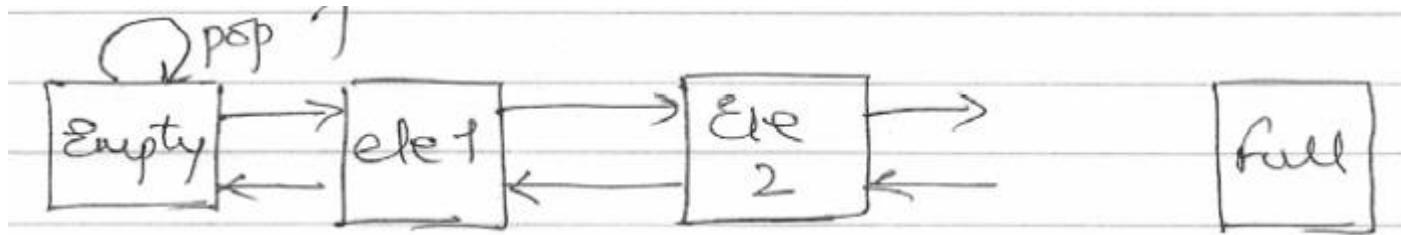
Topic 7.3: Examples

Stack

- Simple stack
- Operations (push and pop)



Notion of State Explosion



- Too many states
- State Explosion problem!

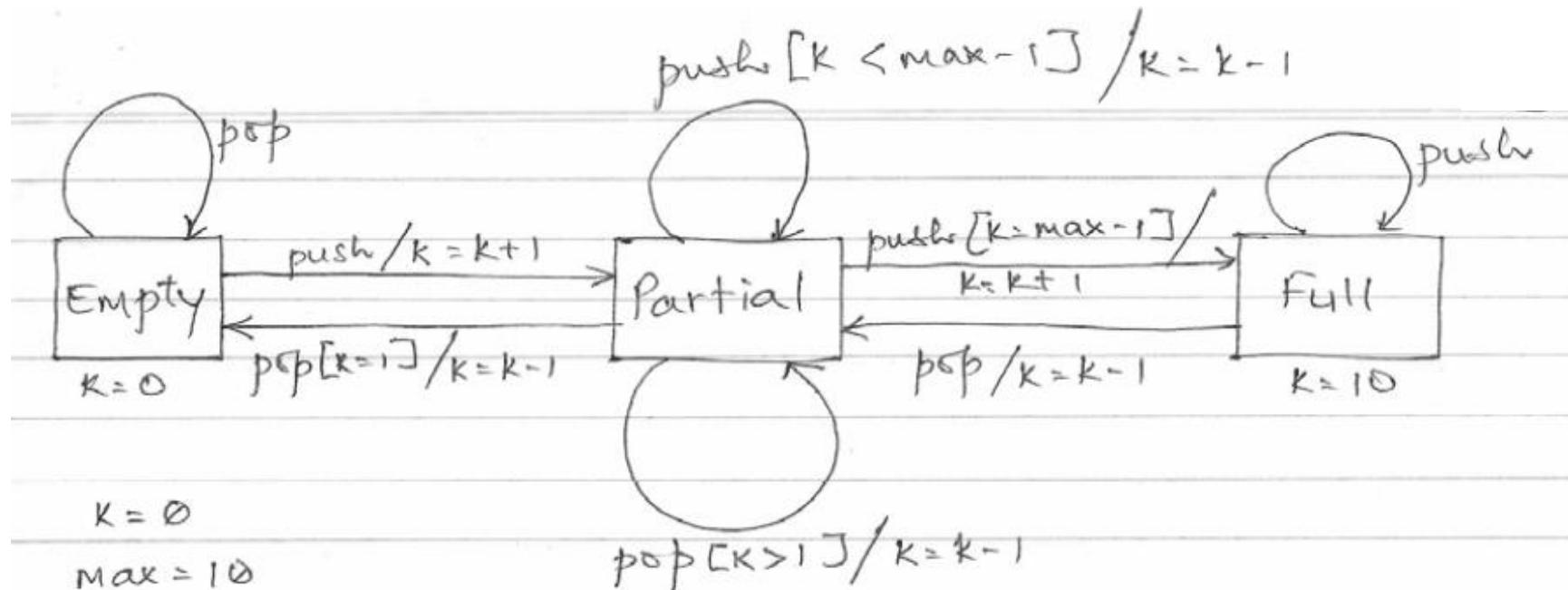
Extended FSM

- 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



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)
s.push(20) } 10 push operations
//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)

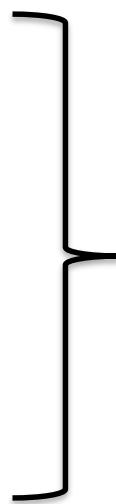
11th push

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()



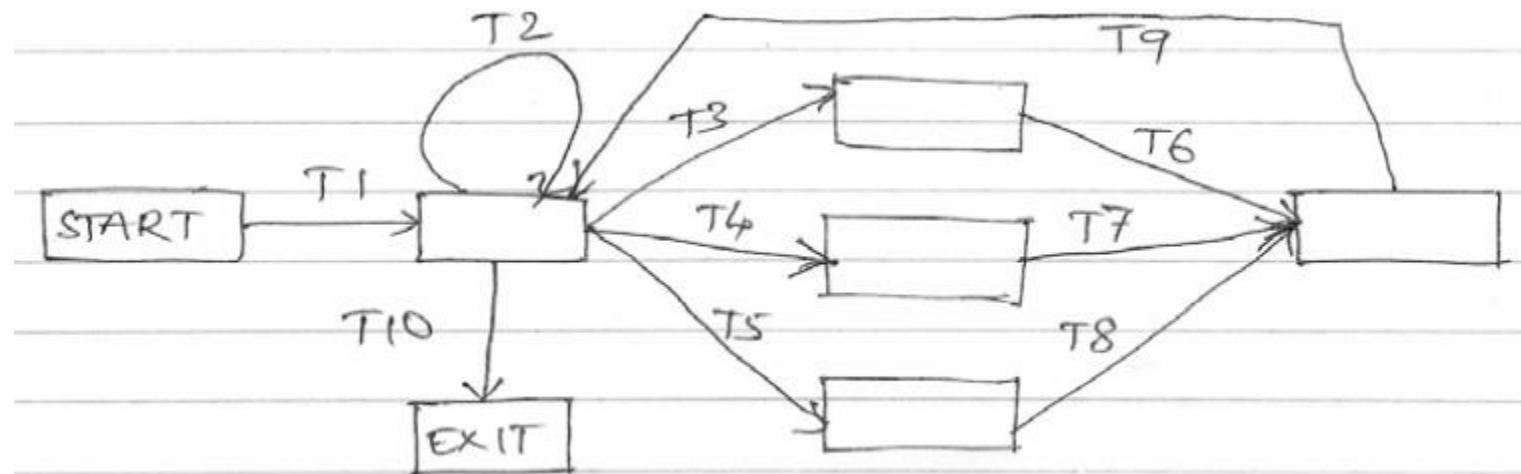
10 push

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

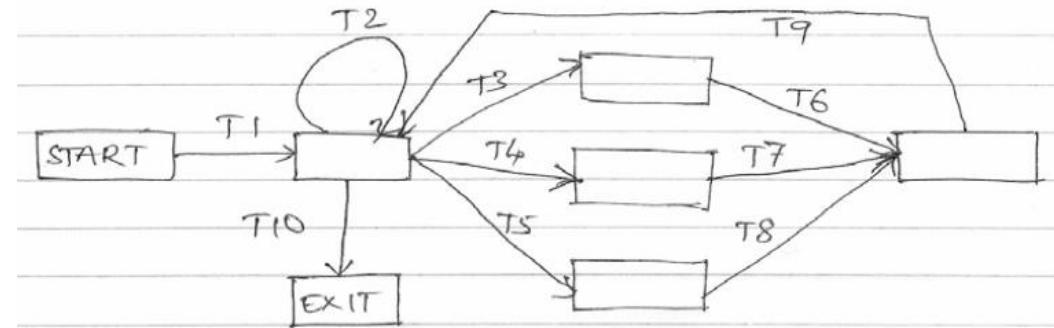
T7, T1, T2, T4, T7, T9, T10

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



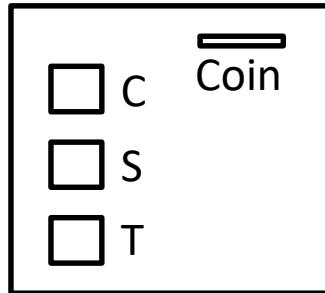
Topic 7.4: Case Study

Simple Vending Machine

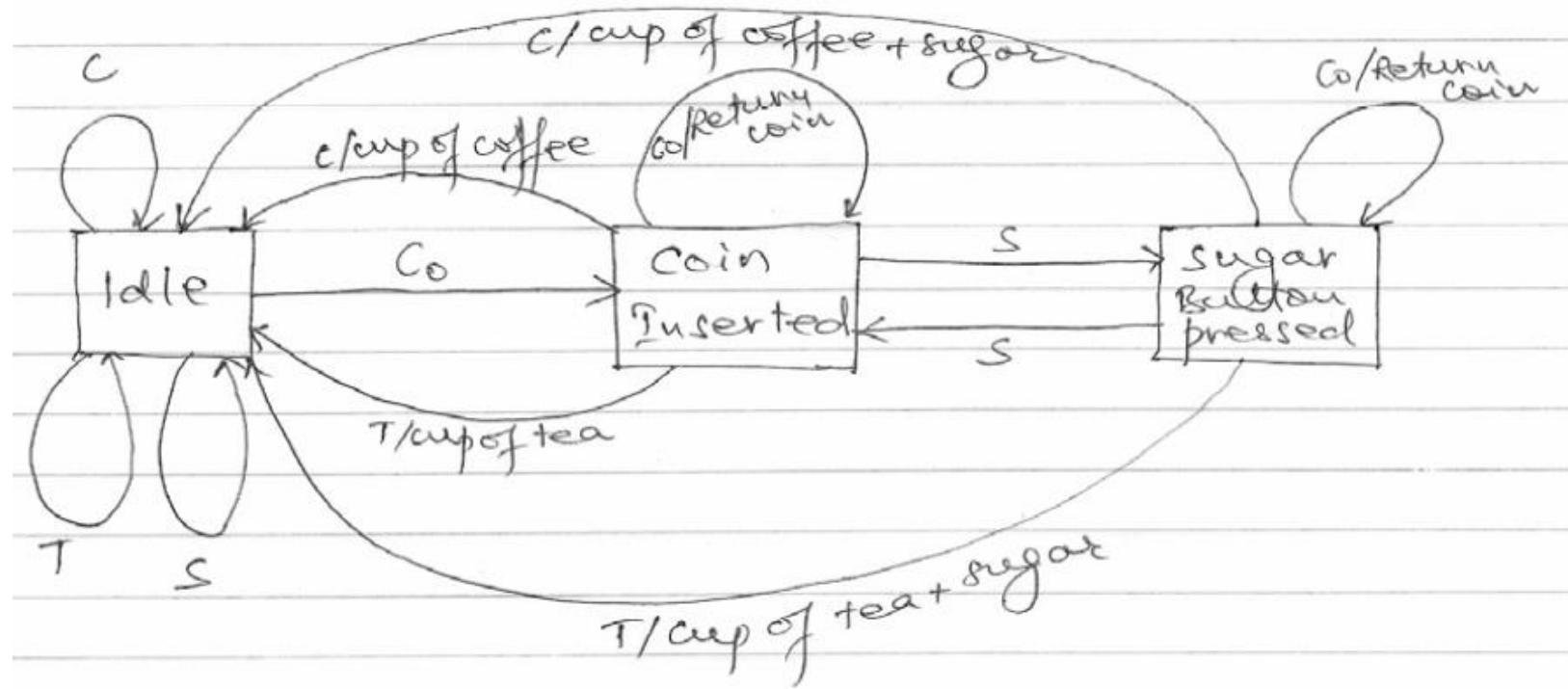


- Tea/Coffee vending Machine
- Options
 - Accepts token/coin
 - Sugar

Vending Machine



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



Module 7 Self Study

- SS7.1 Compare & Contrast – FSM, State Charts and Petri Nets
- SS7.2 Choose a system or a product of day to day use. Model the same using FSM. Derive the test cases based on the Model.



SS 7.1 Compare & Contrast – FSM, State Charts and Petri Nets

7.1 Self Study

To explore:

- Compare & Contrast – FSM, State Charts and Petri Nets

Study Work:

- Take up some among of research of literature on the three – FSM, StateCharts and Petri Nets
- Compare an contrast these techniques on at least 3 unique attributes



SS 7.2 Modelling a system/product of day-to-day use

7.2 Self Study

To explore:

- Model a system

Study Work:

- Choose a system of day to day use (example: Traffic Light, Lift)
- Model the system using one of the discussed techniques
- Derive the test cases for the system

Model Based Testing

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

System of Systems - Characteristics

- A super system
- A collection of cooperating systems
- A collection of autonomous systems
- A set of component systems

Specific Attributes – Maier

- They are built from components that are (or can be) independent systems
- They have managerial/administrative independence
- They are usually developed in an evolutionary way
- They exhibit emergent (as opposed to preplanned) behaviours

Definitions (Maier)

- A directed system of systems is designed, built, and managed for a specific purpose
- A collaborative system of systems has limited centralized management and control
- A virtual system of systems has no centralized management and control

Larger Systems

- Garage Controller (Directed)
- Building Automation (Directed)
- Air Traffic Control Systems (Acknowledged)

Module 8: Agenda

Module 8: Model Based Testing (2/2)

Topic 8.1

Model Based Testing – Systems

Topic 8.2

Model Based Testing – System of Systems

Topic 8.3

Examples



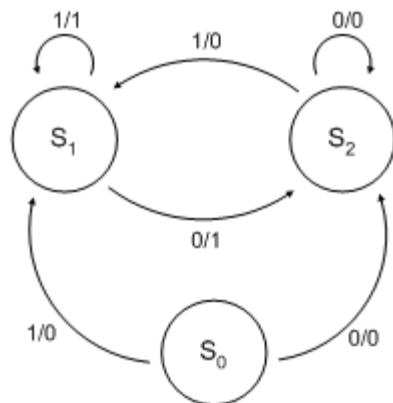
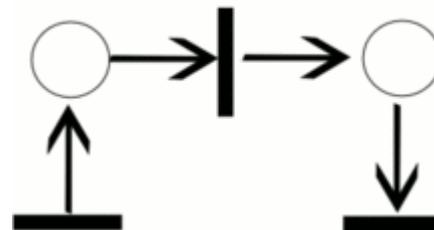
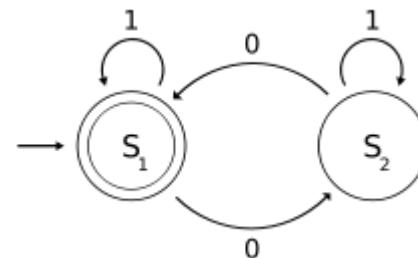
Topic 8.1: Model Based Testing – Systems

Model Based Testing

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





Topic 8.2: Model Based Testing – System of Systems

System of Systems - Characteristics

- A super system
- A collection of cooperating systems
- A collection of autonomous systems
- A set of component systems

Specific Attributes – Maier

- They are built from components that are (or can be) independent systems
- They have managerial/administrative independence
- They are usually developed in an evolutionary way
- They exhibit emergent (as opposed to preplanned) behaviours

Definitions (Maier)

- A directed system of systems is designed, built, and managed for a specific purpose
- A collaborative system of systems has limited centralized management and control
- A virtual system of systems has no centralized management and control

Larger Systems

- Garage Controller (Directed)
- Building Automation (Directed)
- Air Traffic Control Systems (Acknowledged)



Topic 8.3: Examples



SS 8.1 Systems & Systems of Systems

8.1 Self Study

To explore:

- Systems and Systems of Systems around us

Study Work:

- Review systems and system of systems around us
- Write down their characteristics and working (in terms of use cases)
- Further, analyse use of techniques learnt so far for designing test cases
- Document your findings in form of a whitepaper (IEEE format recommended)

Object Orientation

- Well-conceived objects encapsulate functions and data “that belong together”
- Composition
- Enable – Reuse without modification or additional testing

Units for OO

- A unit is the smallest software component that can be compiled and executed
- A unit is a software component that would never be assigned to more than one designer to develop

Implications of Inheritance

- Role of inheritance complicates the choice of classes as units

Class Flattening

- Flattening of classes
- A flattened class is an original class expanded to include all the attributes and operations it inherits

Issues with Flattening

- Uncertainty – flattened class is not part of final system
- Similar issue as instrumented code

Polymorphism

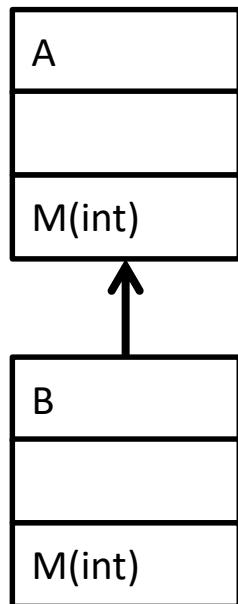
- Using same name for different services

$$y=F(x)$$

$$z=F(x, y)$$

- Polymorphism related to inheritance

- Using same name and interface for different services



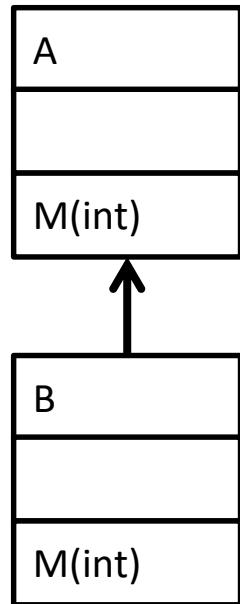
A a;

B b;

a.M(x);

b.M(x);

Polymorphism – Dynamic Objects



```
A *pa;  
B *pb;
```

```
pa=new A;  
pb=new B;
```

```
pa→M(x); 1st  
pb→M(x); 2nd  
pa=pb;  
pa→M(x); 1st
```

Testing Constructors

- Constructor: A method that is executed when the object is created
- Constructors with no parameters
- Constructor with parameters

Strategy

- Create the object and check the state of the object

Testing Destructors

- Destructors: Always executed when object of class is destroyed
- Static Object: On program termination
- Dynamic Object: Object is disposed

Strategy

- Create the object and check the state of the object

Class Pair

```
Class Pair
{
    public:
        void set_x(int);
        int get_x();
        void set_y(int);
        int get_y();
    private:
        int x;
        int y;
}
```

Implementation:

```
Void Pair::set_x(int v)
{
    if (v>0) x=v;
}

Int Pair::get_x()
{
    if (x>0) return x;
    return 0;
}
```

How do you test this class?

Use of test techniques

- Specification Based Testing
 - BVA
 - EC
 - DT
 - CEG
- Code Based Testing
 - Statement, Branch, Loop
 - McCabe Path, Basis Path
 - CF (Data and/or Control)
- Special Value Testing

Use of special methods

- Methods that view state of object
- Drivers (Input and Output)
- Introduce additional methods
 - Return results of execution of method
 - Set/Get values of some private data
- Test using a sequence of operations (data and methods)

Methods as Units

- This choice reduces OO to procedural unit testing (Method = Procedure)
- Need for stub and driver
 - Availability of other methods
- Look for messages

Classes as Units

- Entire class as unit solves intraclass problem
- Views of class testing
 - Static view: as we read the source code
 - Compile time view: Inheritance occurs
 - Execution view: Abstract classes
- Need for other classes
- Flattening of classes

Method or Class as Unit

- Explore both
- Come up with your reasons for making a choice
- Review examples in the text
- Apply them for the stack() and queue()
data structures and their implementations

Currency Converter

INR Amount

Equivalent in



US Dollar



Canadian Dollar



Euro



Pound



Chinese Yuan

Compute

Clear

Quit

Module 9: Agenda

Topic 9.1

OO Software & OO Software test –
Introduction & Overview

Topic 9.2

Issues in Testing OO Software

Topic 9.3

OO Unit Testing

Topic 9.4

Examples



Topic 9.1: OO Software & OO Software test – Introduction & Overview

Object Orientation

- Well-conceived objects encapsulate functions and data “that belong together”
- Composition
- Enable – Reuse without modification or additional testing

Object – An example

Dog as an object

State

- Age
- Name
- Color

Behaviour

- Barking
- Hungry
- Sleeping



Source: <http://retrieverman.files.wordpress.com/2011/09/german-shepherd-dog.jpg>

Thinking OO – Various Examples



- Point – Line
- Data Structures – Stack/Queue
- Employee Data
- ...

Levels – A discussion

- What is a unit?
- Implications of strategy of composition
- Inheritance, encapsulation & polymorphism
- Class, GUI, integration and system testing
- Data flow



Topic 9.2 : Issues in Testing OO Software

Units for OO

- A unit is the smallest software component that can be compiled and executed
- A unit is a software component that would never be assigned to more than one designer to develop

Units for OO

- Class as unit
- Integration – clear goals
 - To check the cooperation of separately tested classes

Implications of Inheritance

- Role of inheritance complicates the choice of classes as units

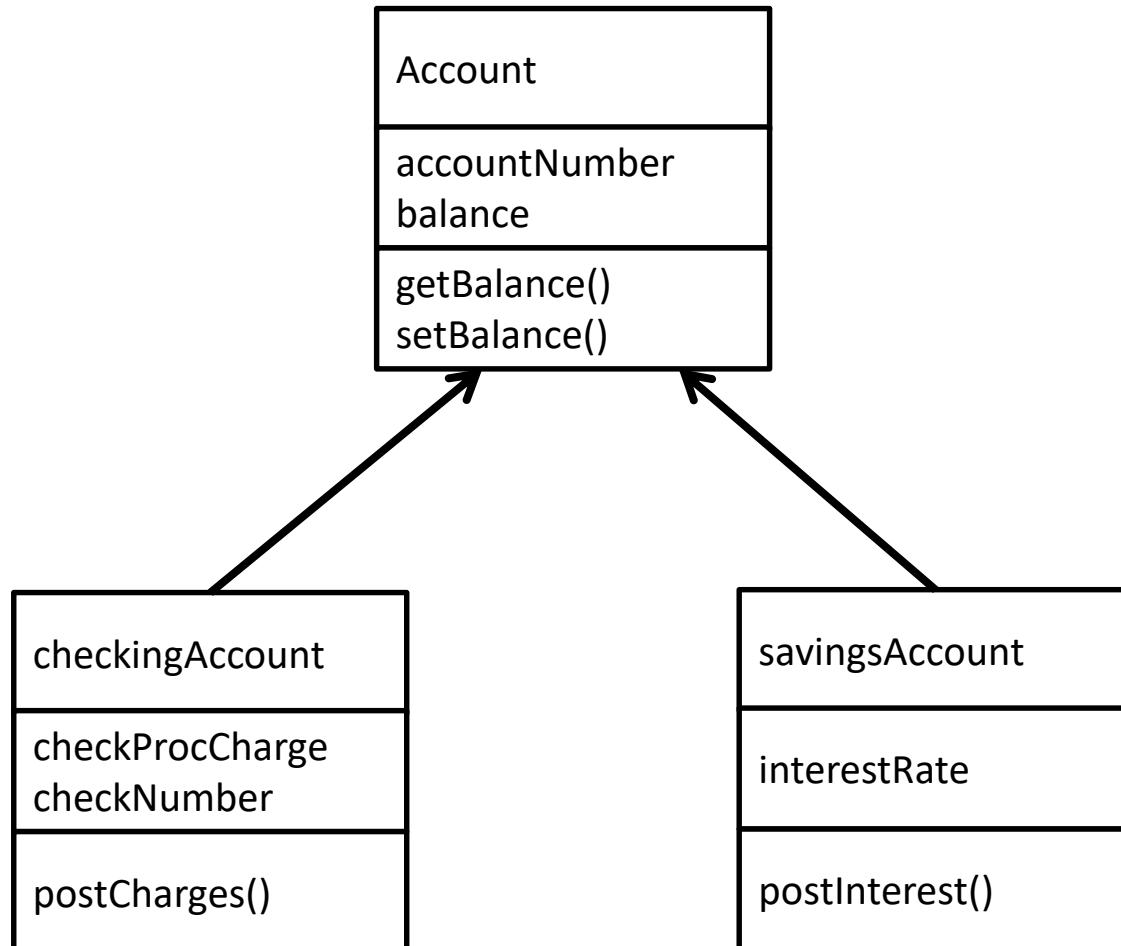
Class Flattening

- Flattening of classes
- A flattened class is an original class expanded to include all the attributes and operations it inherits

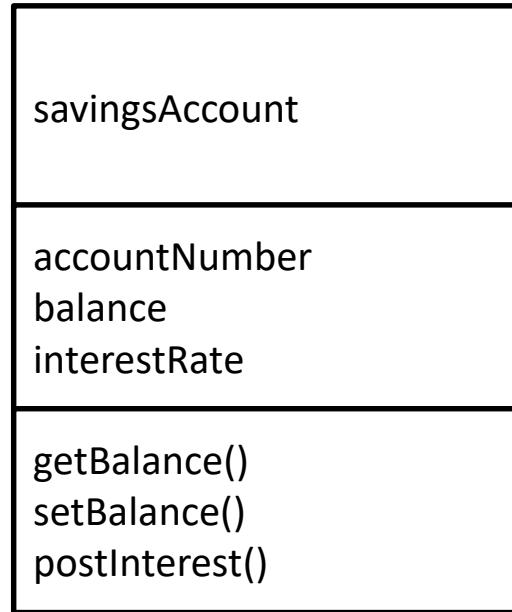
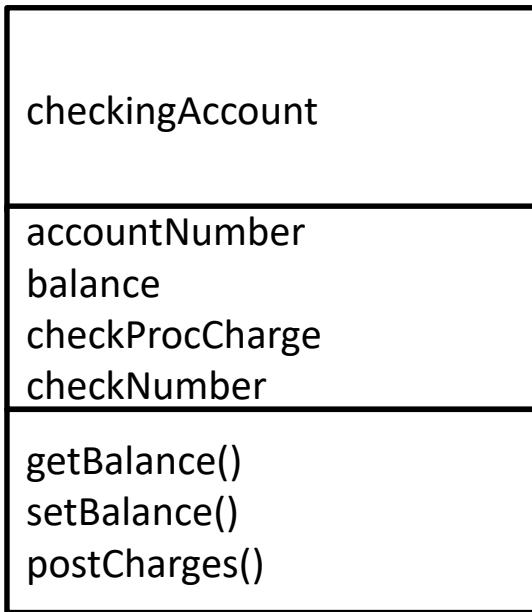
Issues with Flattening

- Uncertainty – flattened class is not part of final system
- Similar issue as instrumented code

Flattening Examples



Flattening Examples



Polymorphism

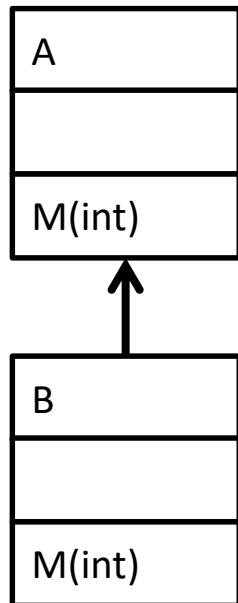
- Using same name for different services

$$y=F(x)$$

$$z=F(x, y)$$

- Polymorphism related to inheritance

- Using same name and interface for different services



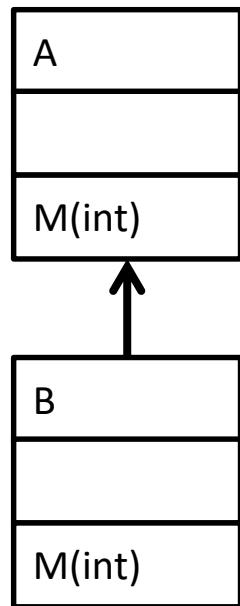
A a;

B b;

a.M(x);

b.M(x);

Polymorphism – Dynamic Objects



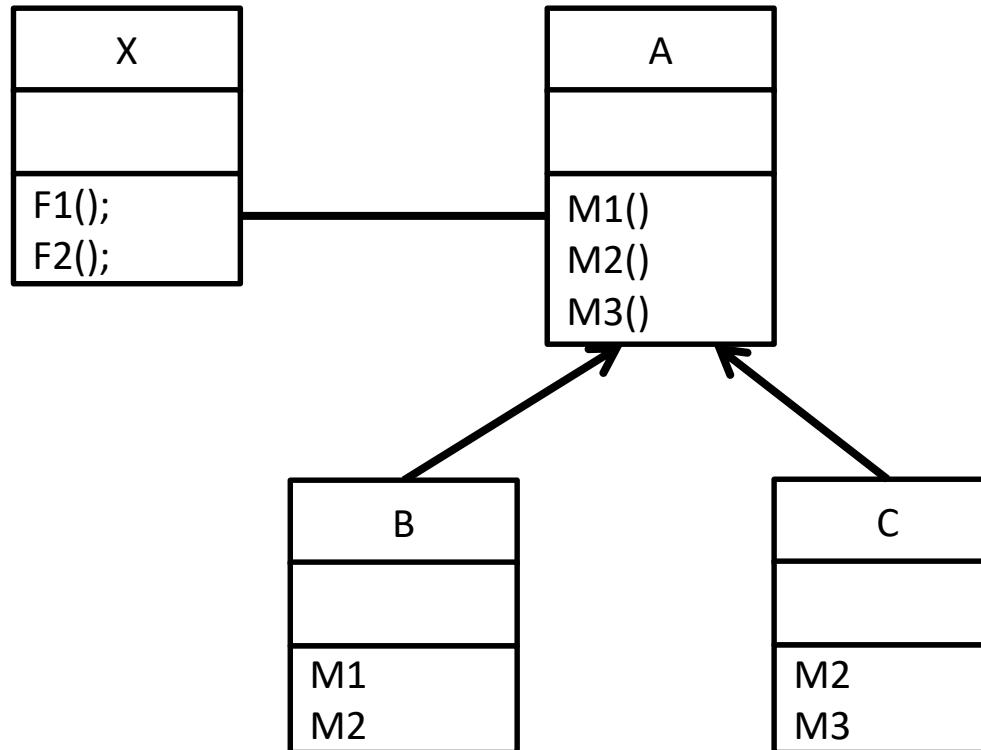
```
A *pa;  
B *pb;
```

```
pa=new A;  
pb=new B;
```

```
pa→M(x); 1st  
pb→M(x); 2nd  
pa=pb;  
pa→M(x); 1st
```

Testing Polymorphism

Object of class A or B or C (Different bindings)



Testing Polymorphism

Testing class X

Method F1 ()

```
void F1 ()  
{  
    A *p;  
  
    ...  
  
    p→M1 ();  
  
    ...  
}
```

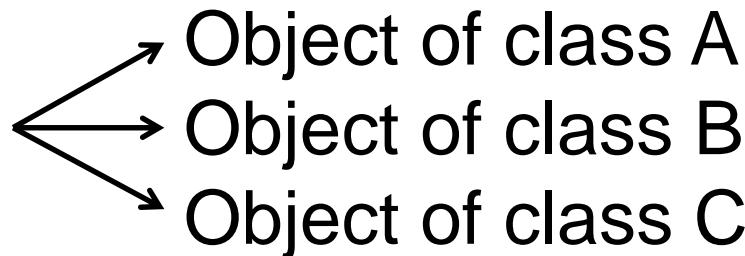
Testing Polymorphism

Testing class X

Method F1 ()

```
void F1 ()  
{  
    A *p;  
    ...  
    p→M1 ();  
    ...  
}
```

Object of class A
Object of class B
Object of class C



Testing Polymorphism

Testing class X

Method F1 ()

```
void F1 ()  
{
```

```
    A *p;
```

```
...
```

```
p→M1 ();
```

```
...
```

```
}
```



Thus

TC1: p→M1() on class A
TC2: p→M1() on class B
TC3: p→M1() on class C

Object of class A
Object of class B
Object of class C

Testing Polymorphism

- Concept: Develop a set of test cases for a client (class X) that exercise all client bindings to the polymorphic server. Test every polymorphic call.
- Write the code snippets for to depict the test case (HW)

Testing Constructors & Destructors



- Review and relook at testing Constructors and Destructors
- How?
- Challenges

Testing Constructors

- Constructor: A method that is executed when the object is created
- Constructors with no parameters
- Constructor with parameters

Strategy

- Create the object and check the state of the object

Constructors with no parameters

1. Create an Object

```
Y *p;
```

```
p = new Y
```

2. Check the results

```
p->checkstate()
```

```
delete p
```

Constructors with parameters

1. Create an Object

```
Y *p;
```

```
Input(a, b) // enter data
```

```
p = new Y(a, b)
```

2. Check the results

```
p->checkstate()
```

```
delete p
```

Testing Destructors

- Destructors: Always executed when object of class is destroyed
- Static Object: On program termination
- Dynamic Object: Object is disposed

Strategy

- Create the object and check the state of the object

Testing Destructors

Create and Destroy

```
Y *p;  
p = new Y  
...  
delete p // Object is destroyed
```

Use checkstate() in the destructor



Topic 9.3 : OO Unit Testing

Object Orientation

- Well-conceived objects encapsulate functions and data “that belong together”
- Reuse without modification or additional testing

Units for OO – definition

- A unit is the smallest software component that can be compiled and executed
- A unit is a software component that would never be assigned to more than one designer to develop

Units for OO

- Class as unit
- Integration – clear goals
 - To check the cooperation of separately tested classes

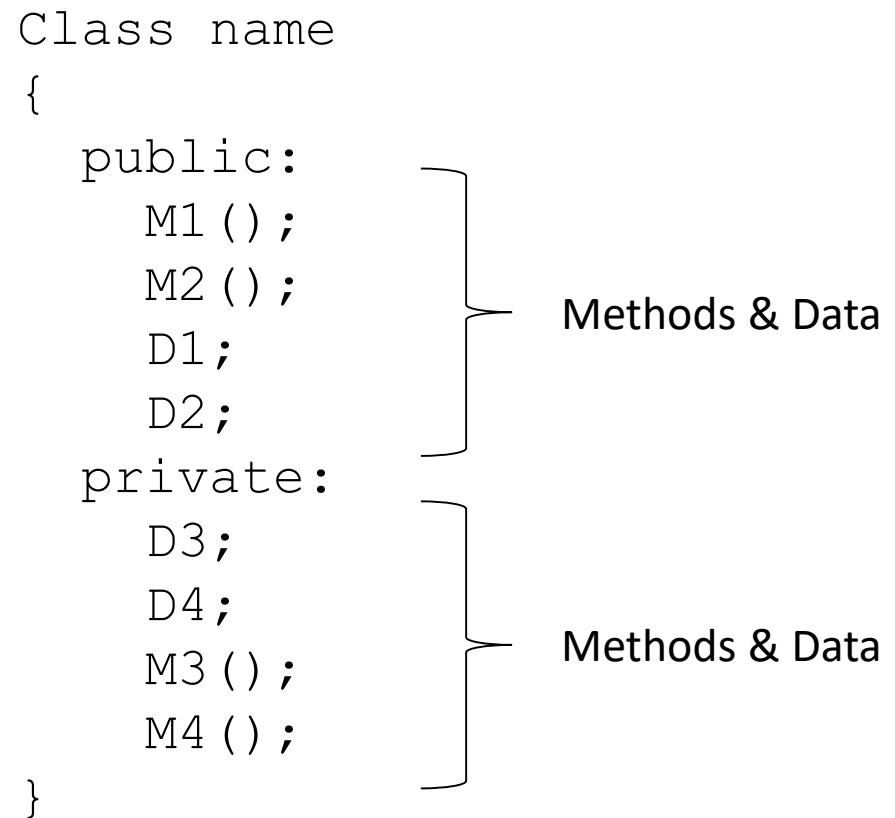
Typical Class

- A Class is a user defined data type
- Class has
 - Interface
 - Operations
 - Data

Typical Class

- A Class is a user defined data type
- Class has
 - Interface
 - Operations
 - Data

```
Class name
{
    public:
        M1 ();
        M2 ();
        D1;
        D2;
    private:
        D3;
        D4;
        M3 ();
        M4 ();
}
```



Methods & Data

Methods & Data

Class Pair

```
Class Pair
{
    public:
        void set_x(int);
        int get_x();
        void set_y(int);
        int get_y();
    private:
        int x;
        int y;
}
```

Implementation:

```
Void Pair::set_x(int v)
{
    if (v>0) x=v;
}

Int Pair::get_x()
{
    if (x>0) return x;
    return 0;
}
```

Class Pair

```
Class Pair
{
    public:
        void set_x(int);
        int get_x();
        void set_y(int);
        int get_y();
    private:
        int x;
        int y;
}
```

Implementation:

```
Void Pair::set_x(int v)
{
    if (v>0) x=v;
}

Int Pair::get_x()
{
    if (x>0) return x;
    return 0;
}
```

How do you test this class?

Use of test techniques

- Specification Based Testing
 - BVA
 - EC
 - DT
 - CEG
- Code Based Testing
 - Statement, Branch, Loop
 - McCabe Path, Basis Path
 - CF (Data and/or Control)
- Special Value Testing

Use of special methods

- Methods that view state of object
- Drivers (Input and Output)
- Introduce additional methods
 - Return results of execution of method
 - Set/Get values of some private data
- Test using a sequence of operations (data and methods)

Develop these for class Pair and Stack

Methods as Units

- This choice reduces OO to procedural unit testing (Method = Procedure)
- Need for stub and driver
 - Availability of other methods
- Look for messages

Classes as Units

- Entire class as unit solves intraclass problem
- Views of class testing
 - Static view: as we read the source code
 - Compile time view: Inheritance occurs
 - Execution view: Abstract classes
- Need for other classes
- Flattening of classes

Method or Class as Unit

- Explore both
- Come up with your reasons for making a choice
- Review examples in the text
- Apply them for the stack() and queue()
data structures and their implementations



Topic 9.4 : Examples

Typical Examples

- Queue
- Stack
- Automated Teller Machine
- Vending Machine
- Washing Machine
- UI of a Mobile Phone
- UI of Settop Box

Currency Converter

INR Amount

Equivalent in



US Dollar



Canadian Dollar



Euro



Pound



Chinese Yuan

Compute

Clear

Quit

Chapter References

- T1 Chapter 15
 - Review of examples is key to understanding the aspects
 - Lecture discussion
- Review the examples
 - OO Calendar
 - Currency Converter Application
- References
- Testing Object Oriented Systems: models, patterns and tools, Robert V Binder, Addison Wesley

Module 9 Self Study

- SS9.1 Write a program in your chosen language (C++/Java) to sort a set of strings in alphabetical order (ascending or descending). Use OO methods. Design test cases for your work.
- SS9.2 Review the high level design of a UI framework like QT/GTK. Write a review on use of OO and testing techniques



SS 9.1 String sort

9.1 Self Study

To explore:

- Test techniques for OO
- Apply techniques for testing OO software

Study Work:

- You are required to write the program in your chosen language (C++/Java). Implement your own sorting algorithm.
- Create the test cases for the OO program using the techniques learnt
- Test your program using the test cases
- Analyse the results of the test run and comment on the effectiveness of the techniques



SS 9.2 Object Oriented UI frameworks study

9.2 Self Study

To explore:

- Application of testing techniques to OO software
- List the limitations that you come across

Study Work:

- Take up either GTK or QT as frameworks to study. Summarise their design. Explore the application of testing techniques to test the framework as well as applications developed using the framework
- Compare and contrast the effectiveness of the techniques to test the System



SS 8.1 Systems & Systems of Systems

8.1 Self Study

To explore:

- Systems and Systems of Systems around us

Study Work:

- Review systems and system of systems around us
- Write down their characteristics and working (in terms of use cases)
- Further, analyse use of techniques learnt so far for designing test cases
- Document your findings in form of a whitepaper (IEEE format recommended)

Module 10: Agenda

Module 10: Object Oriented Testing (2/2)

Topic 10.1

OO Integration Testing

Topic 10.2

OO System Testing

Topic 10.3

OO – GUI Testing

Topic 10.4

Examples



Topic 10.1: OO Integration Testing

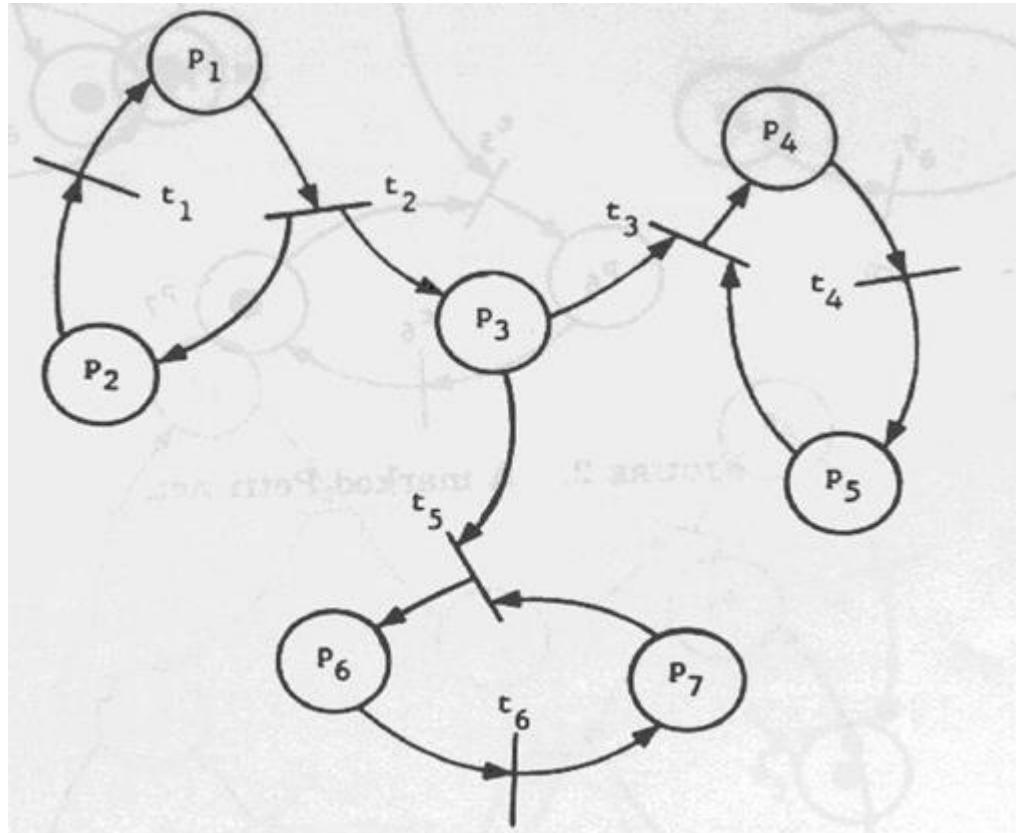
Petri Nets

- Major use:
 - Modeling of systems of events in which it is possible for some events to occur concurrently, but there are constraints on the occurrences, precedence, or frequency of these occurrences

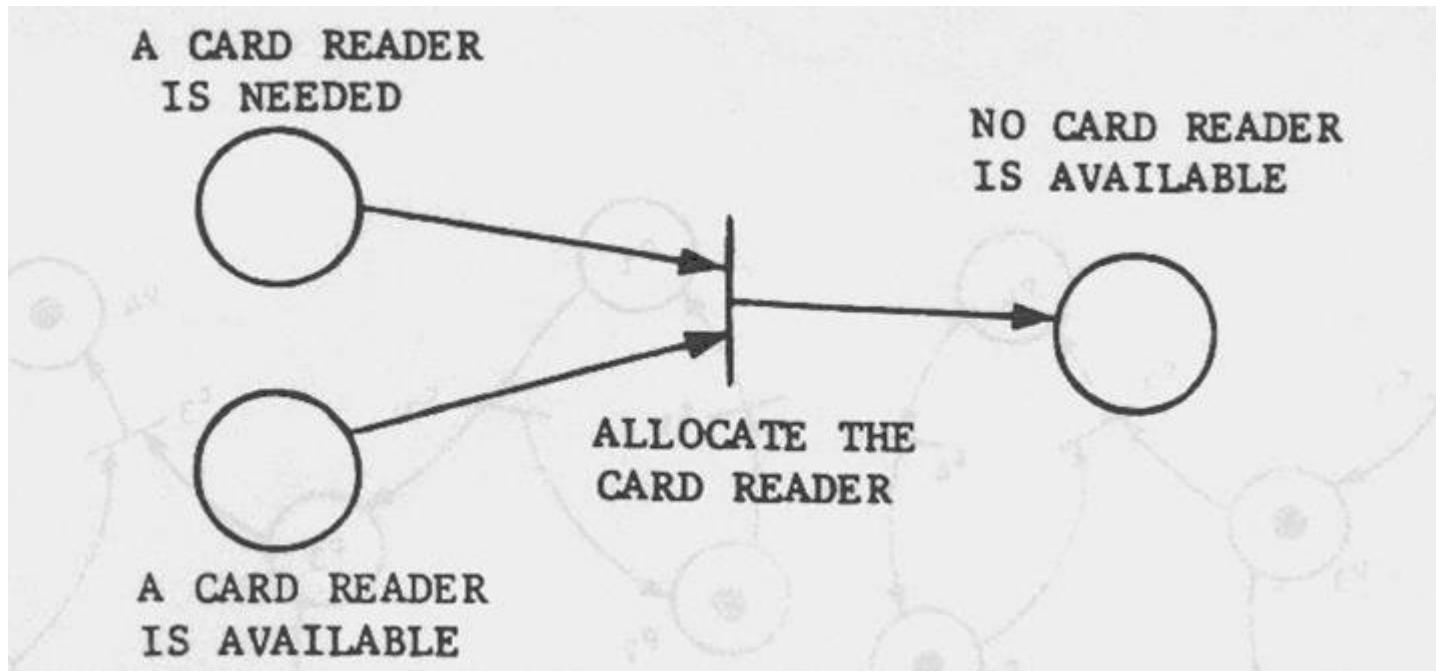
Petri Nets as a Graph

- Models static properties of a system
- Graph contains 2 types of nodes
 - Circles (Places)
 - Bars (Transitions)
- Petri net has dynamic properties that result from its execution
 - Markers (Tokens)
 - Tokens are moved by the firing of transitions of the net.

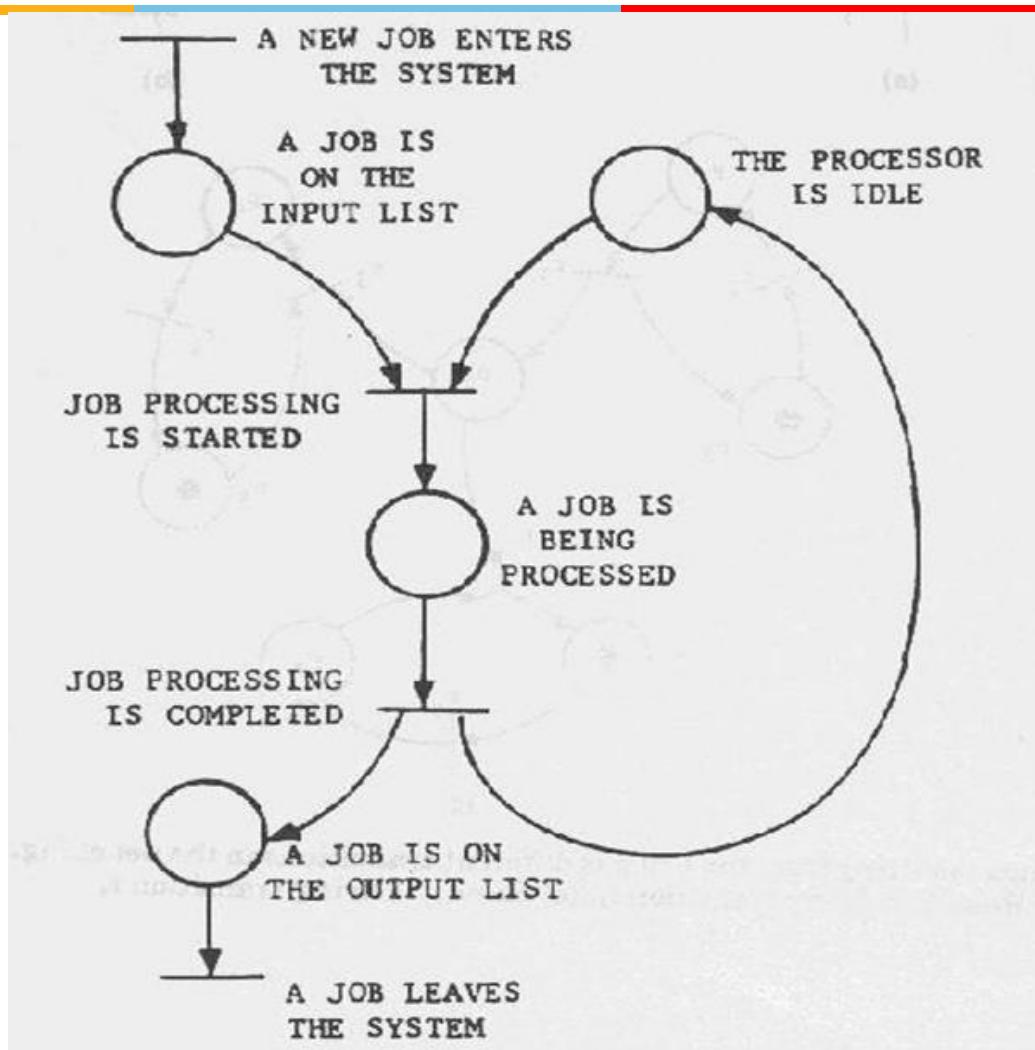
Graph representation



A Simple Model



Model Example



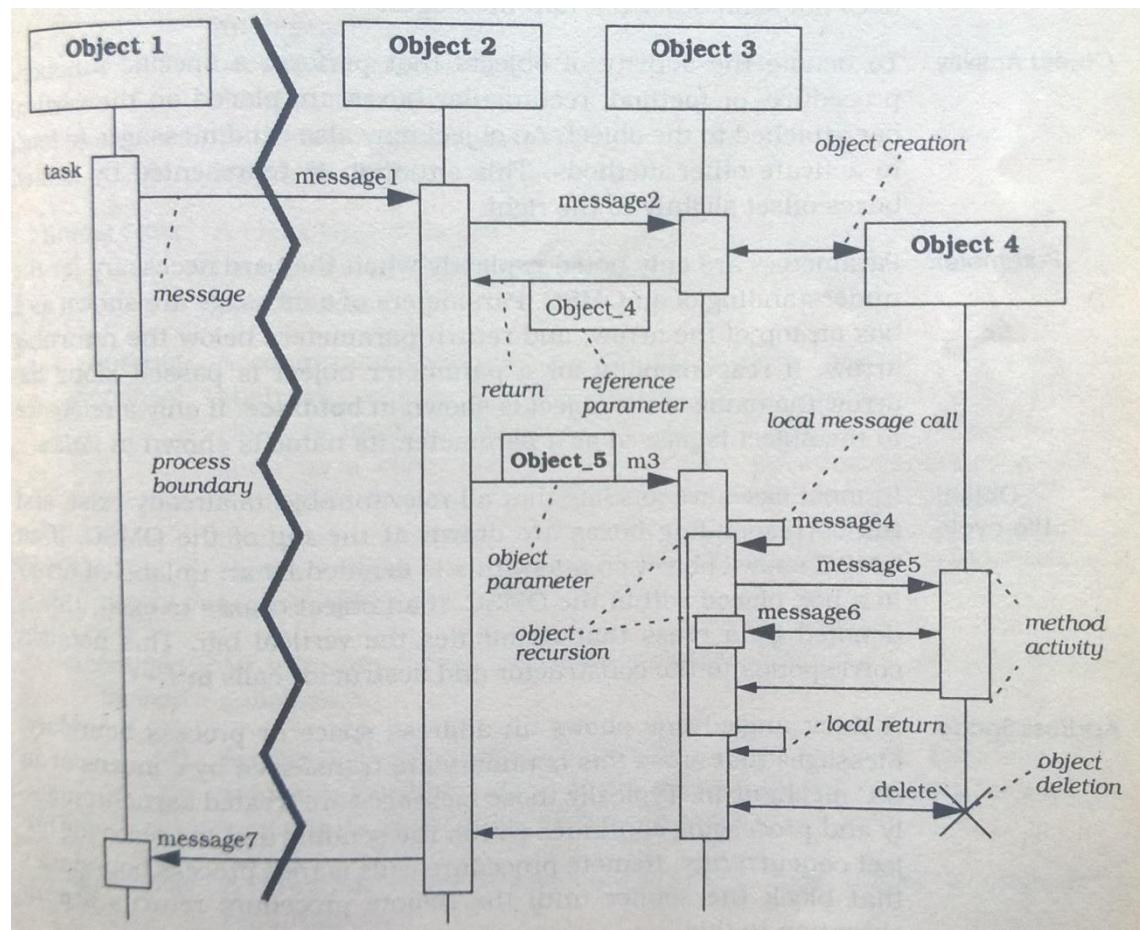
OO Integration Testing

- Flattened to Hierarchy
- Test methods to be removed
- Polymorphism – Test in each polymorphic context

Message Sequence Charts – Dynamic View

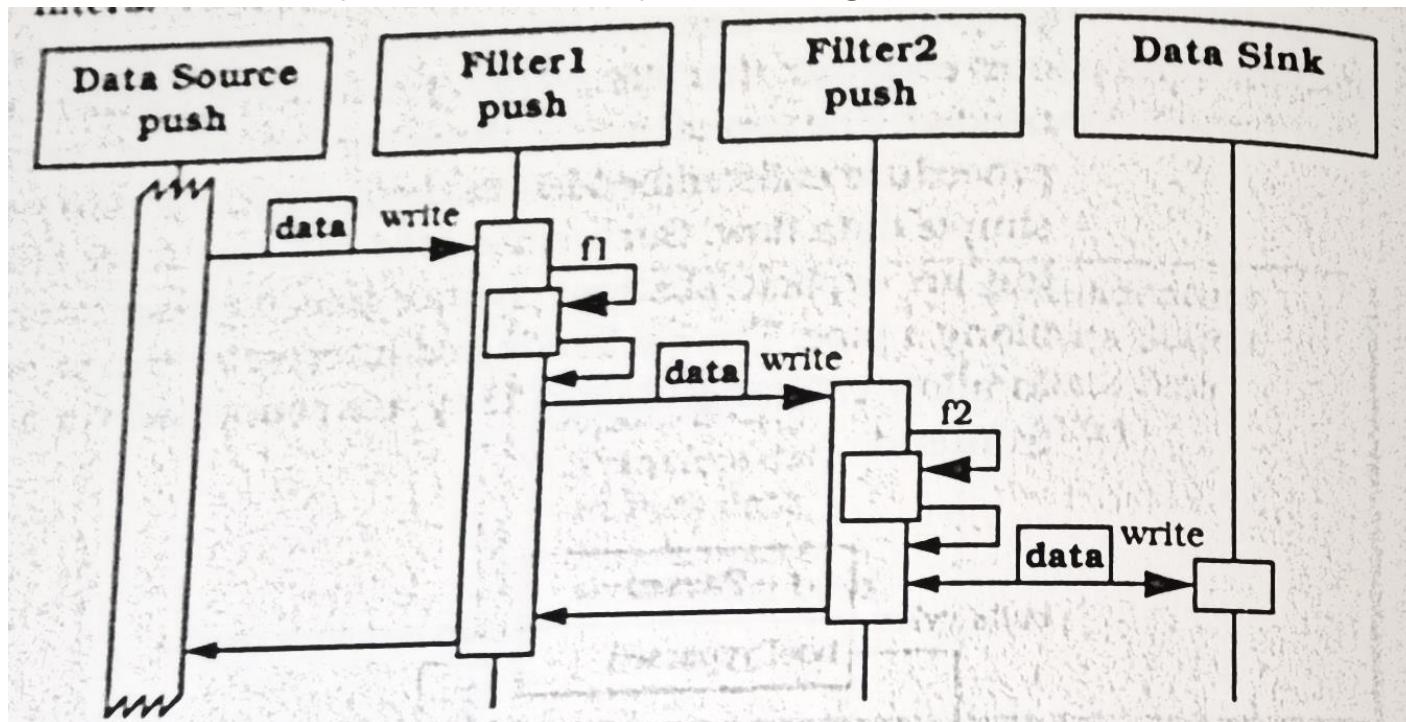


Adapt MSC to demonstrate object or component interaction among the participants of a pattern



Example

- Scenario for Pipes & Filters
 - Push pipeline
 - Filter activity started by writing data to the filters



OO Integration

- MM Paths (Sequence of method execution linked by messages)
- Use of Data Flow



Topic 10.2: OO System Testing

System Testing

- System testing is concerned with the behaviour of a whole system. The majority of the functional failures should already have been identified during unit and integration testing
- System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

System Testing

- It is essential to define what the “system” is made up of
 - A system for one team/organisation may mean a module/part of a larger system
- Generally system level is treated as closest to everyday experience i.e. system (Product) as seen in the hands of the user
- In system testing apart from finding faults the goal is to demonstrate
 - System works
 - System is reliable
 - System is durable
 - Quality of system for defined parameters
- Use of specification based testing techniques to design test cases

System Testing – Aspects

- Thread
 - Thread Possibilities
 - Thread Definitions
 - Atomic Thread Function

System Requirements – Basic Concepts



- Data
 - Information used and created by the system
- Actions
 - Transform, data transform, control transform, process, activity, task, method, service
- Devices
 - Source and destination of system level inputs and outputs
- Events
 - System level I/O; occurs on port device; inputs and outputs of actions
- Threads
 - Model of a system – interactions among data, events and action

OO System Test

- Test Engineer treats system as Black Box!
- Look for input and output events
- OO or Procedural Implementation
- Requirements model
- Use of Petri nets

Focus of Our Testing

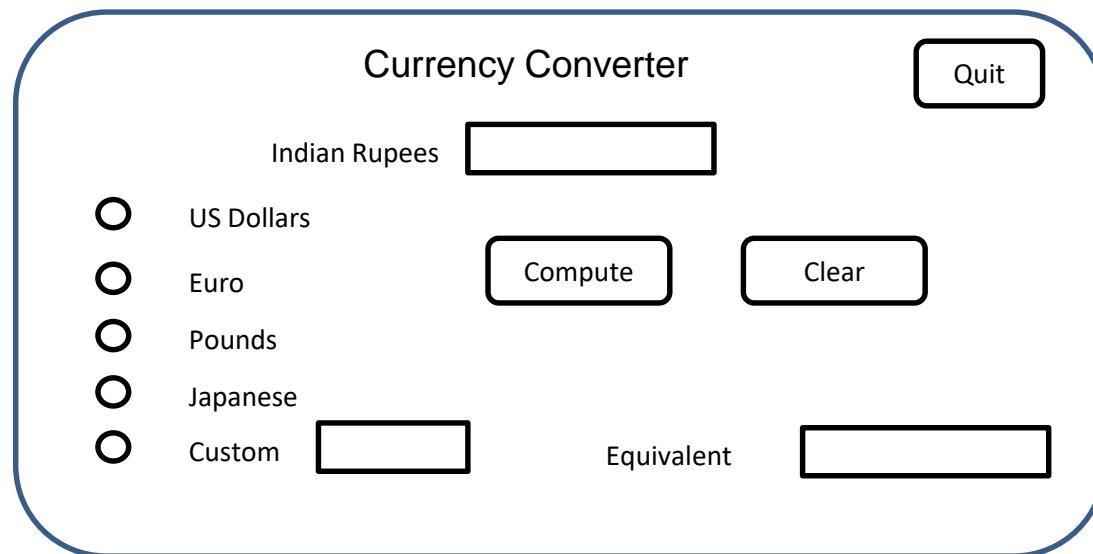
- Requirements
- System Functions
- Presentation Layer
- The way the user or customer sees it!

System Functions

- Use of description by customer/user in general terms
- User Stories → Use Cases
- Gives rise to system functions
 - Evident (Input currency amount)
 - Hidden (Maintain ex-or relationship among countries)
 - Frill (Display of country flags)

Presentation Layer

- Sketching of UI
- Relating User Experience to the user stories and use cases (Making it simple and easy to use)
- Screen transition and Navigation



Use Case Based Testing

- High level Use Cases
 - Essential Use Cases
 - Expanded Essential Use Cases
-
- Each level bring in more refinement in what is done as part of the use case
 - Refine only as much it is required

Example: Use Case

HLUC

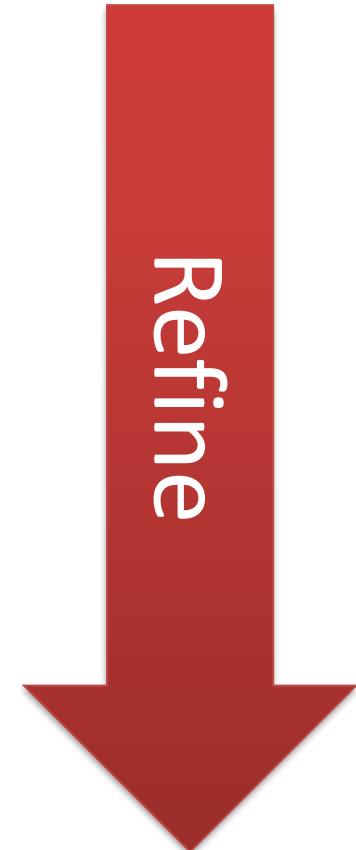
- User Starts the currency converter application in Windows

EUC

- Add an event sequence
- Input: User starts the application by using Run command, double click the application icon
- Output: The CC application GUI appears on the screen

EEUC

- Add next level details
- Details like: Which input box or area has focus. Specifics on static or dynamic data shown on the screen. Any details on memory or specifics of post-condition to be checked.



State Based

- State Based Testing
- Use of Extended Finite State Machine



Topic 10.3: OO – GUI Testing

Currency Converter System

Currency Converter

Quit

Indian Rupees

US Dollars

Euro Compute Clear

Pounds

Japanese

Custom Equivalent

Top areas of focus

- Main characteristic of GUI Testing: Event Driven
 - Users can cause any of several events in any order
- Buttons have functions; and they can be tested
- Focus: test the event driven nature: System Testing of GUI
 - Also at level of unit one can test the function of buttons
- Use of Finite State Machines and Extended Finite State Machines

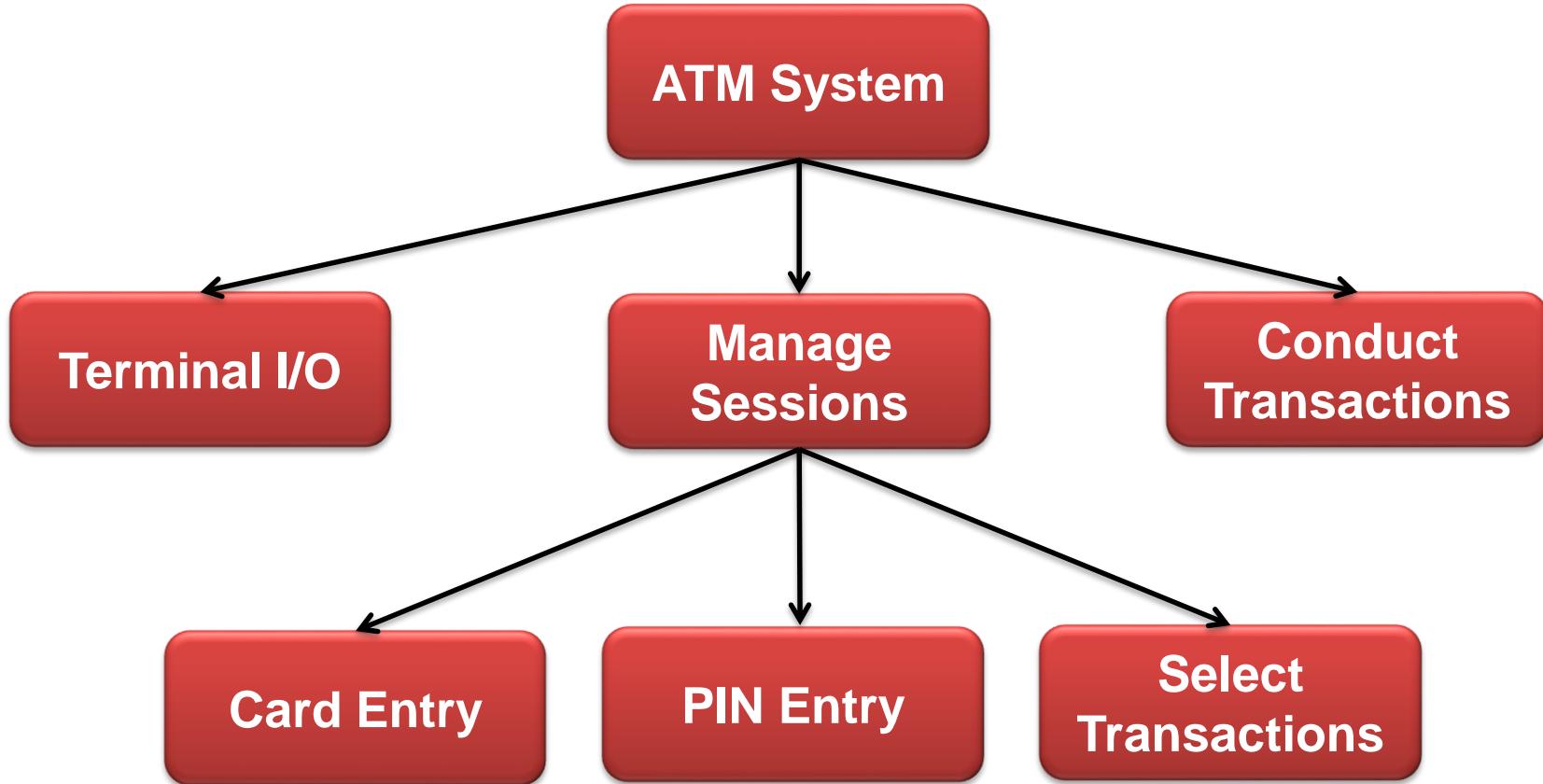
Testing a GUI Application

- Identify all the user input events
- Identify all the system output events
 - Externally visible & observable
- Use the System Level State Machine (State Chart)
 - Use of artificial events (Idea is to simplify the system level state machine)

Unit & Integration Testing

- Example of currency converter program
- Unit Level Testing (Level: Open for discussion)
 - Use of Input and Output for that “unit”
 - System level unit testing! (This poses some issues)
 - Test Driver is preferred for unit testing (Ensures testing of input, output and/or computation)
 - Method or Class as Unit (Remember our discussion on choosing either)
- Integration Level Testing
 - With unit level thoroughly tested; is integration required?
 - Definition of “integration level”

Automated Teller Machine



Source: T1: Figure 12.2

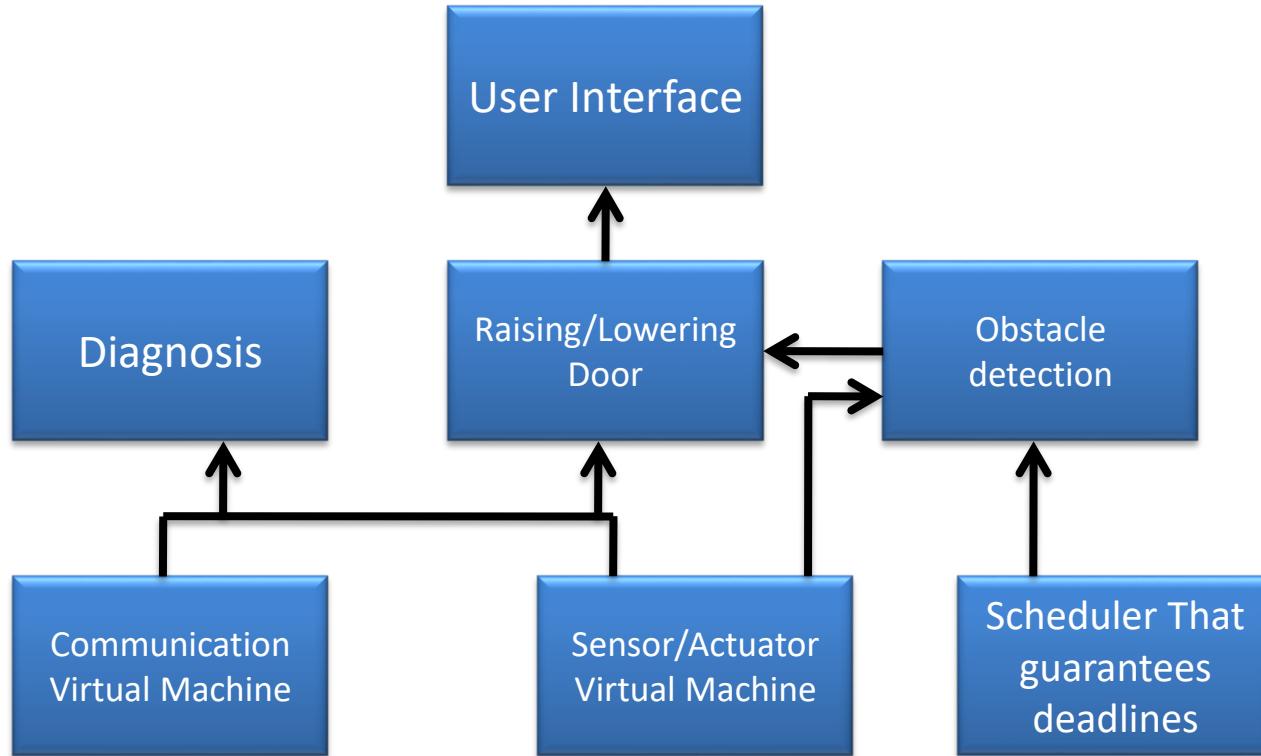


Topic 10.4: Examples

Garage Door

- The device controls for opening and closing the door are different for the various product lines. They may include controls from within a home information system. The product architecture for a specific set of controls should be directly derivable from the product line architecture
- The processor used in different processes will differ. The product architecture for each specific processor should be directly derivable from the product line architecture
- If an obstacle (person or object) is detected by the garage door during descent, it must halt (alternately re-open) within 0.1 second
- The garage door opener should be accessible for diagnosis and administration from within the home information system using product specific diagnosis protocol. It should be possible to directly produce an architecture that reflects this protocol

Garage Door



Higher Levels of Testing

- Integration Testing
- Sub-system Testing
- System Testing
- Product Testing
- *And More*

Integration Testing

- Integration testing is the process of verifying the interaction between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software
- Modern systematic integration strategies are rather architecture-drive, which implies integrating the software components or subsystems based on identified functional threads.

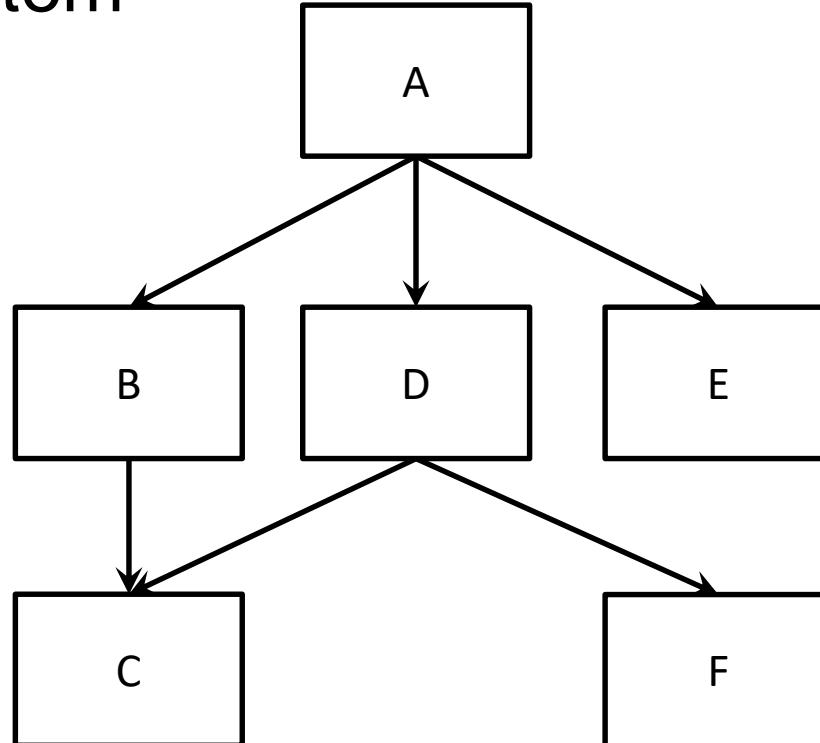
Source: SWEBOK 2004

Integration Testing

- Integration of Software
- Individually test “units” (components or modules or sub-systems)
- Structural
- Behavioural
- Alternative ways
 - Process driven
 - Customer commitment driven
 - Phased product release

Integration of Software

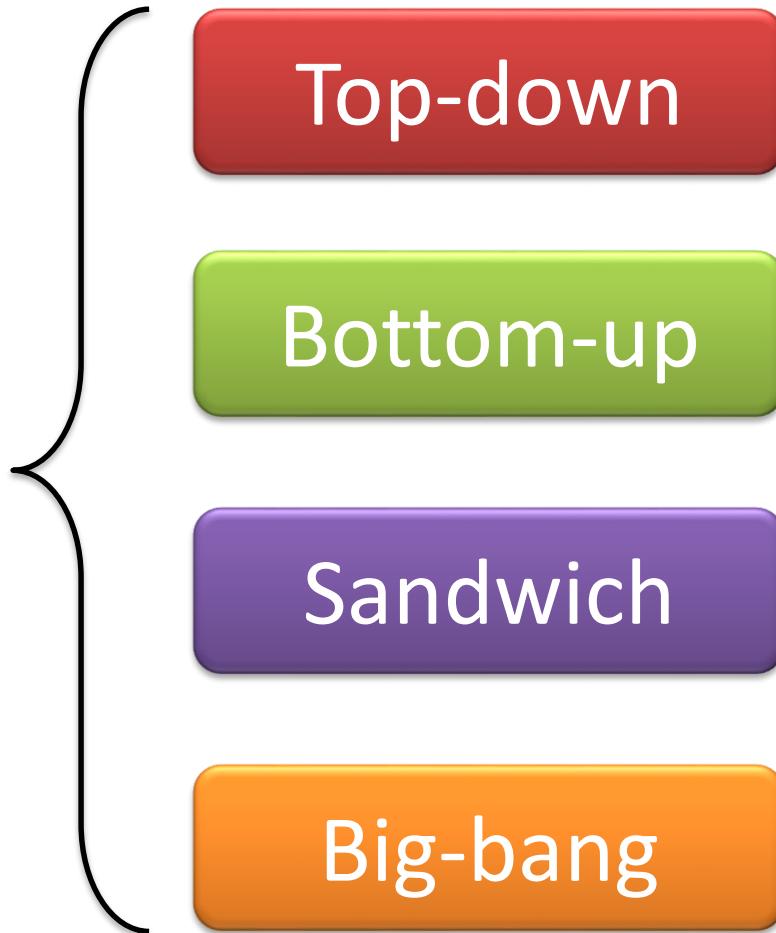
- Integration is concerned with integration of individual components, called “units”, to form the entire system



Decomposition Based Integration



Decomposition
Based Integration



Pros and Cons

- Need of Drivers
- Need of Stubs
- Number of Integration Cycles/Sessions
 - Sessions = nodes – leaves + edges
 - Number of stubs or drivers required
- Number of integration test cases and their runs
- Detection of defects
 - Early
 - Separation of defective module/component

Management driven processes versus engineering and development driven process

Comparison between DBI

Category	BU	TD	BB	S
Drivers	Yes	No	No	In-part
Stubs	No	Yes	No	In-part
Early Working version of the system	Late	Early	Late	Early
Early detection of interface errors	Early	Early	Late	Early
Parallel Testing	Medium	Low	High	Medium

Note: Individual testing of drivers and stubs will need to be considered separately

Call Graph-Based Integration

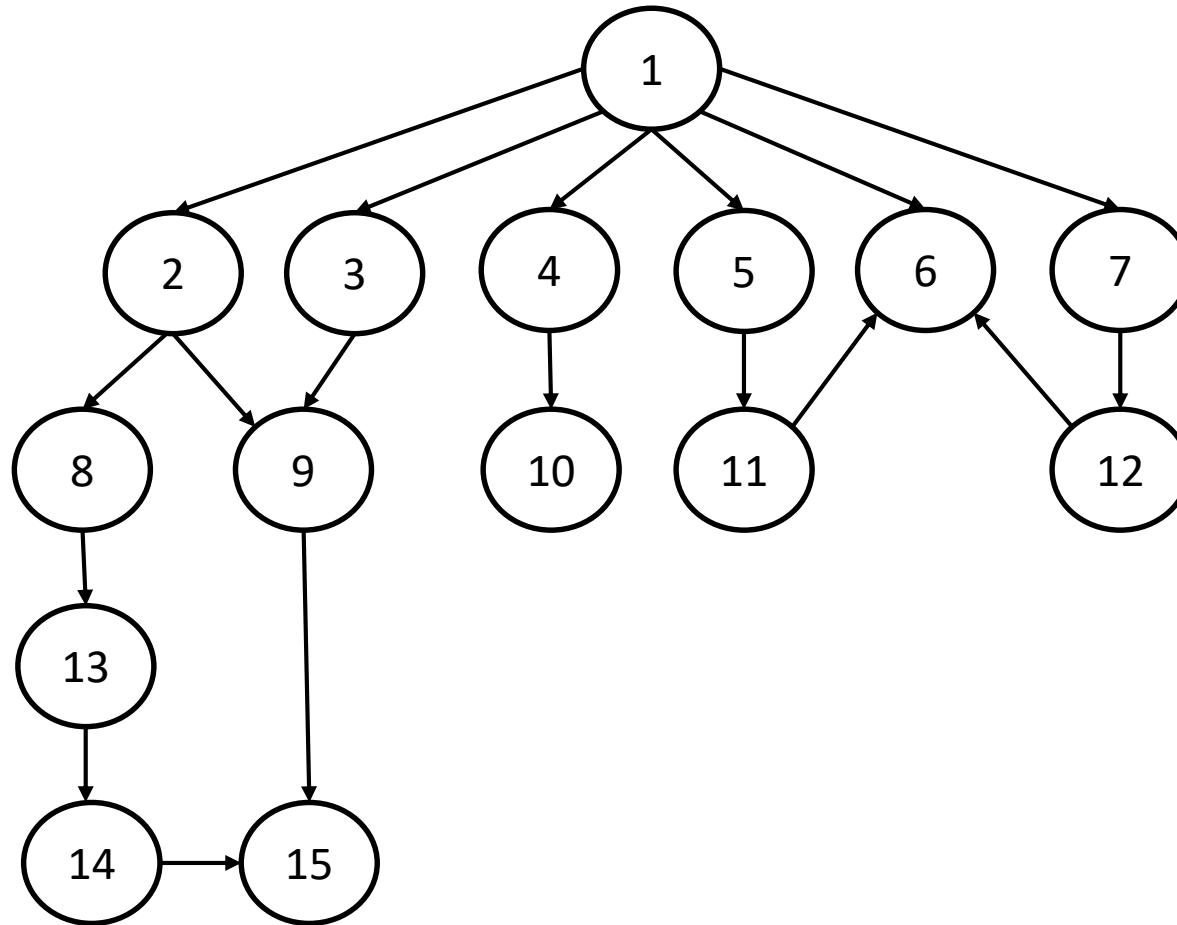
Call Graph-Based
Integration



Path-Based Integration

- Desirable → Structural and functional testing
- Express testing in terms of behavioural threads
- Focus on interaction (instead of interfaces)
 - Co-functioning
 - Interfaces are structural
 - Interactions is behavioural

A Sample Program



Refer: Page 239 of T1

Module 11: Agenda

Topic 11.0

High Level Testing – IT, SST, ST & PT

Topic 11.1

Integration Testing – Introduction, Overview & Issues

Topic 11.2

Integration Testing – Types & Strategies

Topic 11.3

Examples

Topic 11.4

Cases



Topic 11.0 High Level Testing – IT, SST, ST & PT

Higher Levels of Testing

- Integration Testing
- Sub-system Testing
- System Testing
- Product Testing
- *And More*



Topic 11.1: Integration Testing – Introduction, Overview & Issues

Integration Testing

- Integration testing is the process of verifying the interaction between software components. Classical integration testing strategies, such as top-down or bottom-up, are used with traditional, hierarchically structured software
- Modern systematic integration strategies are rather architecture-drive, which implies integrating the software components or subsystems based on identified functional threads.

Source: SWEBOK 2004

Insights

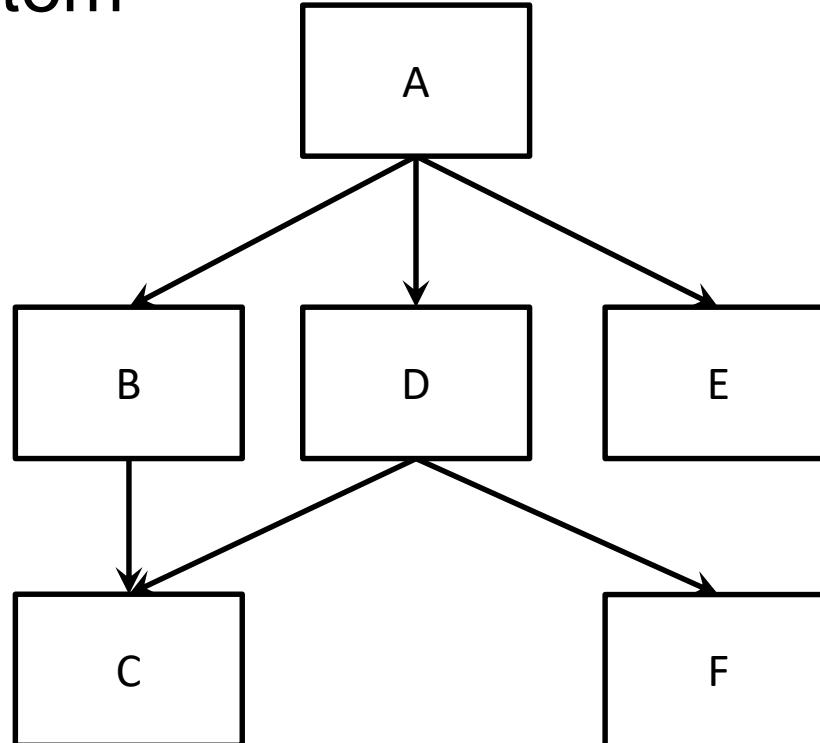
- Structural
- Behavioural

Integration Testing

- Integration of Software
- Individually test “units” (components or modules or sub-systems)
- Structural
- Behavioural
- Alternative ways
 - Process driven
 - Customer commitment driven
 - Phased product release

Integration of Software

- Integration is concerned with integration of individual components, called “units”, to form the entire system



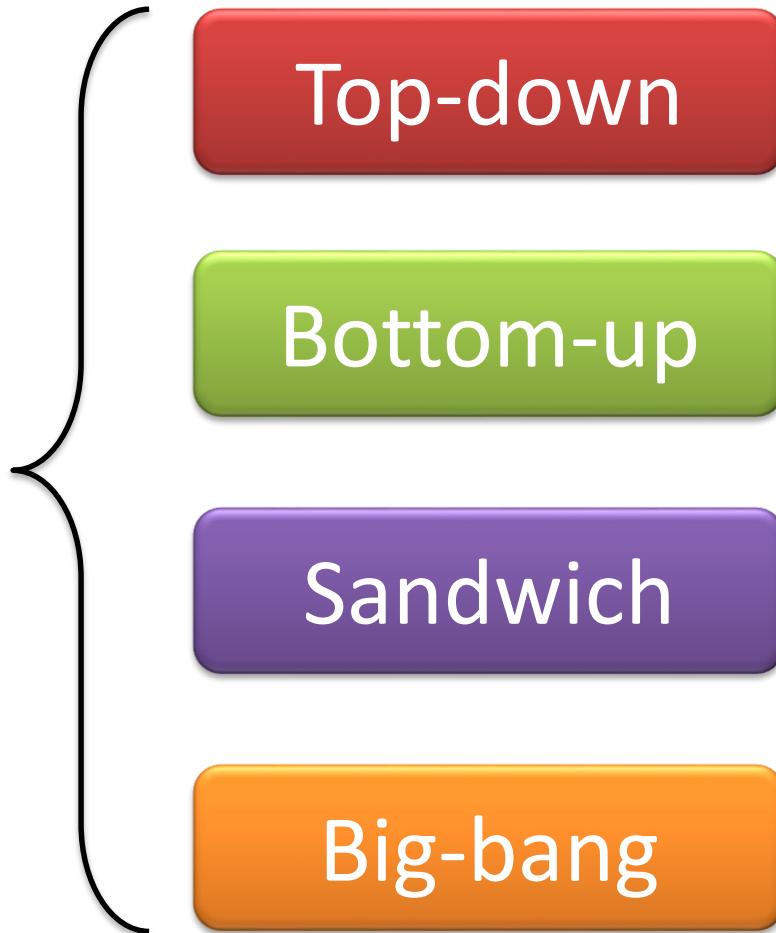


Topic 11.2 : Integration Testing – Types & Strategies

Decomposition Based Integration

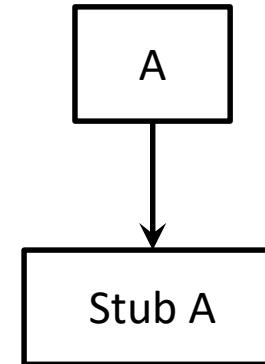
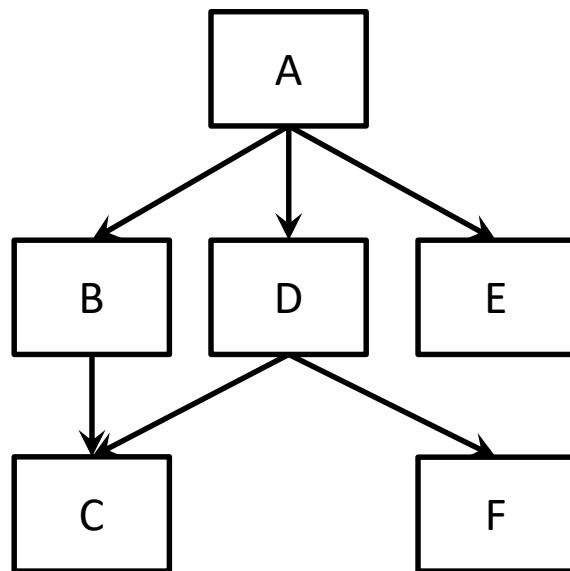


Decomposition
Based Integration



Top Down

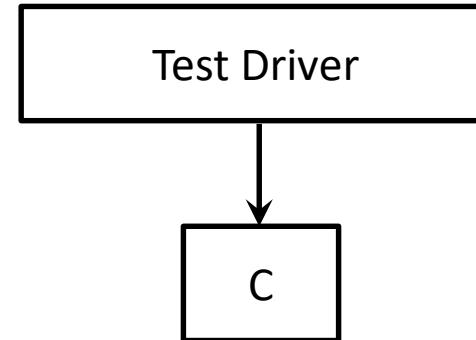
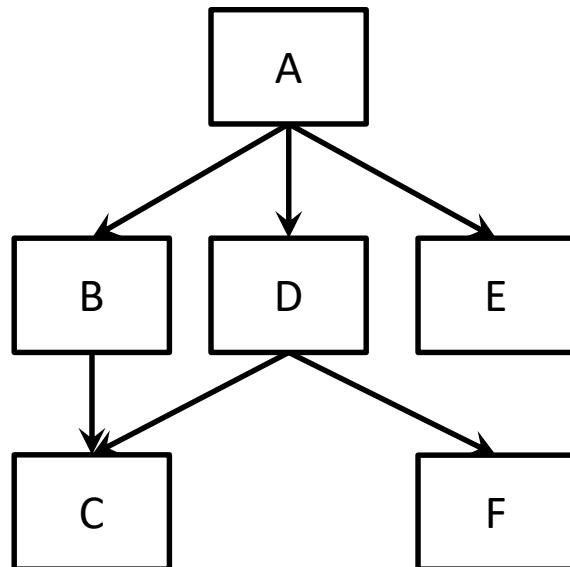
- A system is integrated Top to Bottom
- Integration Sequence: A B D E C F



A stub is a component that is used to simulate the system

Bottom Up

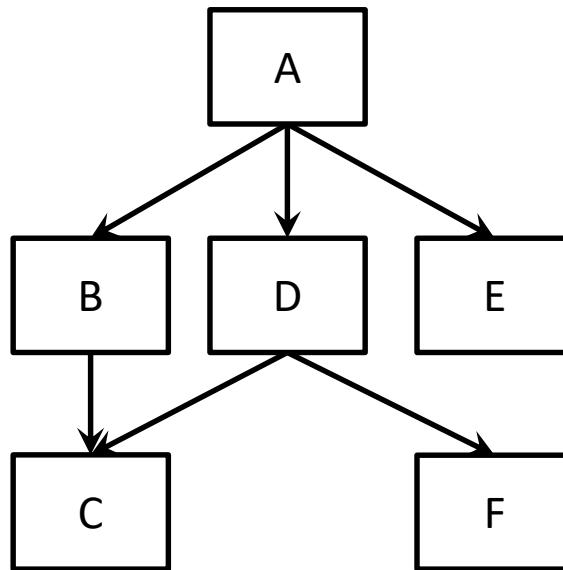
- A system is integrated Bottom to Top
- Bottom Units: C, D and F
- Integration Sequence: C D F B E A



A test driver is used to test (execute) the unit

Sandwich

- Combination of BU and TD



Big Bang

- Each unit is implemented (and separately tested)
- Integrate all units together to form the entire system
- Test the entire(integrated) system

Pros and Cons

- Need of Drivers
- Need of Stubs
- Number of Integration Cycles/Sessions
 - Sessions = nodes – leaves + edges
 - Number of stubs or drivers required
- Number of integration test cases and their runs
- Detection of defects
 - Early
 - Separation of defective module/component

Management driven processes versus engineering
and development driven process

Comparison between DBI

Category	BU	TD	BB	S
Drivers	Yes	No	No	In-part
Stubs	No	Yes	No	In-part
Early Working version of the system	Late	Early	Late	Early
Early detection of interface errors	Early	Early	Late	Early
Parallel Testing	Medium	Low	High	Medium

Note: Individual testing of drivers and stubs will need to be considered separately

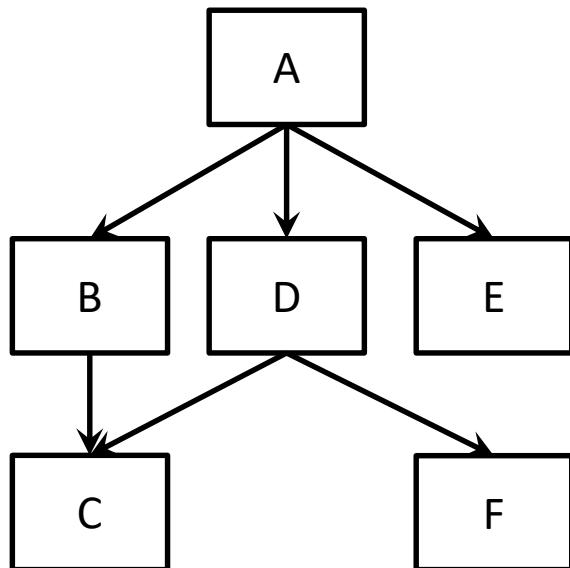
Call Graph-Based Integration

Call Graph-Based
Integration

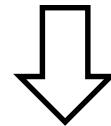


Neighbourhood

- One edge away
- Predecessor and Successor



Interior nodes = nodes – (source nodes + sink nodes)
 Neighbourhoods = interior nodes + source nodes



Neighbourhoods = nodes – sink nodes

Pros and Cons

- Move away from purely structural basis to behavioural basis
- Number of Integration Cycles/Sessions
- Number of integration test cases and their runs
- Detection of defects
 - Early
 - Separation of defective module/component is at times a challenge

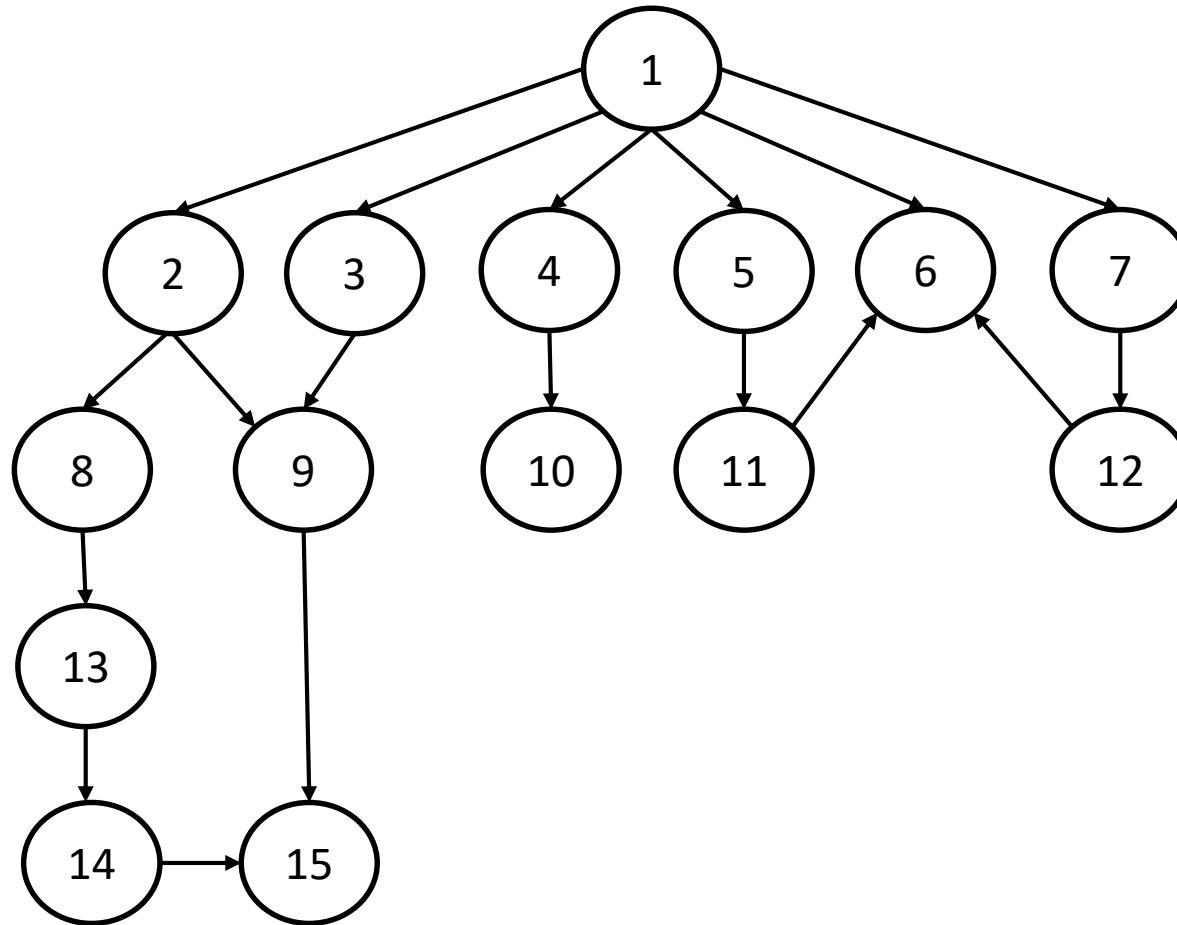
Path-Based Integration

- Desirable → Structural and functional testing
- Express testing in terms of behavioural threads
- Focus on interaction (instead of interfaces)
 - Co-functioning
 - Interfaces are structural
 - Interactions is behavioural



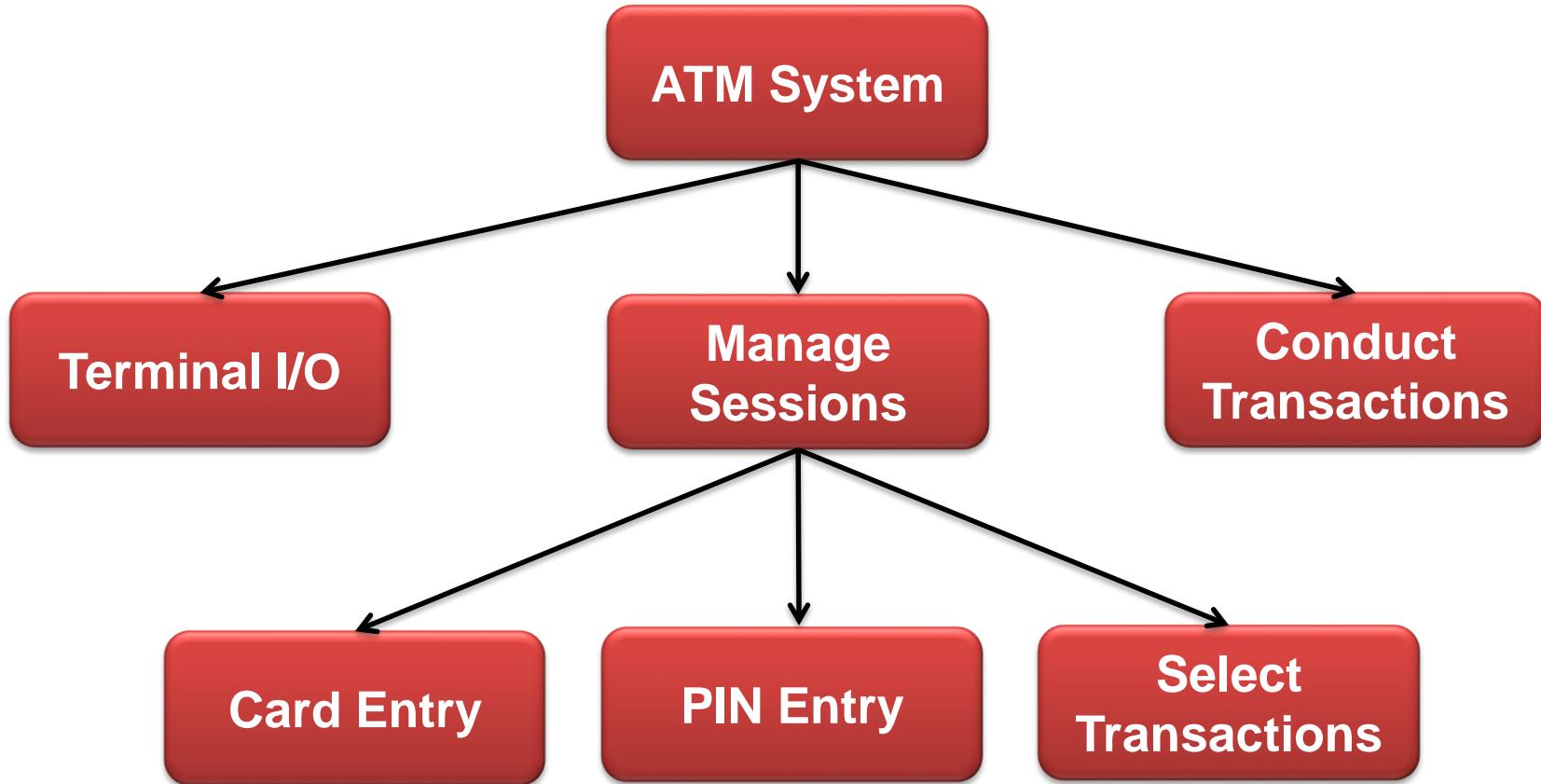
Topic 11.3 : Examples

A Sample Program



Refer: Page 239 of T1

Simple ATM



Source: T1: Figure 12.2

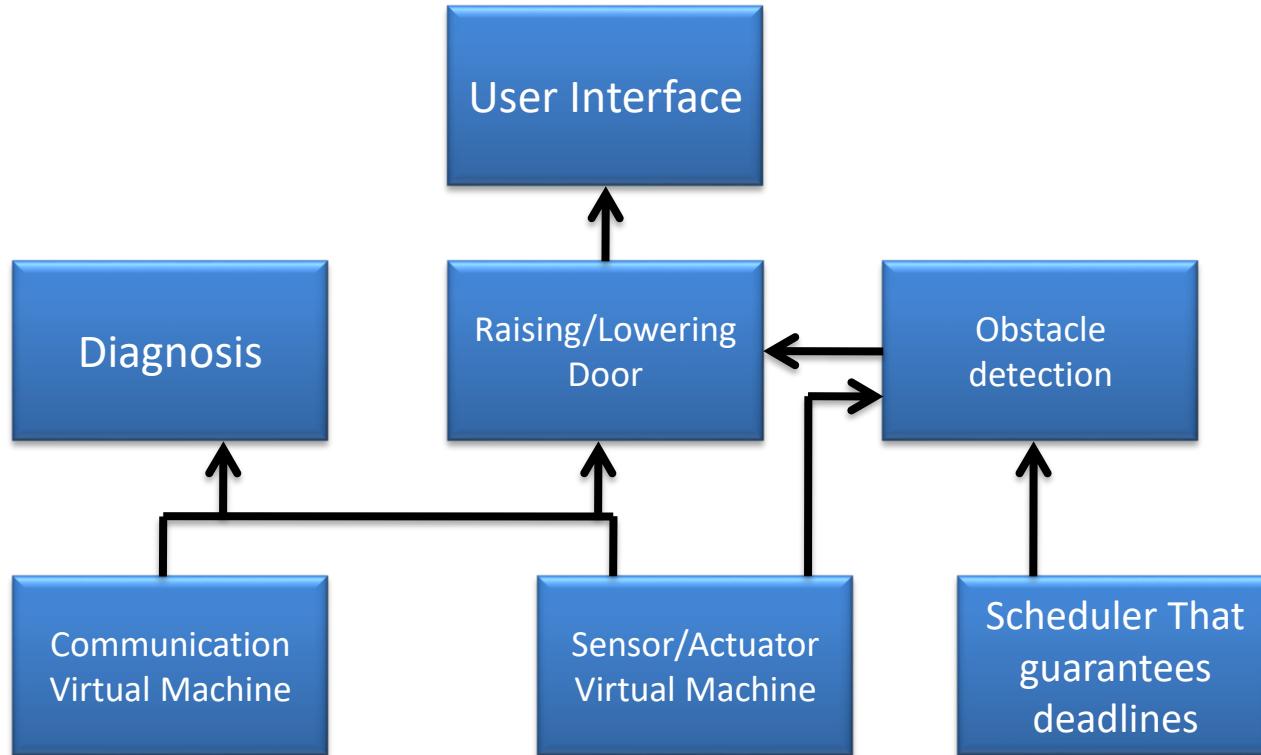


Topic 11.4 : Cases

Garage Door

- The device controls for opening and closing the door are different for the various product lines. They may include controls from within a home information system. The product architecture for a specific set of controls should be directly derivable from the product line architecture
- The processor used in different processes will differ. The product architecture for each specific processor should be directly derivable from the product line architecture
- If an obstacle (person or object) is detected by the garage door during descent, it must halt (alternately re-open) within 0.1 second
- The garage door opener should be accessible for diagnosis and administration from within the home information system using product specific diagnosis protocol. It should be possible to directly produce an architecture that reflects this protocol

Garage Door

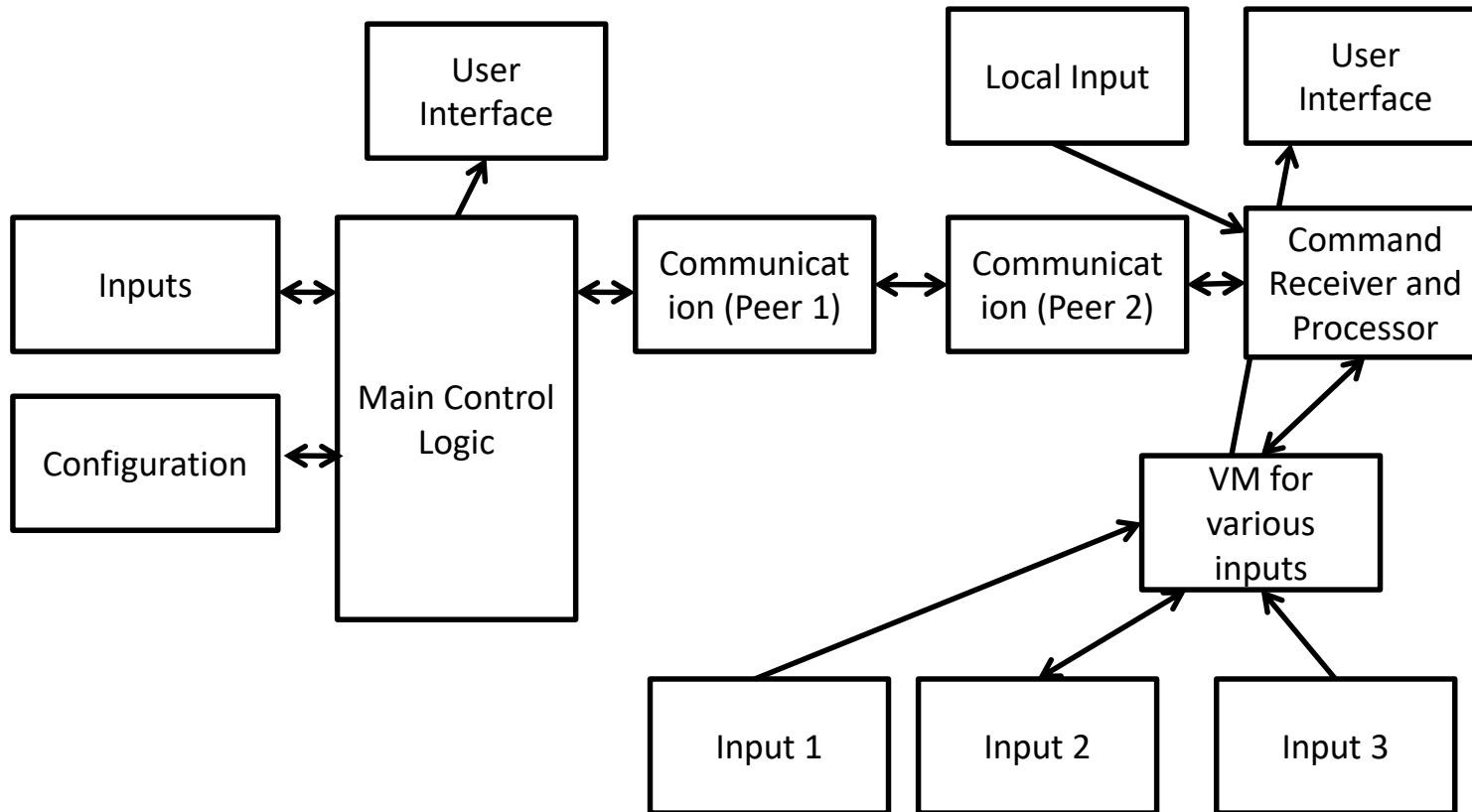


Building Lighting Control System



- Lighting control system for a apartment complex
- Emergency lights are UPS powered
- Corridor lights are timed and based on ambient light
- Garden and Architectural lights are timed

Building Lighting System



Module 11 Self Study

- SS11.1 Take up a program (like currency converter) along with a GUI. Develop integration strategies and compare them or various attributes. Comment on use of various testing techniques for integration testing



SS 11.1 Integration Strategies

11.1 Self Study

To explore:

- Various strategies of integration and their effectiveness for testing software
- Use of testing techniques for integration testing

Study Work:

- You will need a modular program (either design and develop your self or use from open source) like currency converter
- Based on the modules; review and apply various integration strategies. Compare and contrast.
- Review various testing techniques to accomplish integration testing



SS 9.2 Object Oriented UI frameworks study

9.2 Self Study

To explore:

- Application of testing techniques to OO software
- List the limitations that you come across

Study Work:

- Take up either GTK or QT as frameworks to study. Summarise their design. Explore the application of testing techniques to test the framework as well as applications developed using the framework
- Compare and contrast the effectiveness of the techniques to test the System

System Testing

- System testing is concerned with the behaviour of a whole system. The majority of the functional failures should already have been identified during unit and integration testing
- System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

System Testing

- It is essential to define what the “system” is made up of
 - A system for one team/organisation may mean a module/part of a larger system
- Generally system level is treated as closest to everyday experience i.e. system (Product) as seen in the hands of the user
- In system testing apart from finding faults the goal is to demonstrate
 - System works
 - System is reliable
 - System is durable
 - Quality of system for defined parameters
- Use of specification based testing techniques to design test cases

System Testing – Aspects

- Thread
 - Thread Possibilities
 - Thread Definitions
 - Atomic Thread Function

System Requirements – Basic Concepts



- Data
 - Information used and created by the system
- Actions
 - Transform, data transform, control transform, process, activity, task, method, service
- Devices
 - Source and destination of system level inputs and outputs
- Events
 - System level I/O; occurs on port device; inputs and outputs of actions
- Threads
 - Model of a system – interactions among data, events and action

Concepts

- Relationship among basic concepts
 - Interrelation among data, actions, devices, events and threads
- Modeling with Basic Concepts
 - Structural
 - Behavioural
 - Contextual



Structural Strategies for Thread Testing

- Bottom Up Threads
- Node and Edge Coverage Metrics

Functional Strategies for Thread Testing

- Event-based Thread Testing
- Port-based Thread Testing
- Data-based Thread Testing

Testing – In action

- Process View
- Progression Vs. Regression

Test Plan

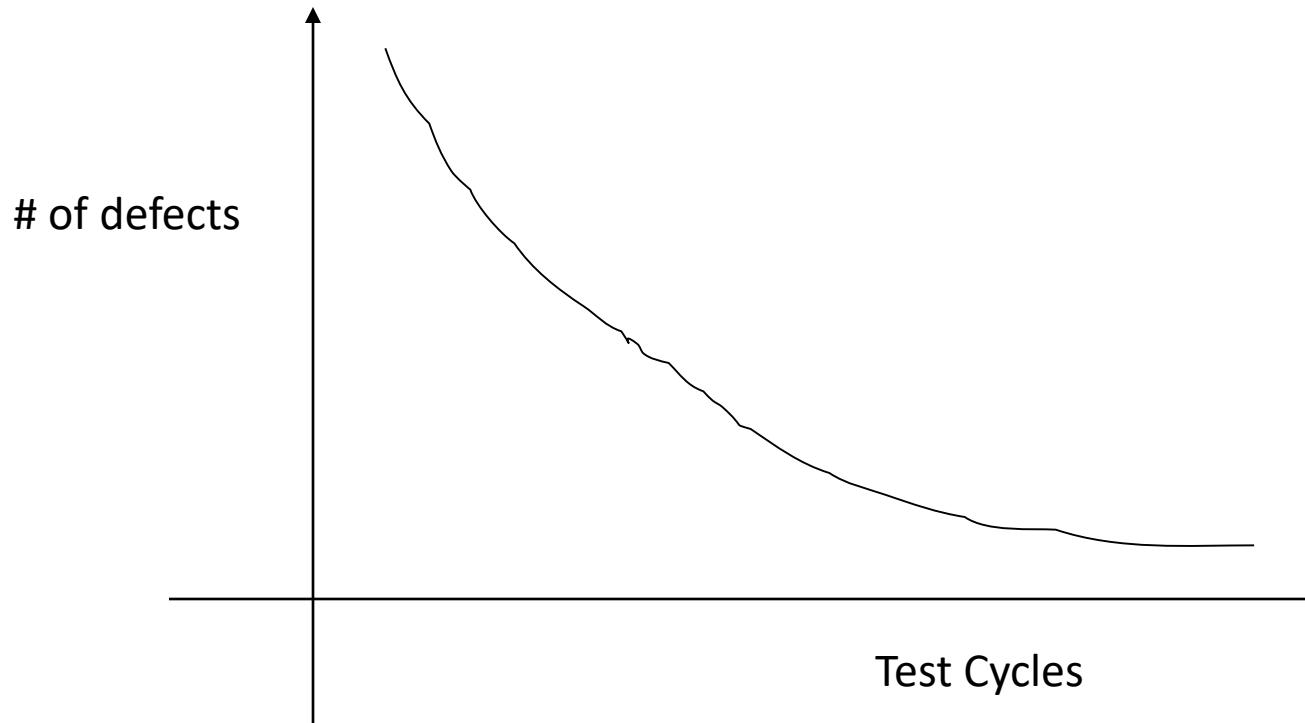
- IEEE Test Plan (Review the paper)
- Test Cycles
- Test Cases per cycle
- Test Execution Report
- Test Automation
- Test Effort Estimation
 - Design
 - Development
 - Execution

Reliability Analysis

- Reliability theory is concerned with determining the probability, that a system, possibly consisting of many components will function.
- Overall, System components can be treated in configurations
 - Series System: System will function if all components are functioning
 - Parallel System: System will function if one of the component is functioning. (Provided the ultimate aim is to have the same component working).
 - Composite System: (Series + Parallel) in varying configurations

Reliability Analysis

- This study gives us the mean life time of a system
- Analysis requires us to look at the system when failed components are subjected to repair



Module 12: Agenda

Module 12: System Testing

Topic 12.1

System Testing- Introduction, Overview & Issues

Topic 12.2

System Testing – Types, Techniques & Strategies

Topic 12.3

Examples



Topic 12.1: System Testing- Introduction, Overview & Issues

System Testing

- System testing is concerned with the behaviour of a whole system. The majority of the functional failures should already have been identified during unit and integration testing
- System testing is usually considered appropriate for comparing the system to the non-functional system requirements, such as security, speed, accuracy, and reliability. External interfaces to other applications, utilities, hardware devices, or the operating environment are also evaluated at this level.

System Testing

- It is essential to define what the “system” is made up of
 - A system for one team/organisation may mean a module/part of a larger system
- Generally system level is treated as closest to everyday experience i.e. system (Product) as seen in the hands of the user
- In system testing apart from finding faults the goal is to demonstrate
 - System works
 - System is reliable
 - System is durable
 - Quality of system for defined parameters
- Use of specification based testing techniques to design test cases

System Testing – Aspects

- Thread
 - Thread Possibilities
 - Thread Definitions
 - Atomic Thread Function

System Requirements – Basic Concepts



- Data
 - Information used and created by the system
- Actions
 - Transform, data transform, control transform, process, activity, task, method, service
- Devices
 - Source and destination of system level inputs and outputs
- Events
 - System level I/O; occurs on port device; inputs and outputs of actions
- Threads
 - Model of a system – interactions among data, events and action



Topic 12.2: System Testing – Types, Techniques & Strategies

Concepts

- Relationship among basic concepts
 - Interrelation among data, actions, devices, events and threads
- Modeling with Basic Concepts
 - Structural
 - Behavioural
 - Contextual



Structural Strategies for Thread Testing

- Bottom Up Threads
- Node and Edge Coverage Metrics

Functional Strategies for Thread Testing

- Event-based Thread Testing
- Port-based Thread Testing
- Data-based Thread Testing

Testing – In action

- Process View
- Progression Vs. Regression



Test Plan

Test Plan

- IEEE Test Plan (Review the paper)
- Test Cycles
- Test Cases per cycle
- Test Execution Report
- Test Automation
- Test Effort Estimation
 - Design
 - Development
 - Execution

Reliability Analysis

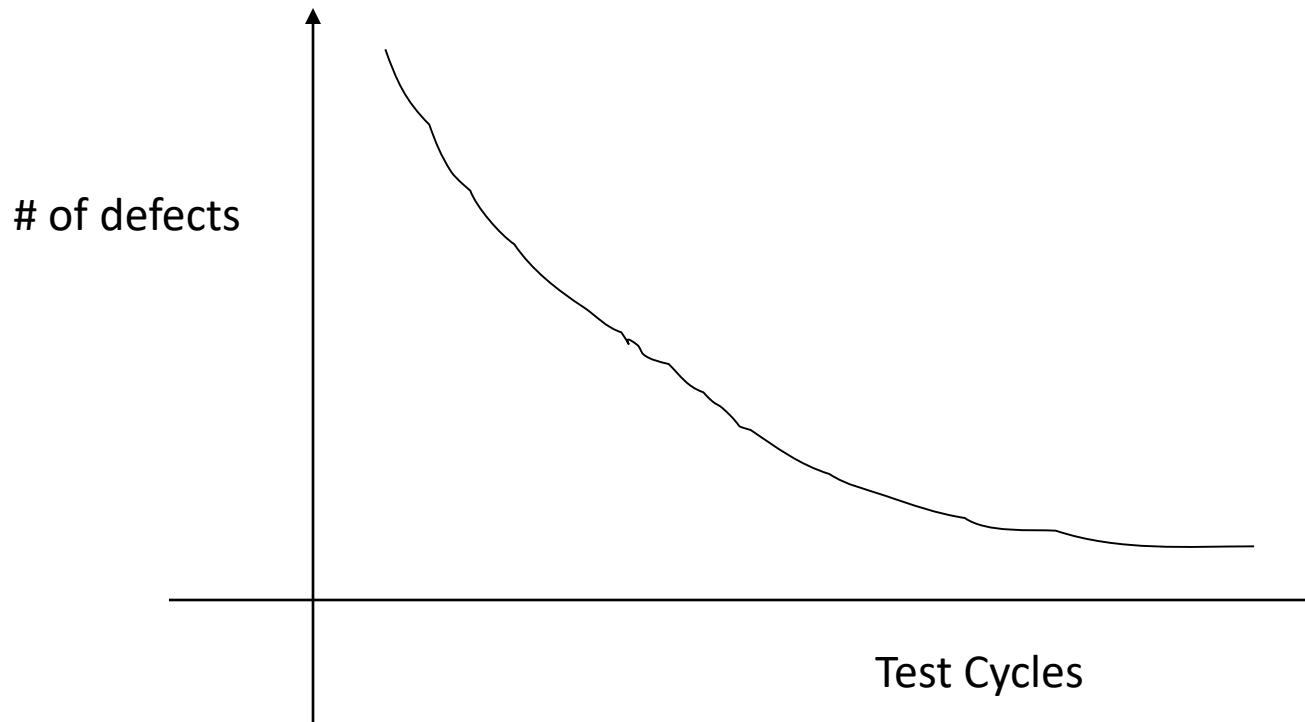
- Reliability theory is concerned with determining the probability, that a system, possibly consisting of many components will function.
- Overall, System components can be treated in configurations
 - Series System: System will function if all components are functioning
 - Parallel System: System will function if one of the component is functioning. (Provided the ultimate aim is to have the same component working).
 - Composite System: (Series + Parallel) in varying configurations

Reliability Analysis

- Each component will function with some known probability (independent of each other)
- Upper and lower bounds of probability
- A component may show a dynamic behaviour over time i.e. it functions for a random length of time and after which it fails
- Amount of time a system functions is related to the distribution of a component lifetimes
- In particular, it turns out that if the amount of time that a component functions has an increasing failure rate on average distribution, then so does the distribution of system lifetimes

Reliability Analysis

- This study gives us the mean life time of a system
- Analysis requires us to look at the system when failed components are subjected to repair



Reliability Analysis

- Mean Time to Failure (MTTF) = Total Testing time/# of defects

$$MTTF = T/d$$

- Failure Rate

$$F_a = \frac{1}{MTTF}$$

- Mean Time to Repair (MTTR) = Sum(|Time of detection of fault – time of closure|)/Total # of defects

$$MTTR = \sum (T_d - T_c)/d$$

- Availability

$$A = MTTF/(MTTR + MTTF)$$

Reliability Analysis

- Problem: A SW component was tested for a period of 3 months. The total number of defects found were 300. On an average the time required to fix the defect was 2.5 days. Find the availability of the system
- Solution:

$$MTTF = (3 * 30 * 8) / 300 = 2.4$$

$$F_a = 1 / 2.4 = 0.4166$$

$$MTTR = (2.5 * 8) = 20$$

$$\text{Availability } A = MTTF / (MTTF + MTTR) = \\ 0.107 \sim 10.7\%$$

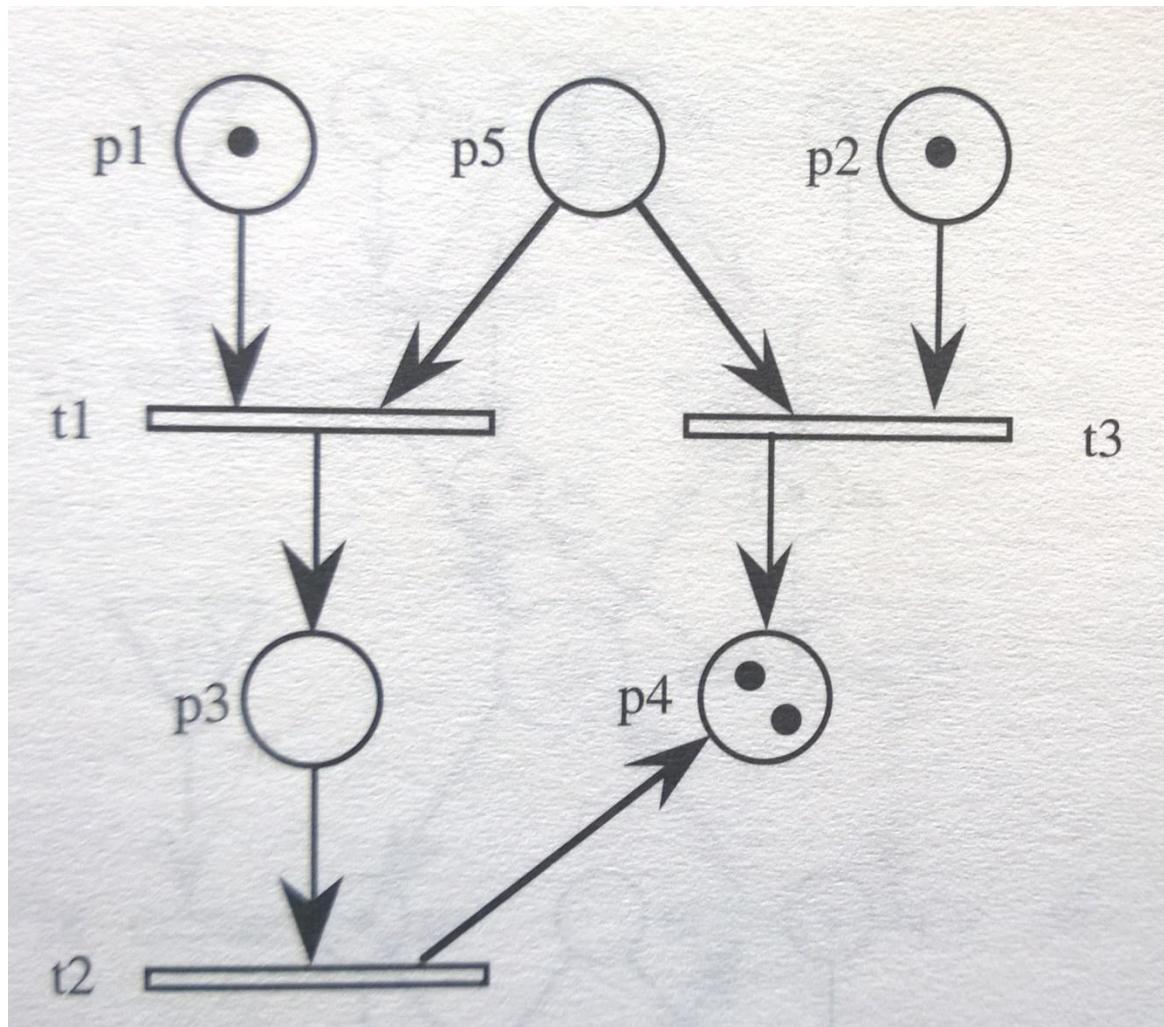


Petri Nets & System Test

Petri Net

- Carl Adam Petri 1963
- Accepted Model for protocol and other application involving concurrency and distributed processing
- Form of directed graph: a bipartite directed graph
- A bipartite graph has two sets of nodes V_1 and V_2 and a set of edges E , with the restriction that every edge has its initial node on one of the sets V_1 , V_2 and its terminal node in the other set
- In Petri net one set is Places and the other is Transitions

A Marked Petri Net



Event Driven Petri Net

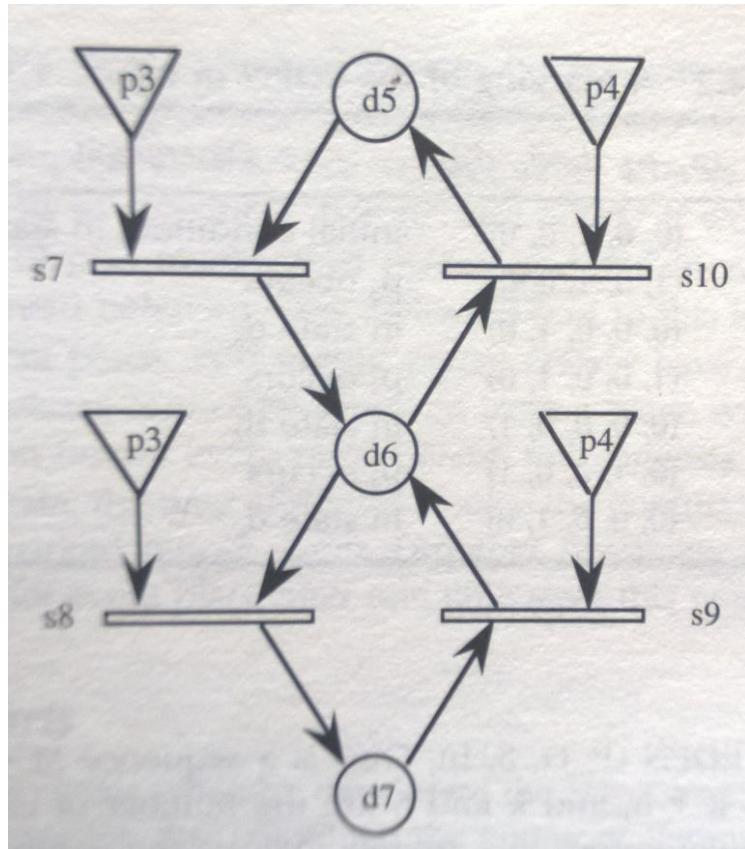
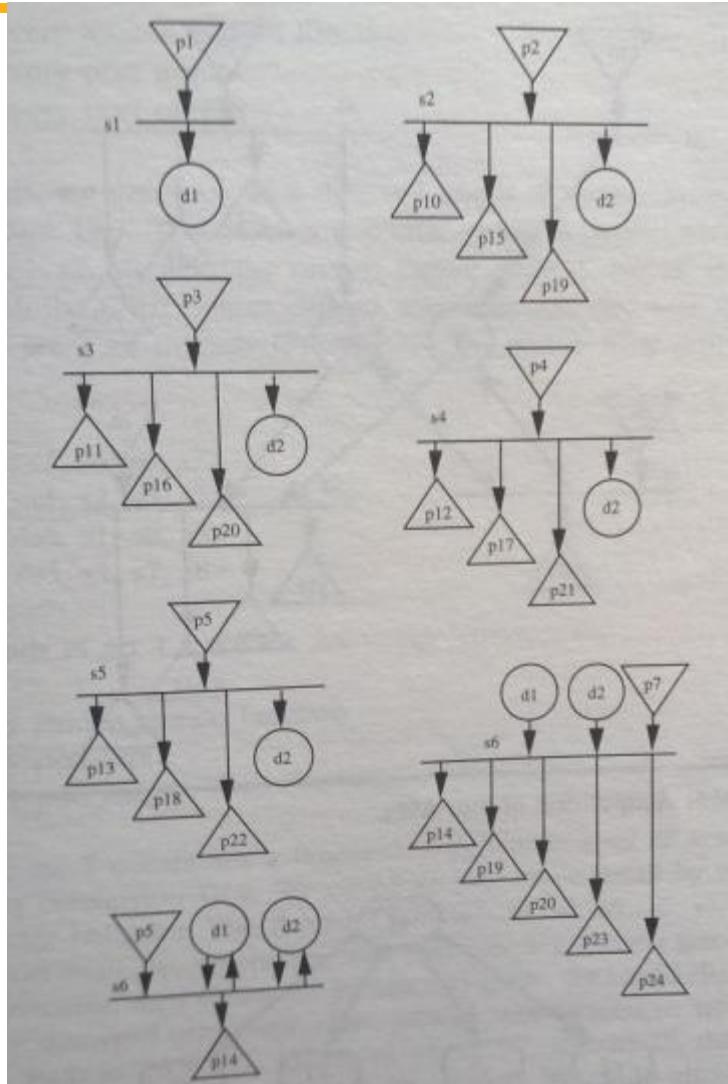


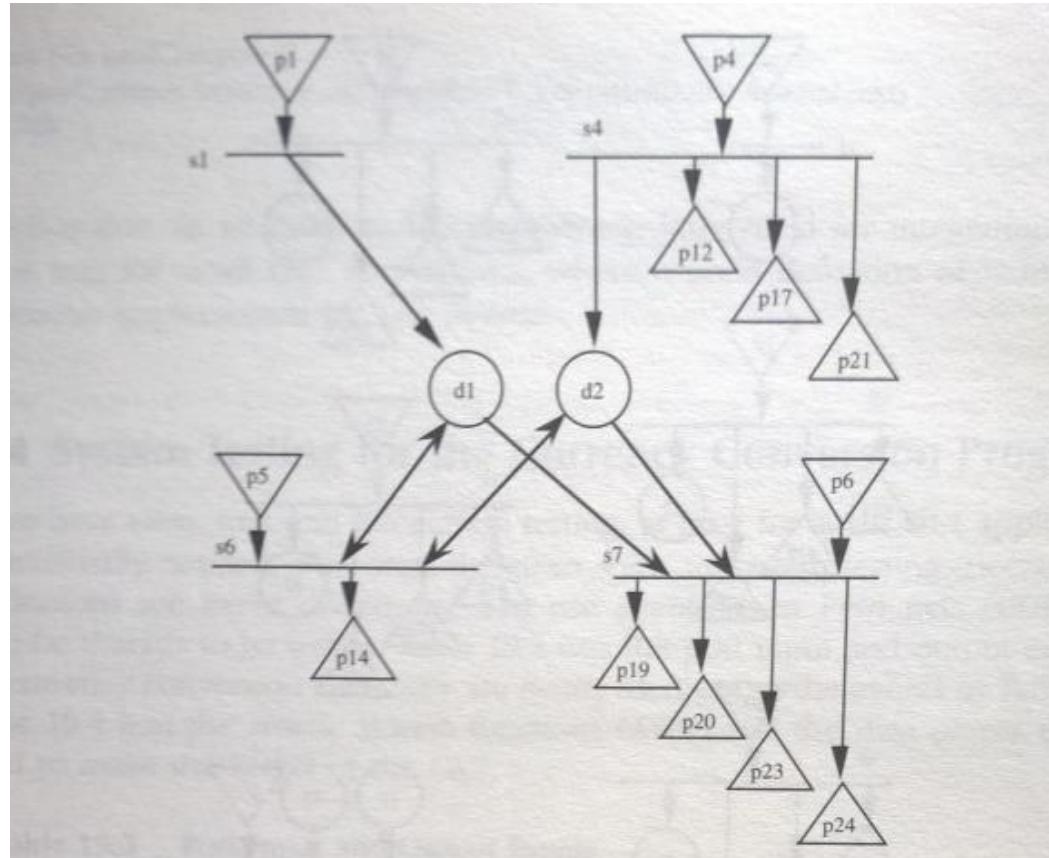
Table 4.1 EDPN Elements in Figure 4.10

Element	Type	Description
p_3	port input event	rotate dial clockwise
p_4	port input event	rotate dial counterclockwise
d_5	data place	dial at position 1
d_6	data place	dial at position 2
d_7	data place	dial at position 3
s_7	transition	state transition: d_5 to d_6
s_8	transition	state transition: d_6 to d_7
s_9	transition	state transition: d_7 to d_6
s_{10}	transition	state transition: d_6 to d_5

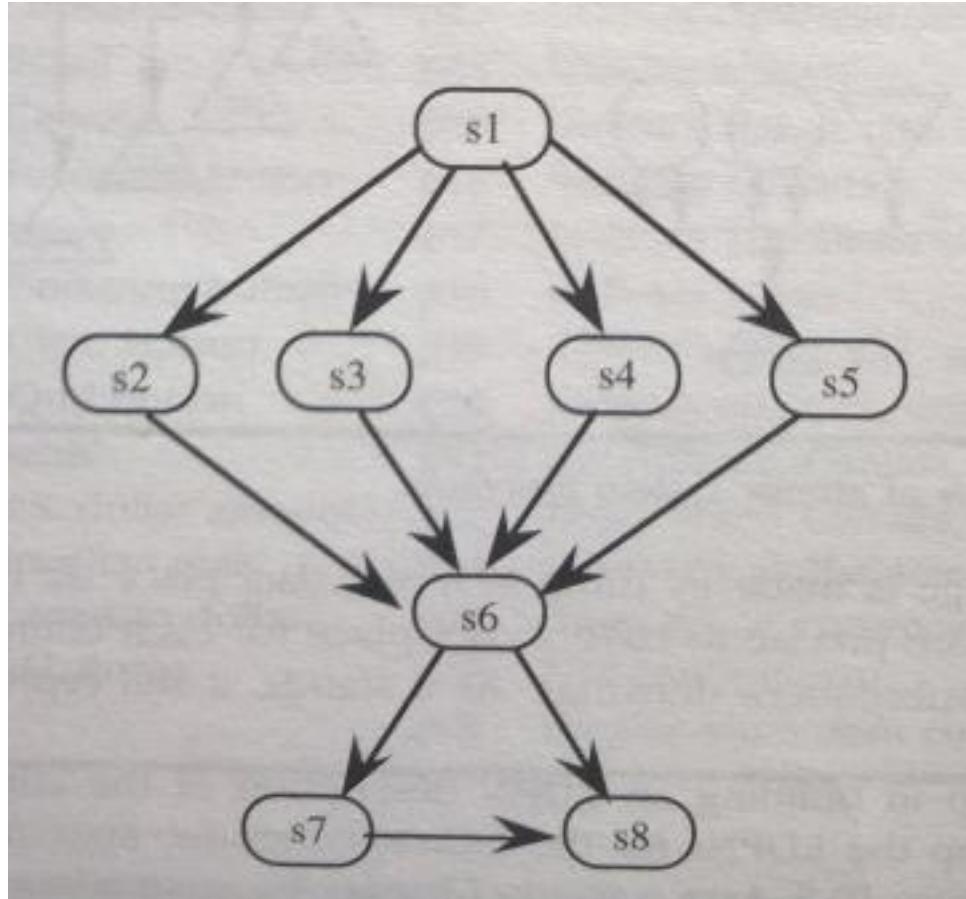
System Testing Currency Converter Program



EPDN Composition of Four ASFs



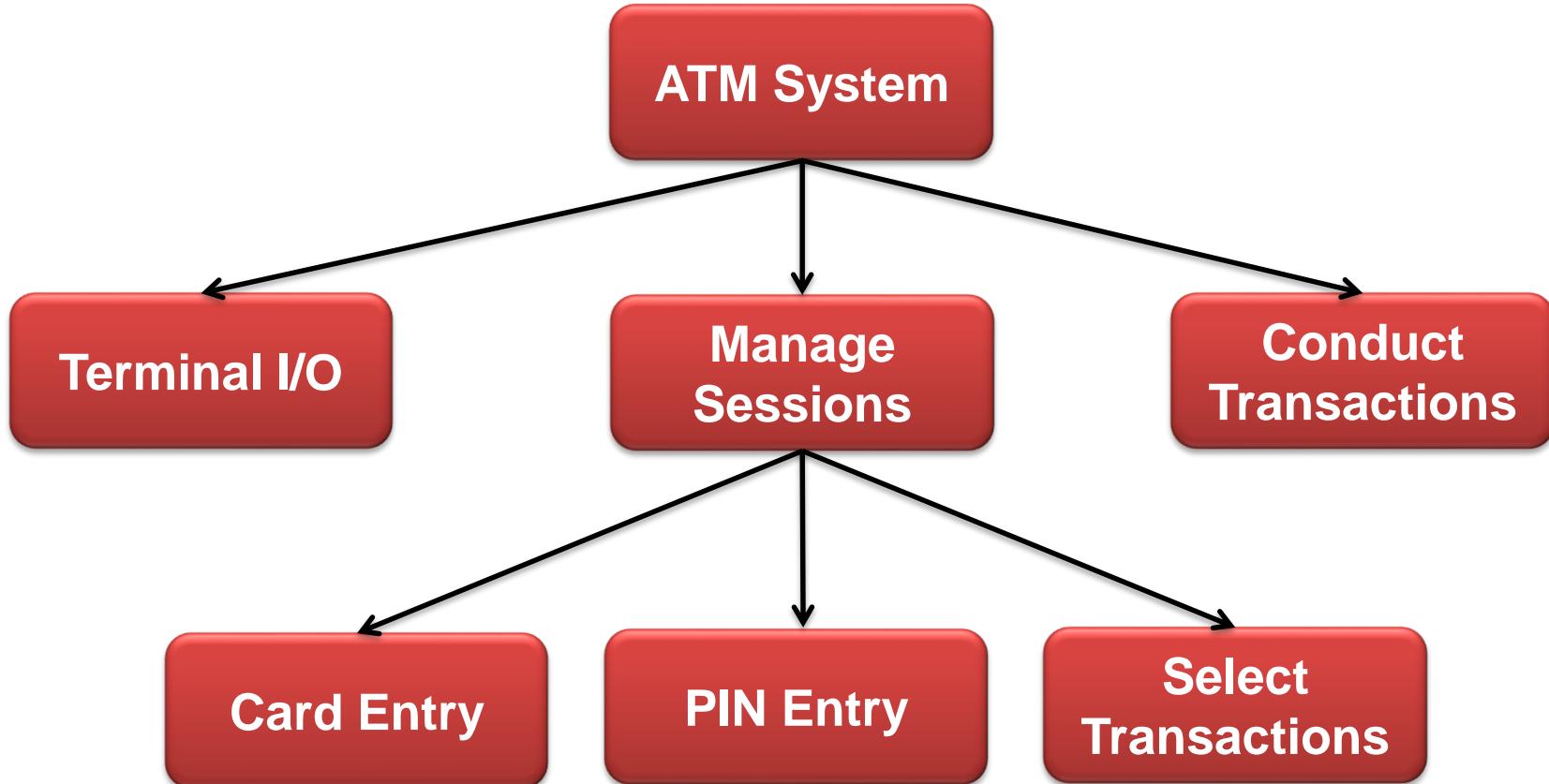
Directed Graph of ASF sequences





Topic 12.3: Examples

Automated Teller Machine

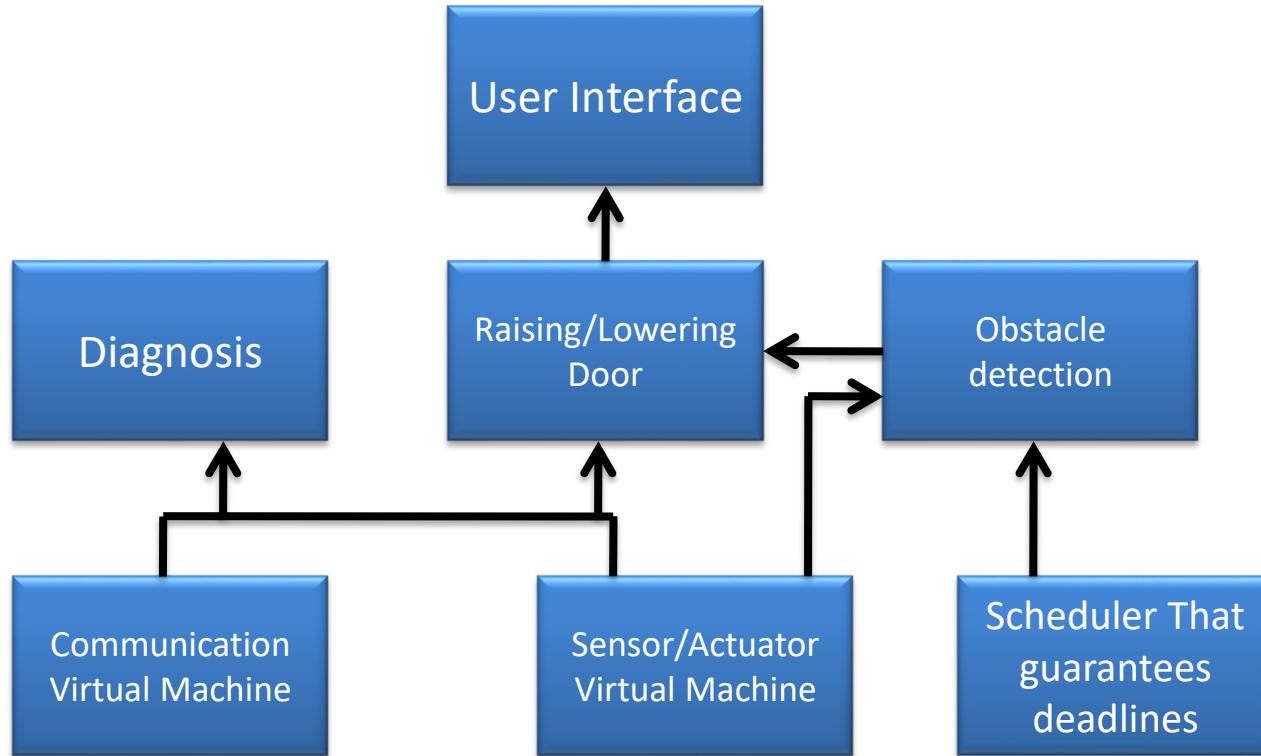


Source: T1: Figure 12.2

Garage Door

- The device controls for opening and closing the door are different for the various product lines. They may include controls from within a home information system. The product architecture for a specific set of controls should be directly derivable from the product line architecture
- The processor used in different processes will differ. The product architecture for each specific processor should be directly derivable from the product line architecture
- If an obstacle (person or object) is detected by the garage door during descent, it must halt (alternately re-open) within 0.1 second
- The garage door opener should be accessible for diagnosis and administration from within the home information system using product specific diagnosis protocol. It should be possible to directly produce an architecture that reflects this protocol

Garage Door

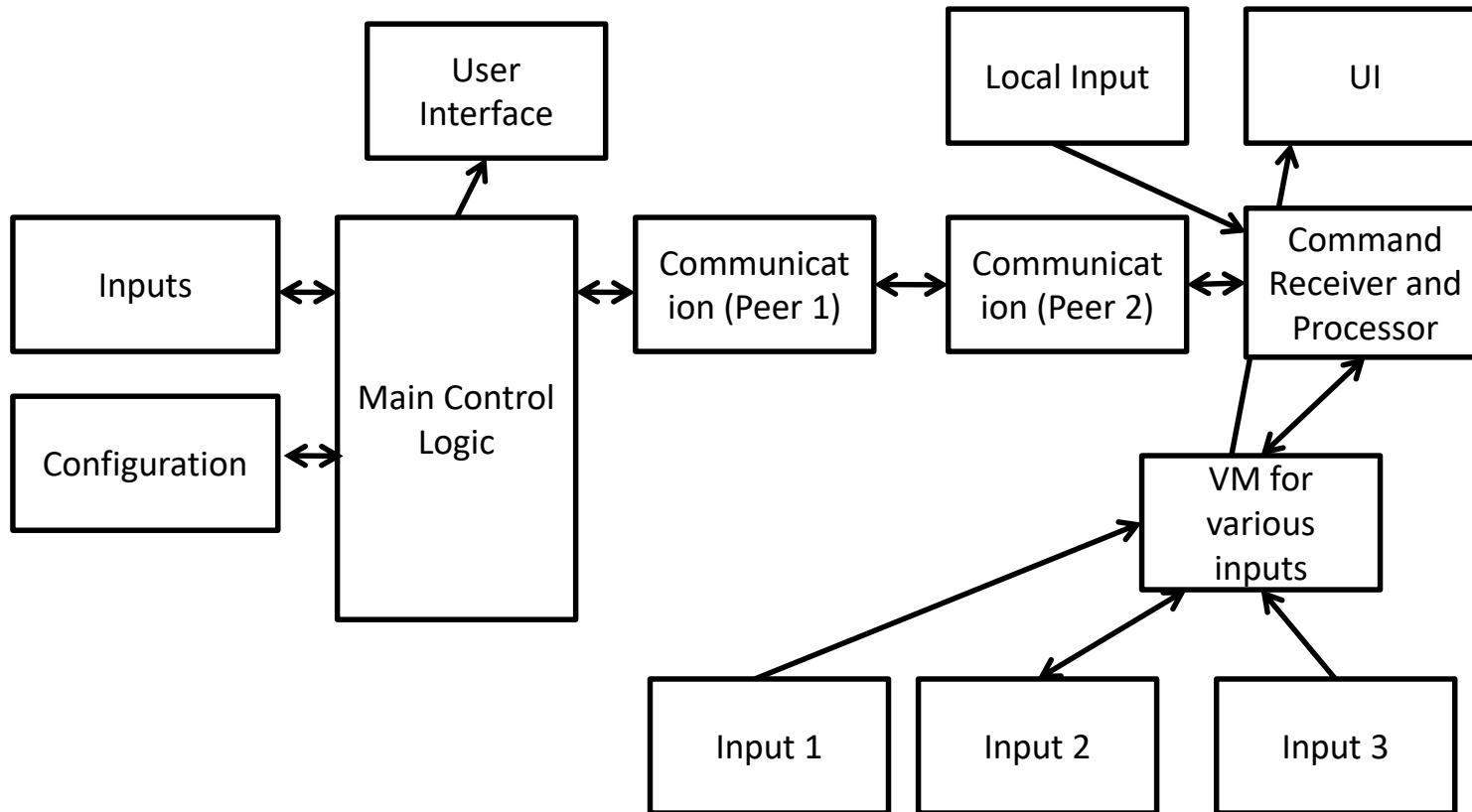


Building Lighting Control System



- Lighting control system for a apartment complex
- Emergency lights are UPS powered
- Corridor lights are timed and based on ambient light
- Garden and Architectural lights are timed

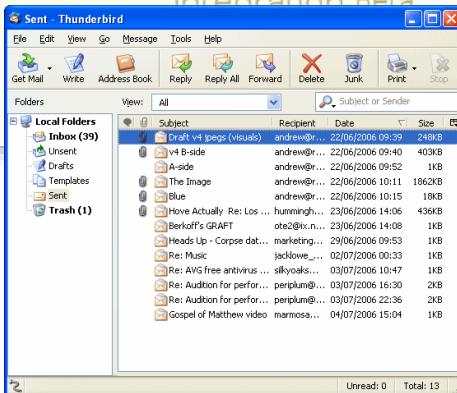
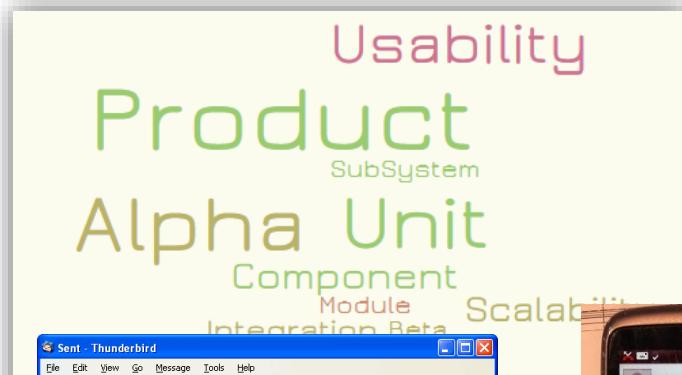
Building Lighting System



Life-Cycle & Model View



- Software Development Life Cycles
- Model Based Testing
- Testing at all levels



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

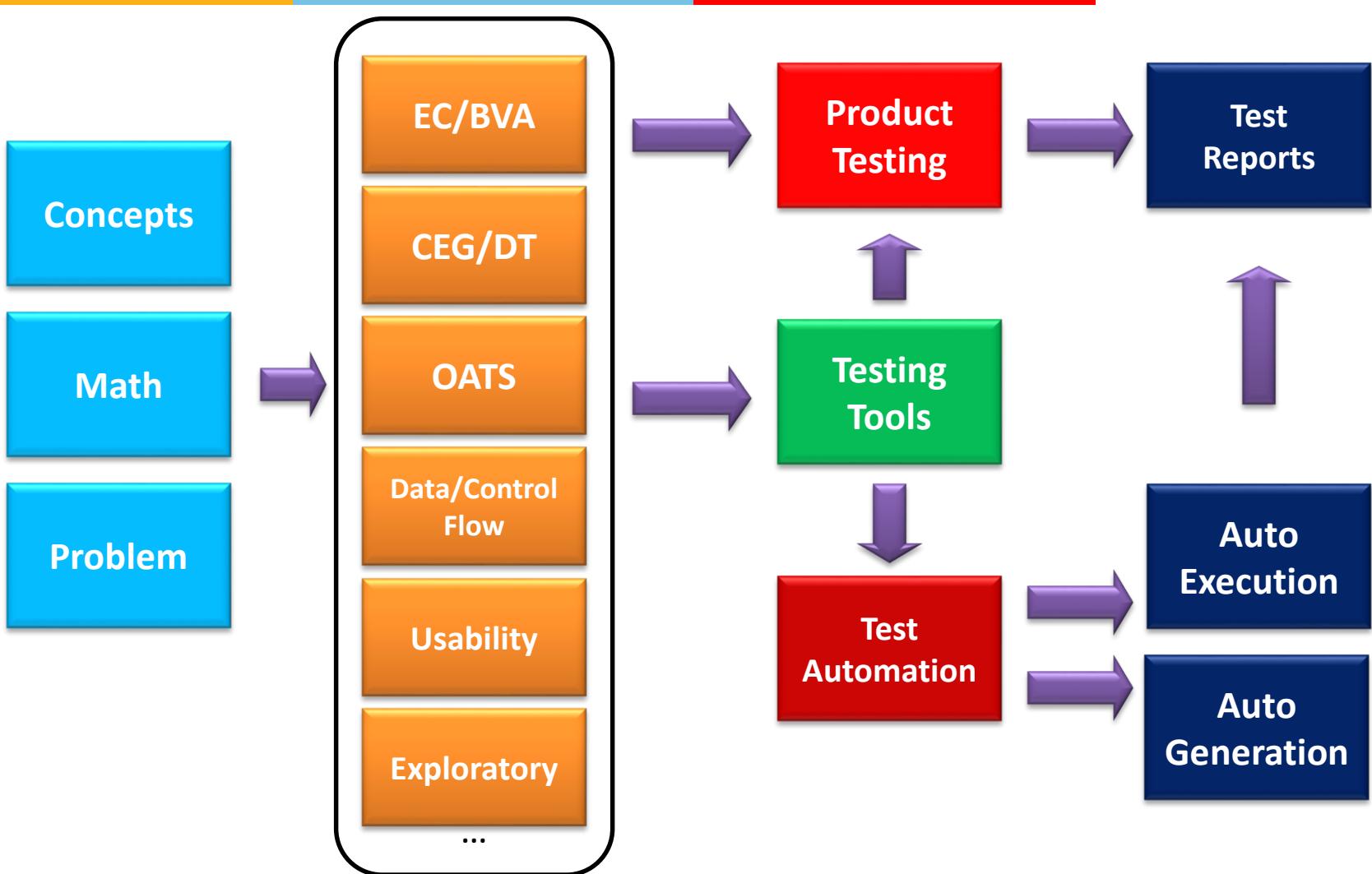
Concepts

SW Arch, Design, Data Structures

Problem

Code coverage
Quality
Zero Defects

Progression



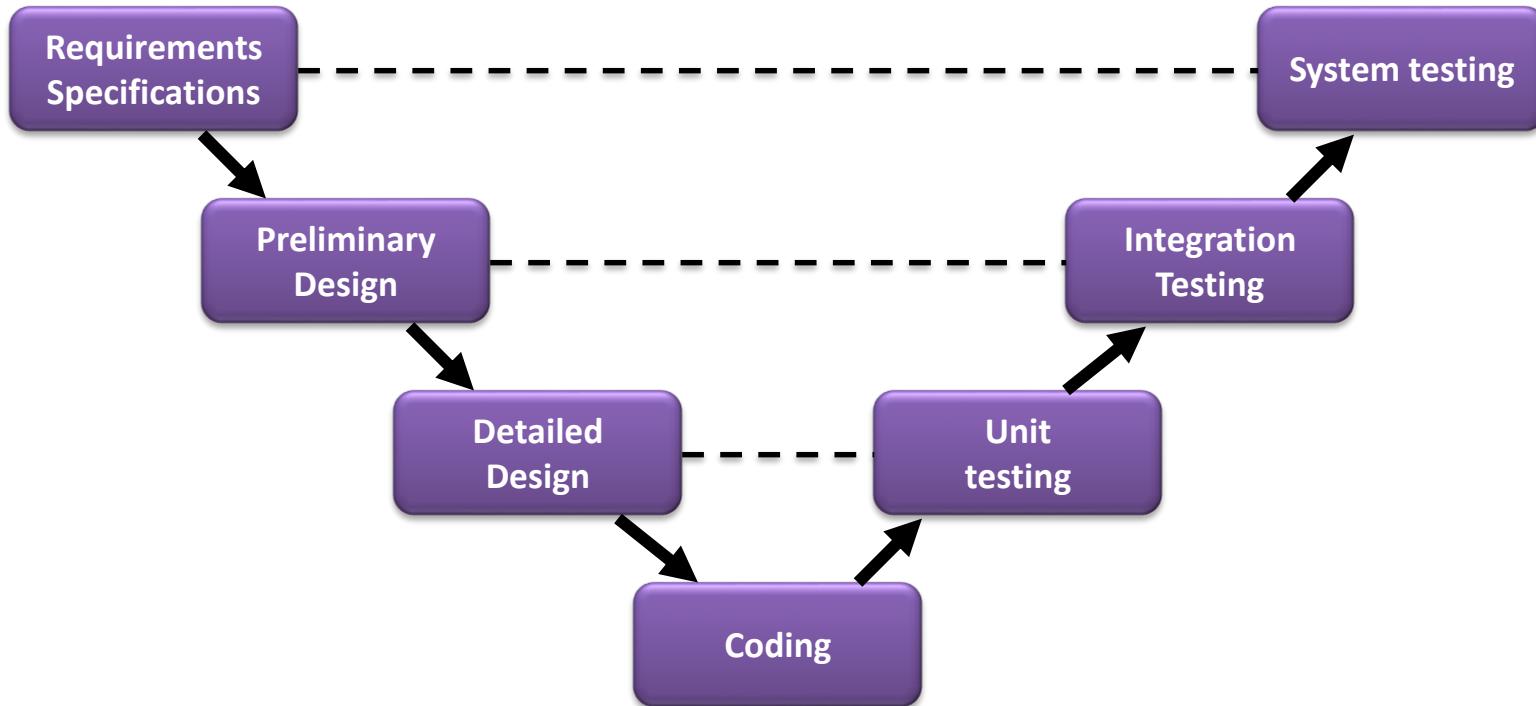
Test Process – Test Activities

- Test Planning
- Test case generation
- Test environment development
- Test Execution
- Test results evaluation
- Problem reporting/Test log
- Defect tracking

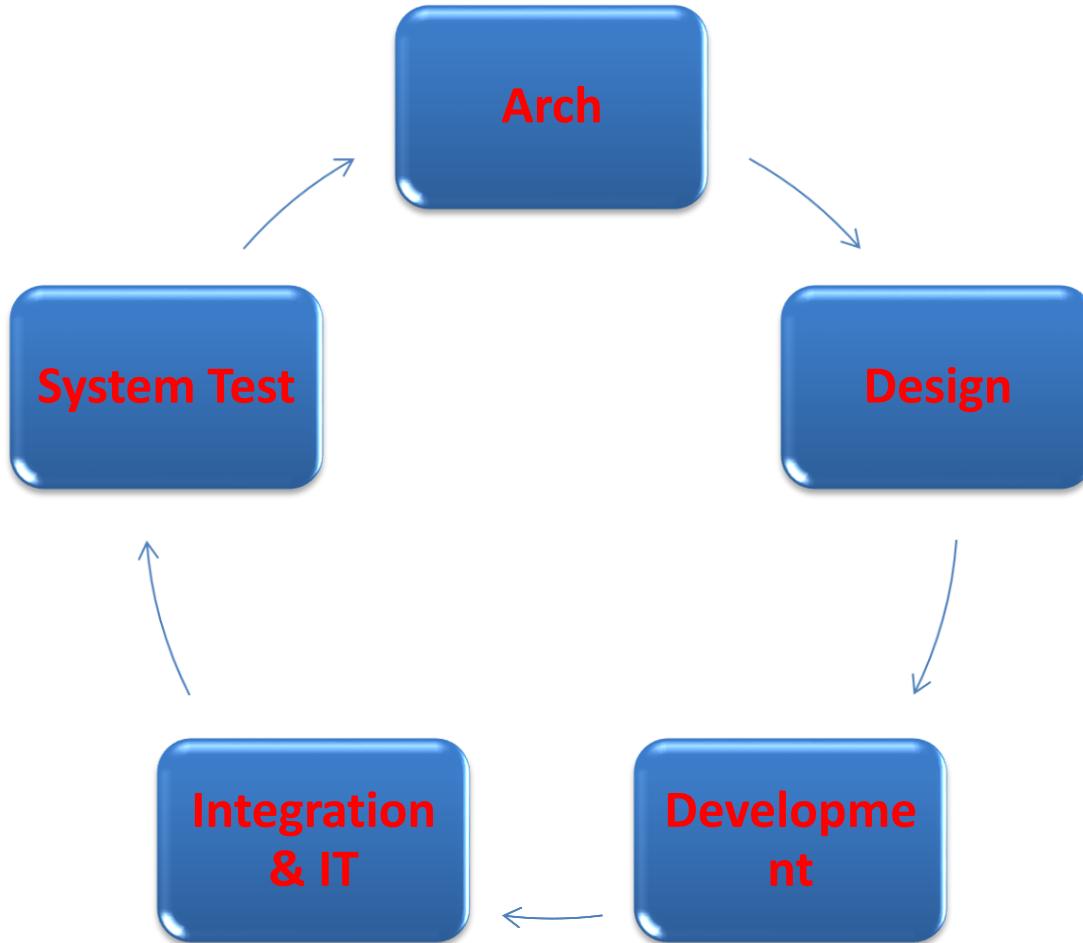
The Waterfall



The Waterfall

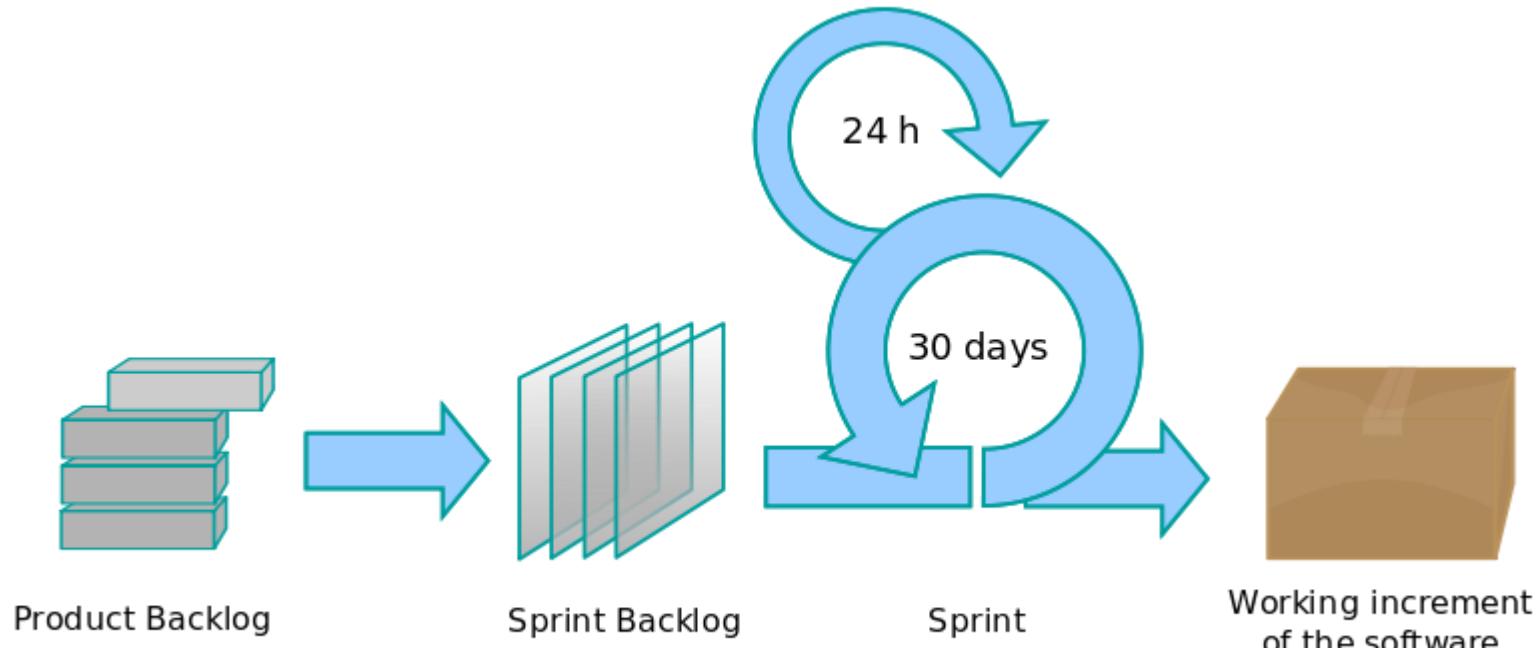


Iterative



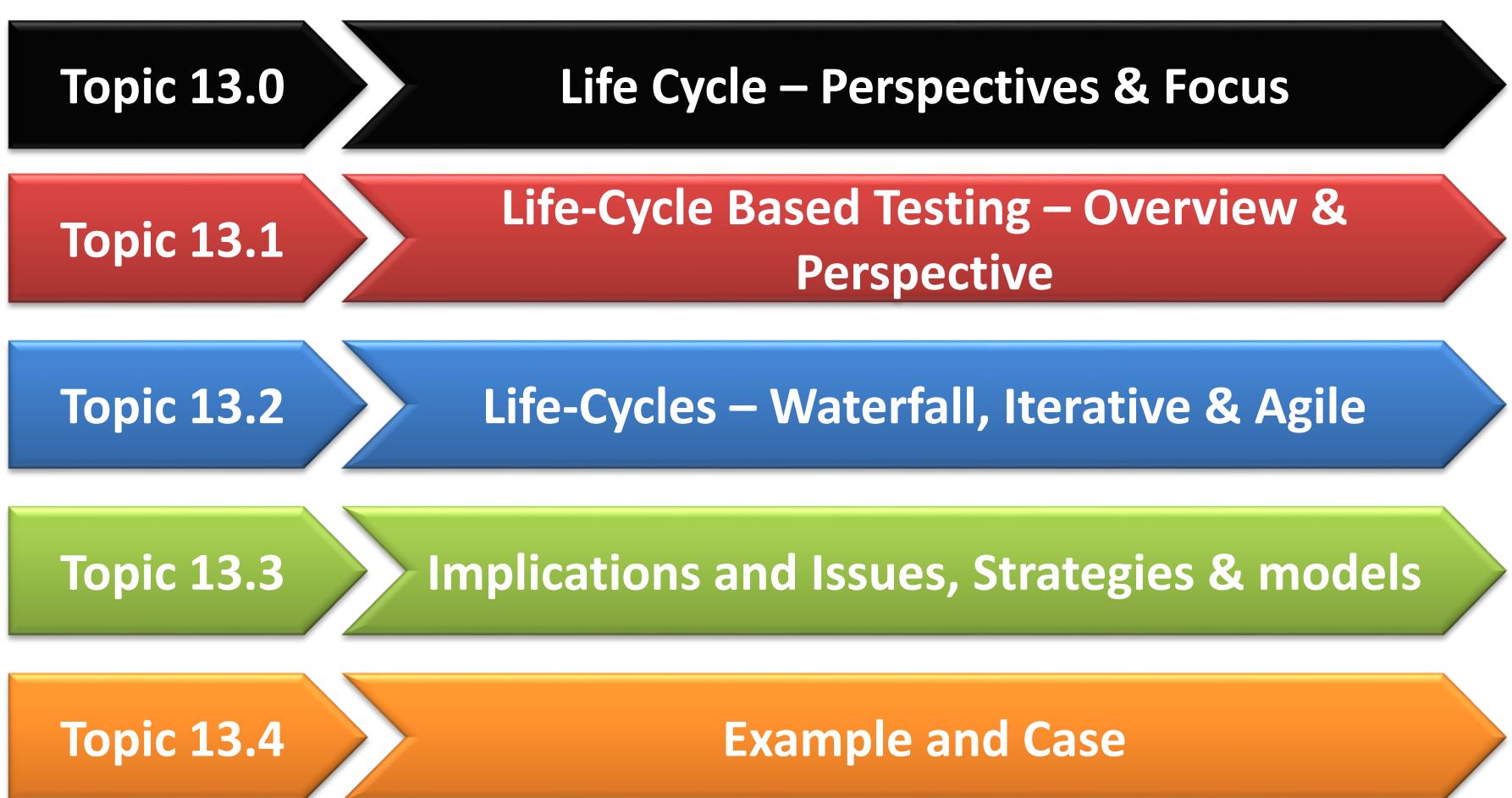
- A View
- A Debate

Scrum



Reference: SCRUM Development Process by Ken Scheweber

Module 13: Agenda





Topic 13.0 Perspectives & Focus

Life-Cycle & Model View

- Software Development Life Cycles
- Model Based Testing
- Testing at all levels



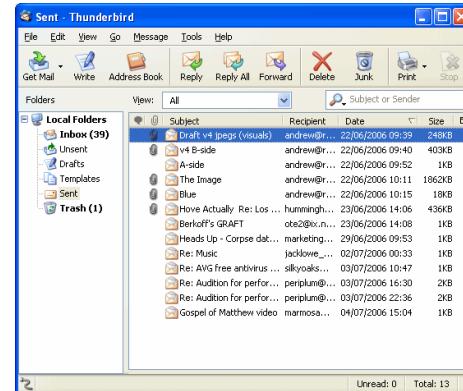
Review & Remember

innovate

achieve

lead

- Levels of Testing
 - Targets
 - Unit
 - Integration
 - System
 - Objectives
 - Acceptance
 - Installation
 - Conformance
 - Functional
 - Correctness
 - Regression
 - Performance
 - Stress
 - ...



104

Review & Remember

- Test Techniques (Our course, Our Focus)
 - Specification-based
 - Code-based
- Based on Nature of Application
 - OO testing
 - GUI
 - ...
- Selecting and Combining Techniques
 - Functional and structural
 - Deterministic vs. Random

Ref: SWEBOK 2004

Review & Remember

- Test Related Measures
 - Evaluation of Program under test
 - Evaluation of the Tests Performed
- Test Process
 - Practical Considerations
 - Test Activities

Ref: SWEBOK 2004



Topic 13.1: Life-Cycle Based Testing – Overview & Perspective

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

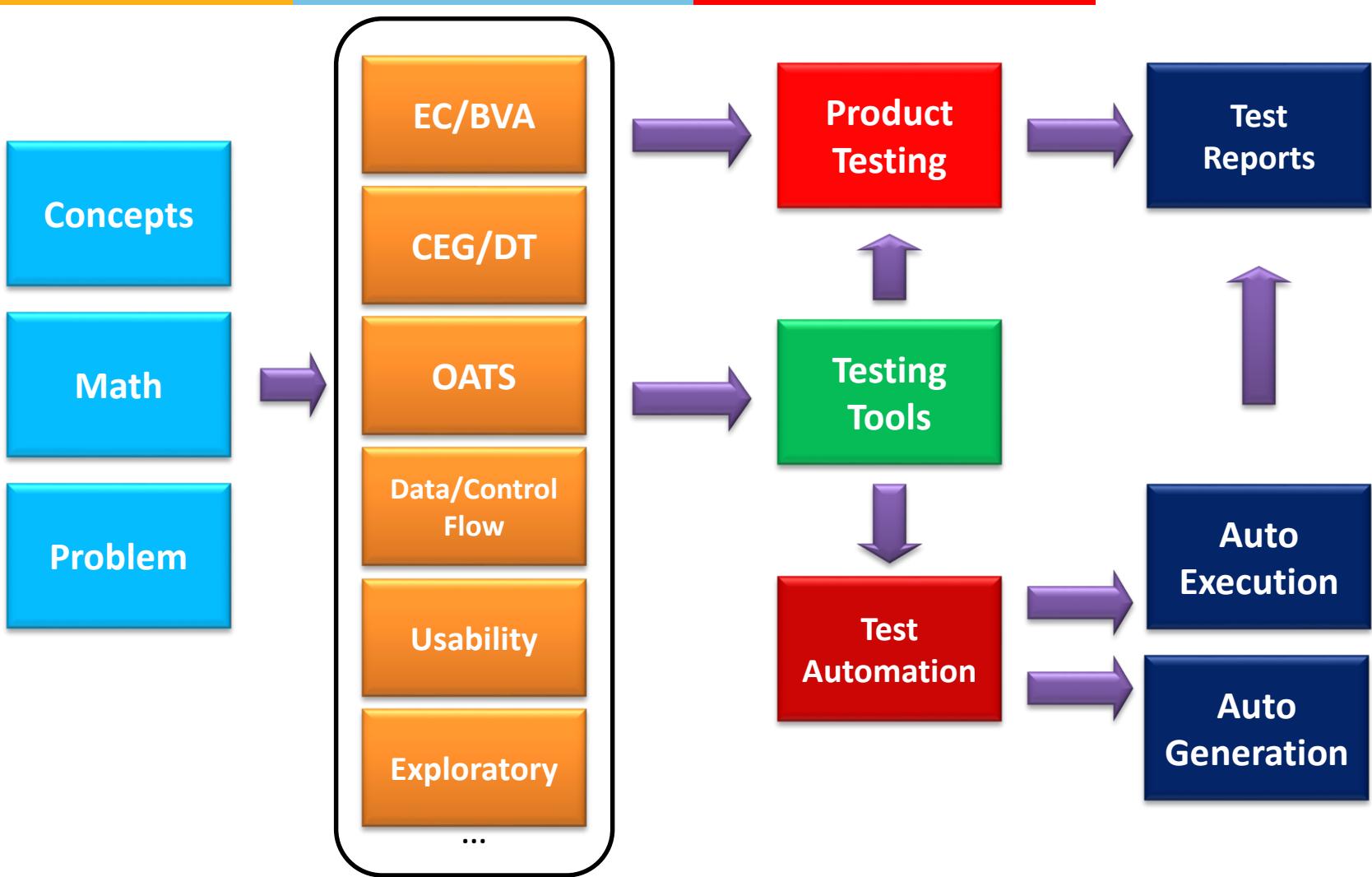
Concepts

SW Arch, Design, Data Structures

Problem

Code coverage
Quality
Zero Defects

Progression



Test Process – Test Activities

- Test Planning
- Test case generation
- Test environment development
- Test Execution
- Test results evaluation
- Problem reporting/Test log
- Defect tracking

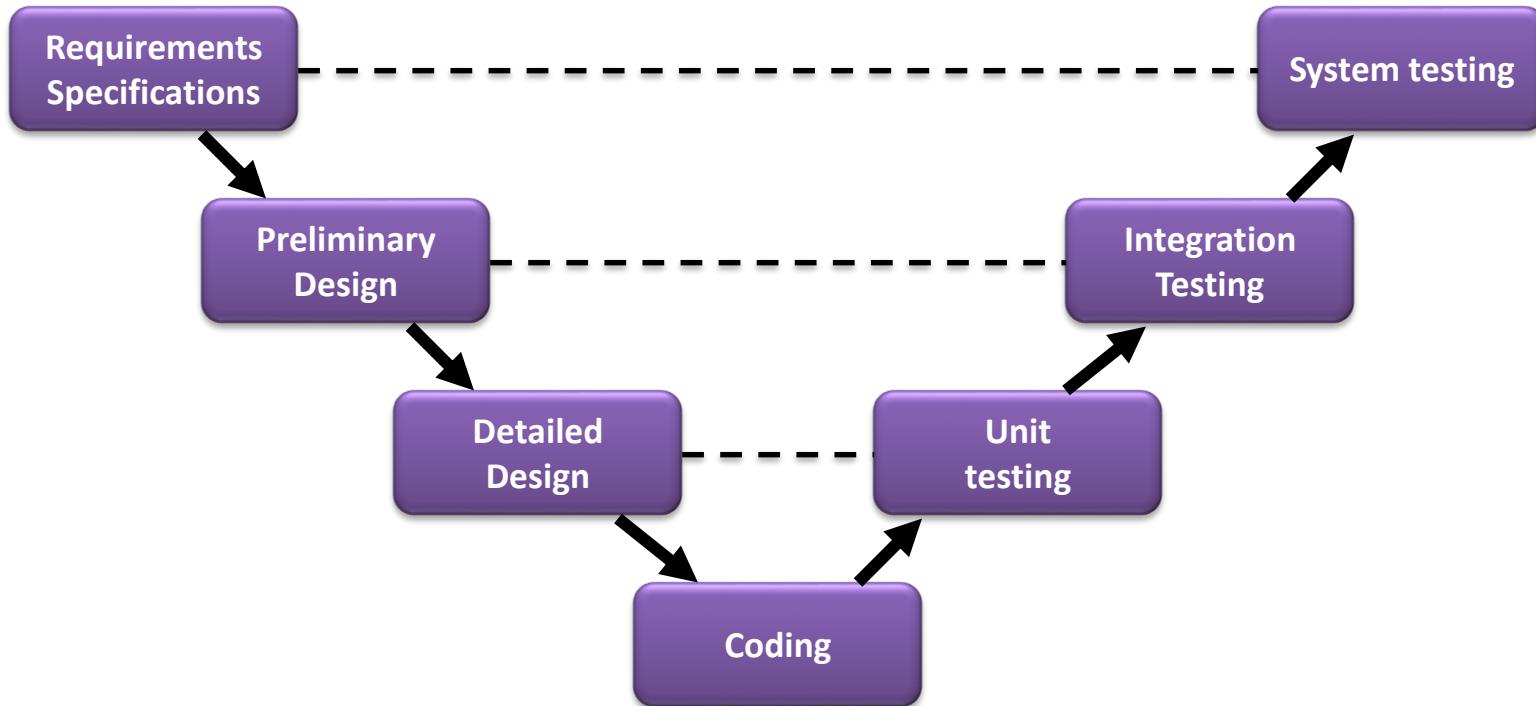
Software Development Life Cycles



Life Cycle Models

- Waterfall
- V-model
- Prototyping
- Waterfall – spin-offs
 - Evolutionary
 - Incremental Model
 - Spiral

The V Model



Please refer to the paper on the taxila

Focus Areas

- People
- Process
- Product
- Technology
- Research
- Business
- Innovation
- Engineering

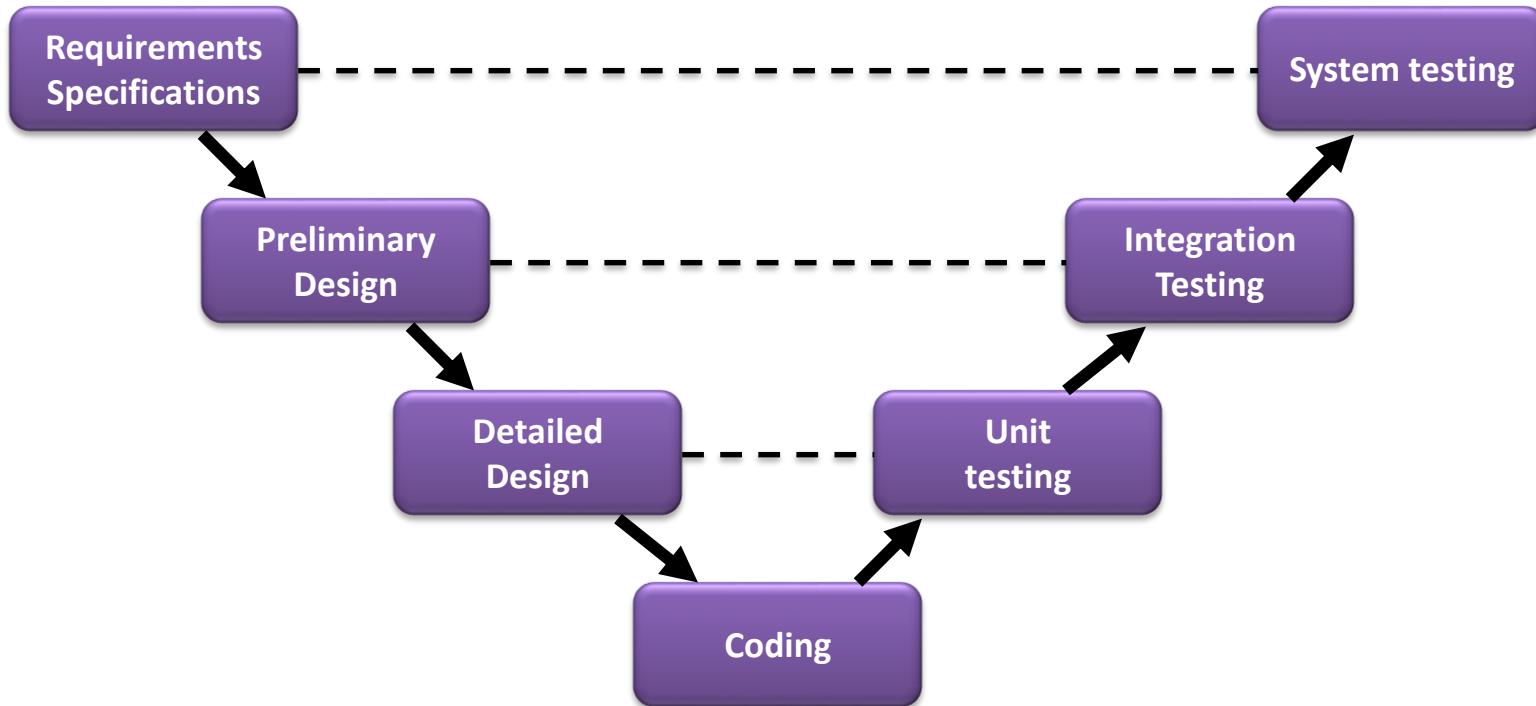


Topic 13.2 : Life-Cycles – Waterfall, Iterative & Agile

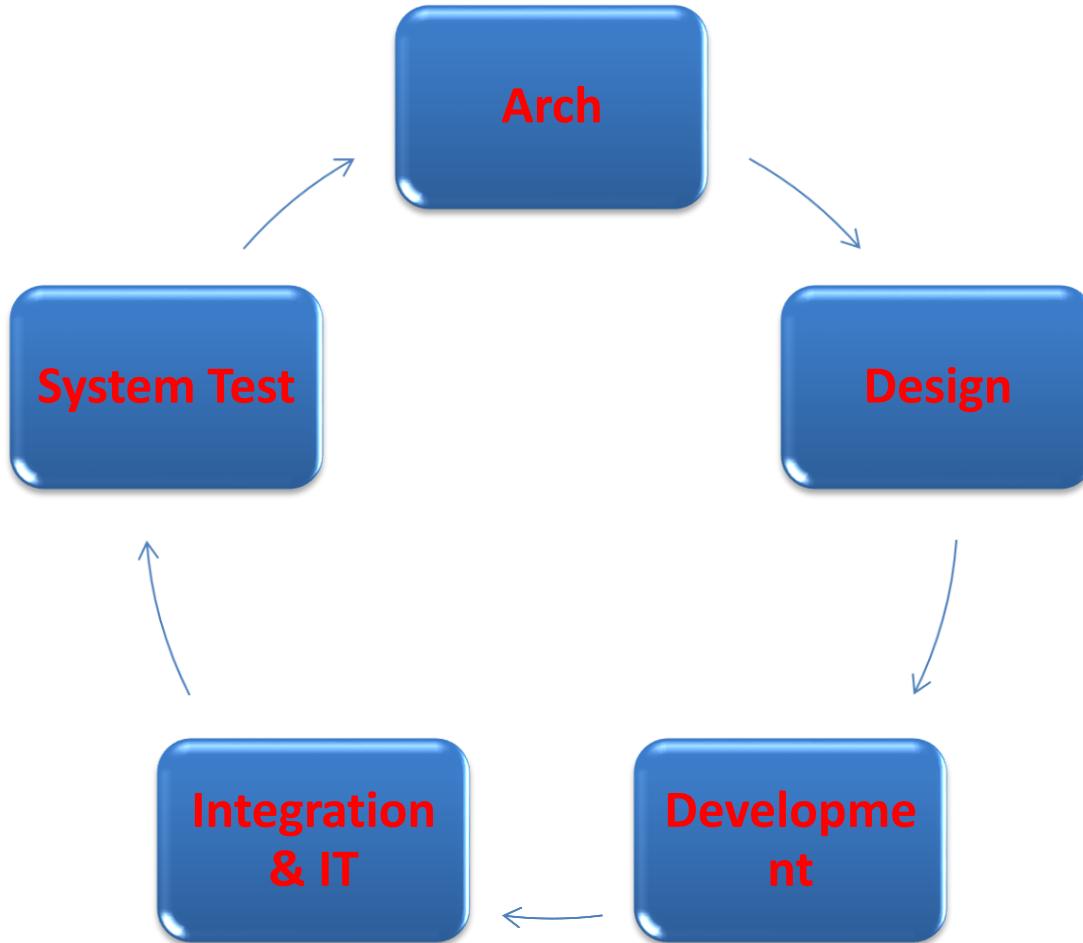
The Waterfall



The Waterfall



Iterative



- A View
- A Debate

AGILE Manifesto

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Source: <http://agilemanifesto.org/>

Principles Behind Agile Manifesto

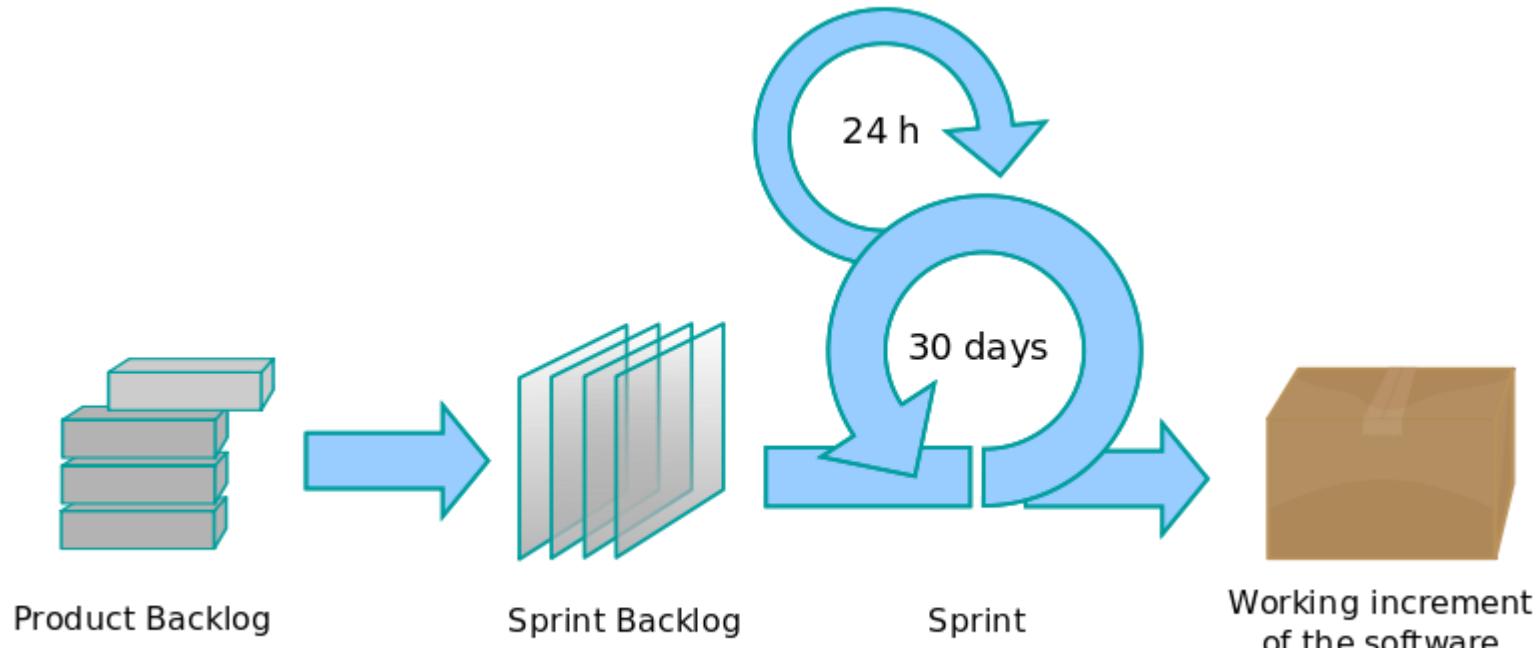


We follow these principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Source: <http://agilemanifesto.org/>

Scrum



Reference: SCRUM Development Process by Ken Scheweber



Topic 13.3 : Implications and Issues, Strategies & models



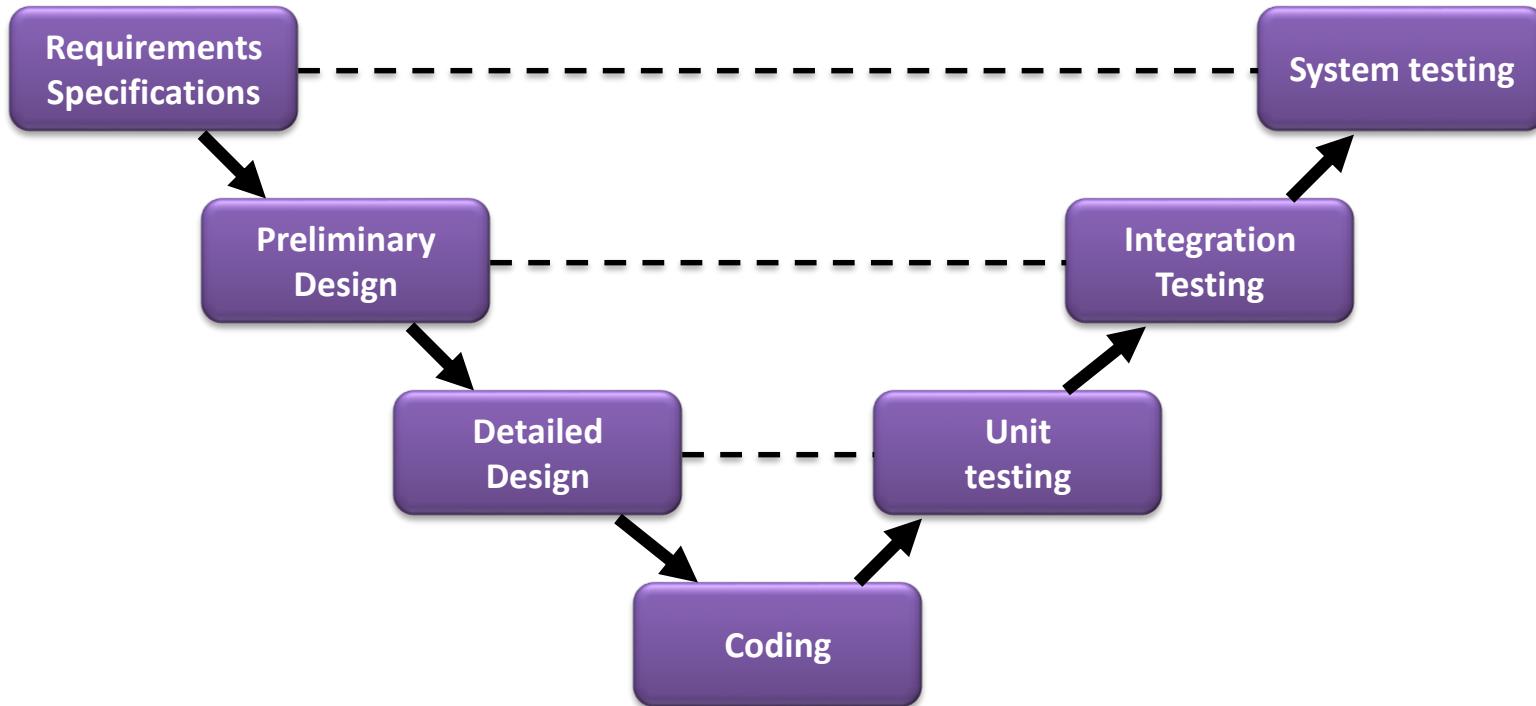
Issues & Implications with SDLCs

- Waterfall
- V Model
- Iterative
- Agile

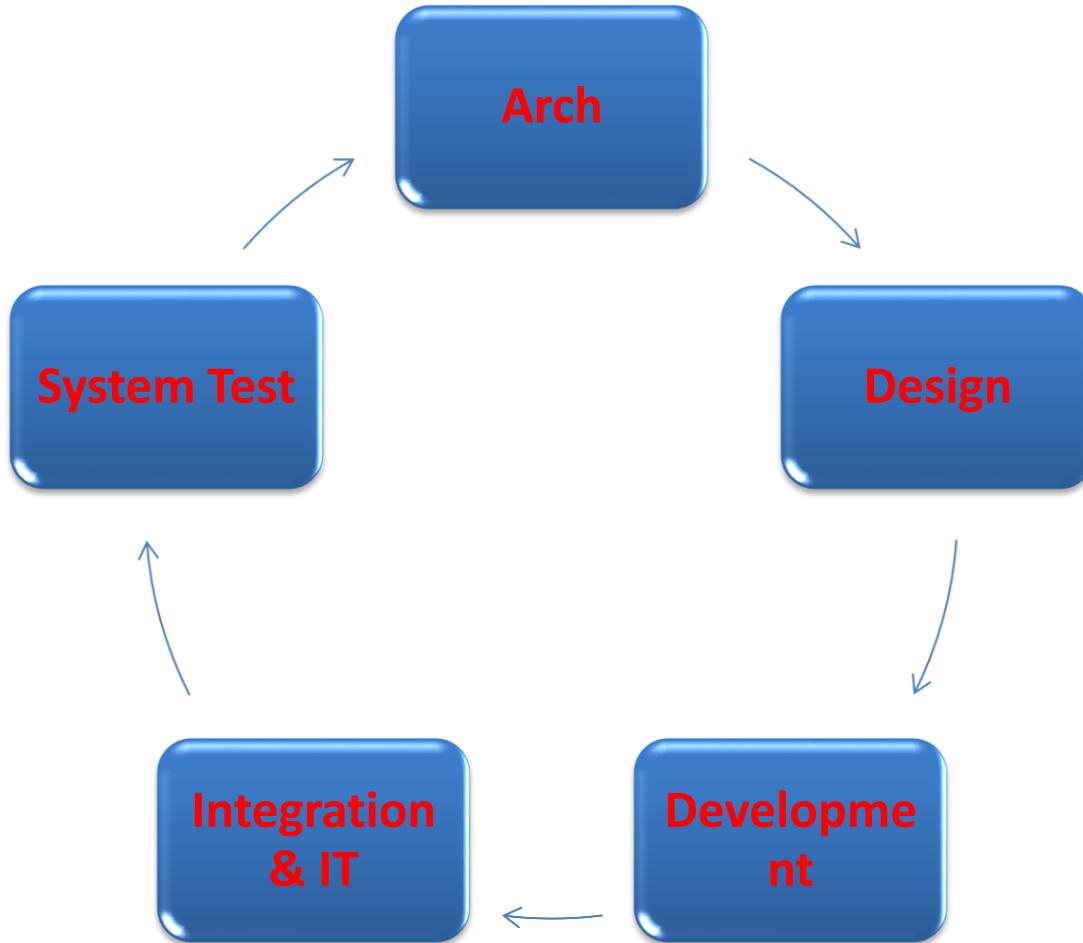
The Waterfall



The Waterfall

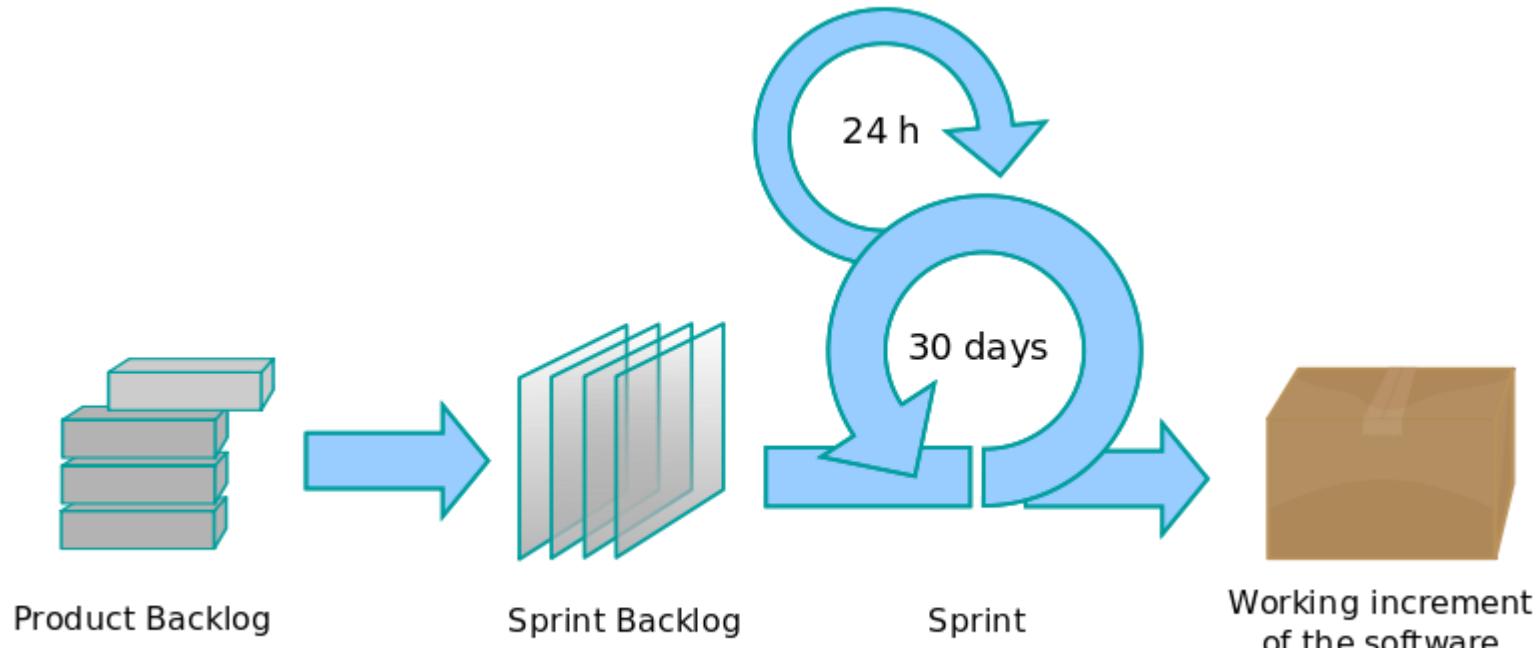


Iterative



- A View
- A Debate

Scrum



Reference: SCRUM Development Process by Ken Scheweber



Topic 13.4 : Example and Case

Garage Door

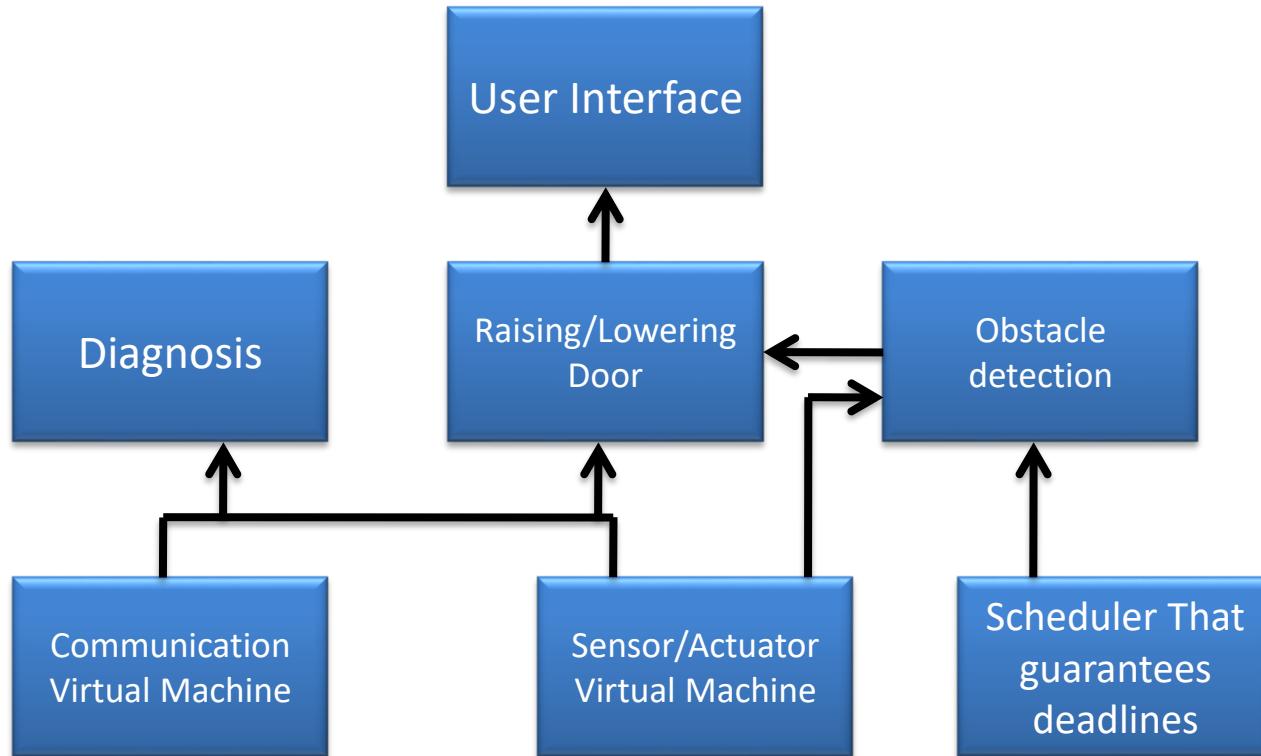
- The device controls for opening and closing the door are different for the various product lines. They may include controls from within a home information system. The product architecture for a specific set of controls should be directly derivable from the product line architecture
- The processor used in different processes will differ. The product architecture for each specific processor should be directly derivable from the product line architecture
- If an obstacle (person or object) is detected by the garage door during descent, it must halt (alternately re-open) within 0.1 second
- The garage door opener should be accessible for diagnosis and administration from within the home information system using product specific diagnosis protocol. It should be possible to directly produce an architecture that reflects this protocol

Building Lighting Control System

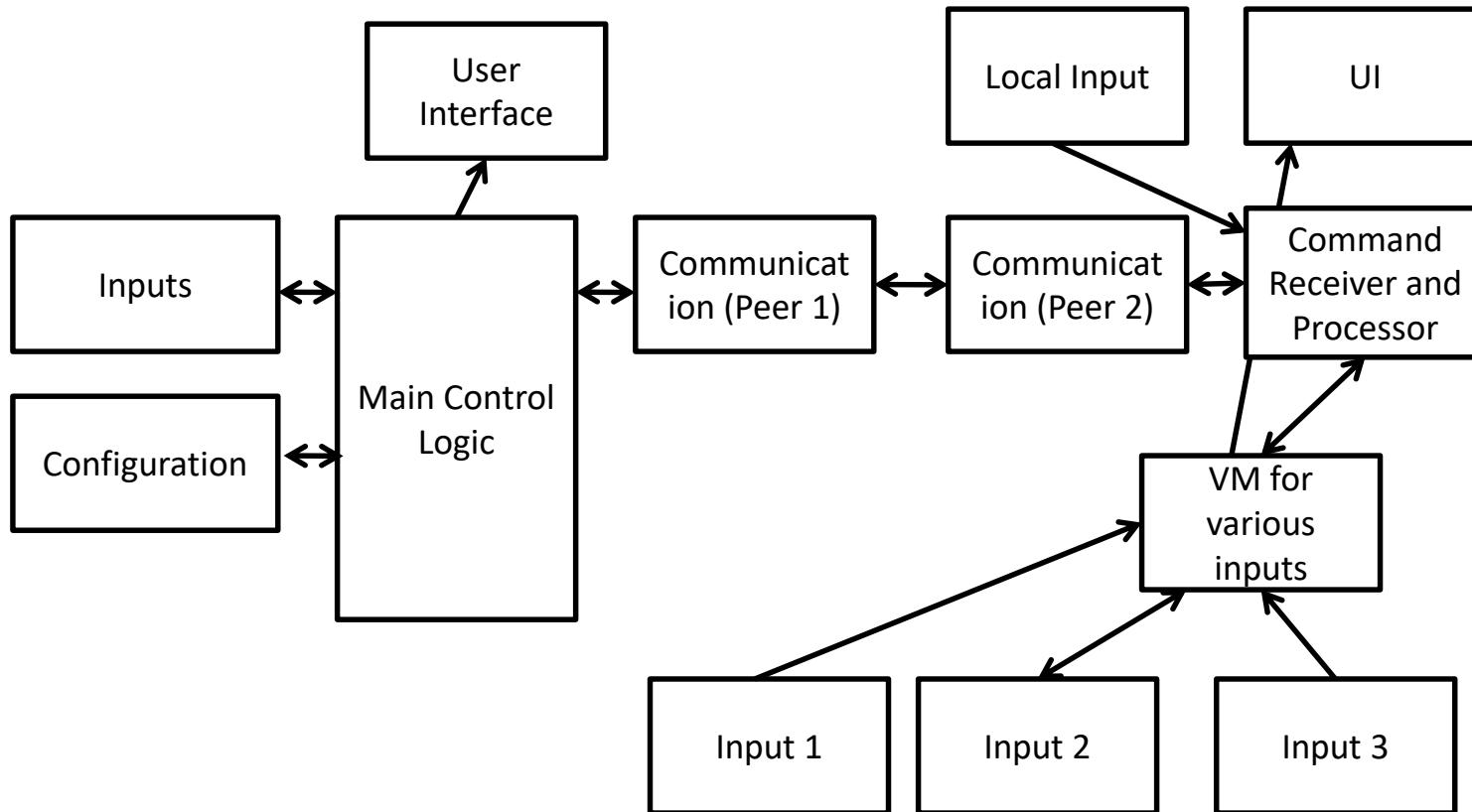


- Lighting control system for a apartment complex
- Emergency lights are UPS powered
- Corridor lights are timed and based on ambient light
- Garden and Architectural lights are timed

Garage Door



Building Lighting System



Module 13 Self Study

- SS13.1 Research the Agile Methodologies and find out various way in which developed software is expected to be tested. Find out and discuss the specific mechanisms and tools which are required for those approaches.



SS 8.1 Systems & Systems of Systems

8.1 Self Study

To explore:

- Systems and Systems of Systems around us

Study Work:

- Review systems and system of systems around us
- Write down their characteristics and working (in terms of use cases)
- Further, analyse use of techniques learnt so far for designing test cases
- Document your findings in form of a whitepaper (IEEE format recommended)

Module 14: Agenda

Module 14: Test Adequacy & Enhancement

Topic 14.1

Test Adequacy – Need & Overview

Topic 14.2

Test Adequacy Assessment – Data Flow

Topic 8.3

Test Adequacy Assessment – Control Flow

Topic 8.3

Examples



Topic 14.1: Test Adequacy – Need & Overview

-
- Is my testing adequate?
 - Is my testing complete?
 - Is my testing good enough?
 - Have I tested my program thoroughly?
 - Is testing covering all that needs to be covered

- Is my testing adequate?
 - Is my testing complete?
 - Is my testing good enough?
 - Is testing covering all that needs to be covered
- What do you think?**

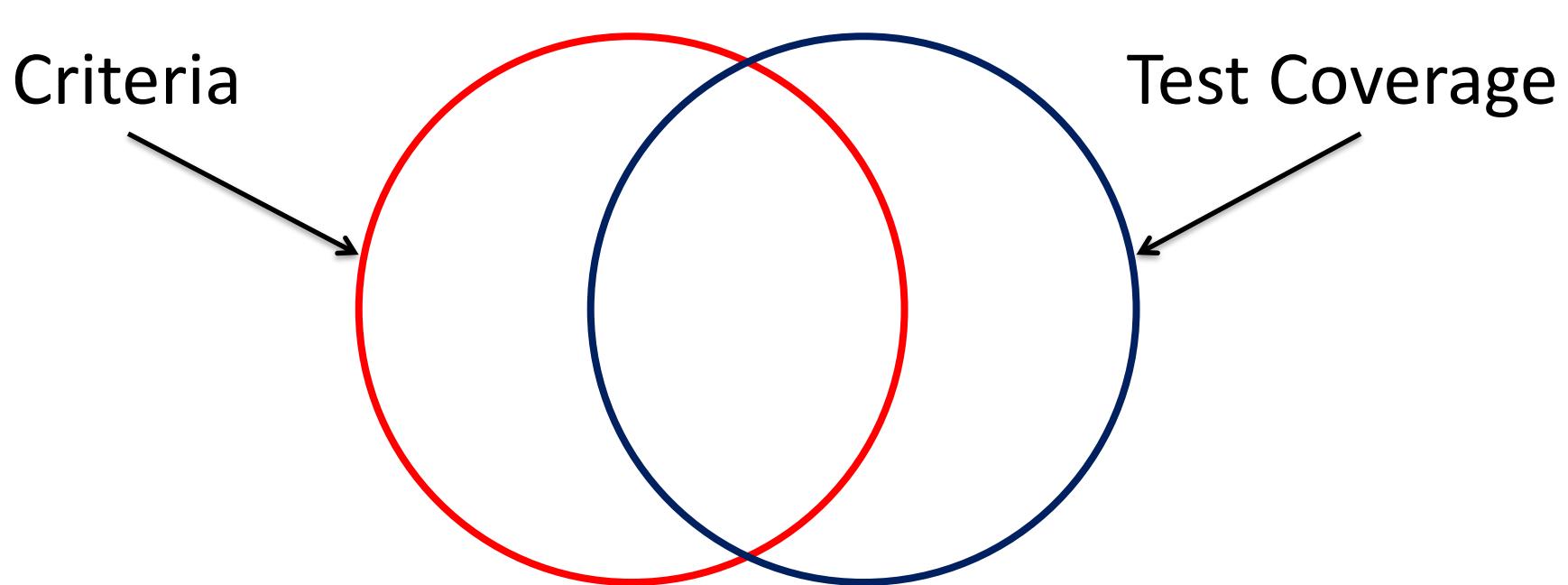
Some more questions

- What is test adequacy?
- What are the measures of test adequacy?
- If not adequate, How do we enhance tests?
- Achievable goals
 - Feasibility
 - Error Detection

Test Adequacy

- For us the term **adequate** means
 - Thorough
 - Good or Great
 - Completeor any other term you would like to suggest
- A test set is considered adequate if it satisfies a specific criteria
- A test set T is adequate to test program P , iff it satisfies the criteria C

Visual Representation



- **Establish the criteria**
- Use appropriate instrumentation
- Measure the coverage
- Enhance as required

Measurement of Test Adequacy

- For each adequacy criterion C, we derive a finite set known as the coverage domain and denoted by C_e
- A criterion C is a **white-box test adequacy** criterion if the corresponding coverage domain C_e depends solely on program P under test
- A criterion C is a **black-box test adequacy** criterion if the corresponding coverage C_e depends solely on requirements R for the program P under test

Test Enhancements using Measurements of Adequacy

- Increase the possibility of detecting any uncovered errors
- Adequacy does not guarantee an error-free program
 - Inadequate is a cause of worry for sure
- Identification of the deficiency helps in enhancement of the inadequate test set

Measure → Enhance → Adequacy

Measurement of Test Adequacy

- Measure Test Adequacy of T
- Measure if all elements of Ce are indeed covered by T
 - All covered: Adequate coverage
 - All not covered: Inadequate coverage
- k elements are covered out of n element
- Coverage is measured by k/n

Infeasibility and Test Adequacy

- An element of a coverage domain e.g. a path, might be infeasible to cover. This implies that there exists no test case that covers such an element
- It is quite possible that in spite of detecting the inadequacy an enhancement may not be possible.

Error Detection and Test Enhancement



- Test Enhancement → To determine test cases that test the untested parts of the program
- Test Enhancement → Uncover new errors of the program by exercising the new areas of the program

Test Adequacy & Execution

- Number of times program is executed with certain inputs will enable newer error detection
- Single or multiple execution matter
- Same input across multiple runs can detect the error which in a single execution may not be possible to detect



Topic 14.2: Test Adequacy Assessment – Data Flow

Test Adequacy – Data Flow

- The d, u and k
- DU Pairs (and other pairs)
- c-use & p-use

Infeasible Flow

- Nodes are not reachable
- Hence cannot determine the Data flow
- No test case or input may satisfy the path for the data flow of a variable



Topic 14.3: Test Adequacy Assessment – Control Flow

Test Adequacy – Control Flow

- Statement and Block Coverage
- Conditions and Decisions
- Multiple Condition Coverage

Infeasible Flow

- Unreachable code – unreachable Control Flow
- Uncovers “dead code”



Topic 14.4: Examples

Suggested Examples

- Review the sample code in T2 Chapter 7 in appropriate sections

Module 15: Agenda

Module 15: Test Case Minimization, Prioritization & Optimization (1/2)

Topic 15.1

Need & Motivation

Topic 15.2

Techniques

Topic 15.3

Regression Testing – Test Selection
(Execution Trace, Dynamic Slicing)



Topic 15.1: Need & Motivation

Need & Motivation

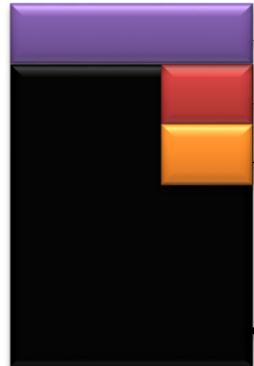
- All systems change over a period of time
 - Upgrades
 - Defect fixes
 - Roadmap – Platforms, change in technology
- On change
 - Regression testing takes front seat
 - Product = Base + Change
- Ensure
 - Base works as it should
 - New features and fixes work as well
 - System as a whole is of high quality

Regression Testing

- A program that is modified for reasons such as correction, addition of features etc., is often retested to ensure that the unchanged parts of the program continue to work correctly. Such testing is commonly referred to as regression testing

Regress means to return to previous,
usually worst, state

A Minefield!



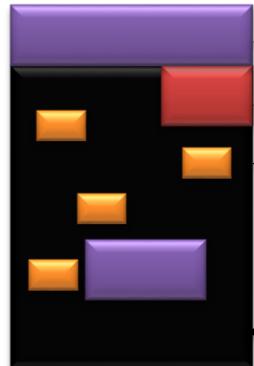
Version 2

← Newly Added
← Modified
← Deleted
← Unchanged/Base



Version 3

← Newly Added
← Modified
← Deleted
← Unchanged/Base



Version 4

← Newly Added
← Modified
← Deleted
← Unchanged/Base



Definitions - Optimisation

- Dictionary: an act, process, or methodology of making something (as a design, system, or decision) as fully perfect, functional, or effective as possible
- In mathematics, computer science, or management science, mathematical optimization is the selection of a best element from some set of available alternatives

Definitions - Minimisation

- Dictionary: to make (something bad or not wanted) as small as possible
- In mathematics, computer science, or management science, mathematical optimization is the selection of a least set of elements from the set of available alternatives



Our View

- Optimization
- Minimization
- Prioritization

Motivations & Constraints

- Resource
 - Effort
 - Staff
 - Time
 - Budget
- Management
 - Do more with less
 - Best use of resources
- Technical
 - Get the most efficacy



Topic 15.2: Techniques

Regression Test Process

- Selection, Prioritization and Minimization
- Test Setup
- Test Sequencing
- Test Execution
- Output Comparison

Test Selection

- Test All
- Random Selection
- Selecting Modification traversing tests
- Test Minimization
- Test prioritization

Test Minimization

- Reduce the set T
- Discard Redundant Test Cases
- Criteria
 - Code Coverage
 - Requirements Coverage
 - A form of data or control flow
 - ANY OTHER

Test Prioritization

- Select test cases based on priority
- Discard None
- Prioritization Criteria
 - Requirements Coverage
 - Code Coverage
 - Desired Metric

Regression Test Selection

- Algorithms for test selection
- Tools for test selection

Test Optimization

- Selection of best element
- Remember the “best of the breed”
- Criteria Based
- Technique Based
 - OATS (Orthogonal Array)
 - Combinatorial
 - Choice of Values in EC
 - Special Value



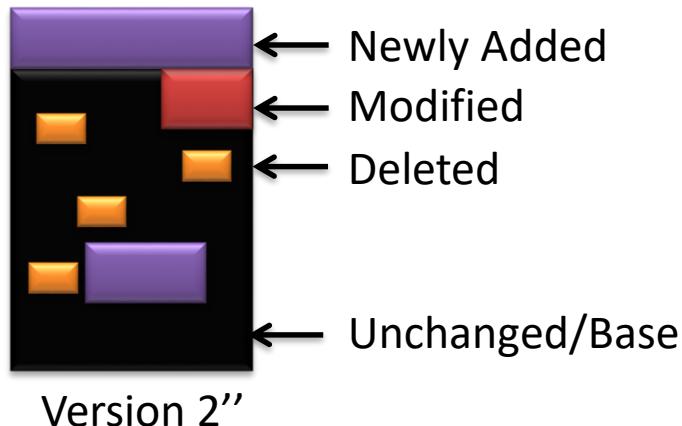
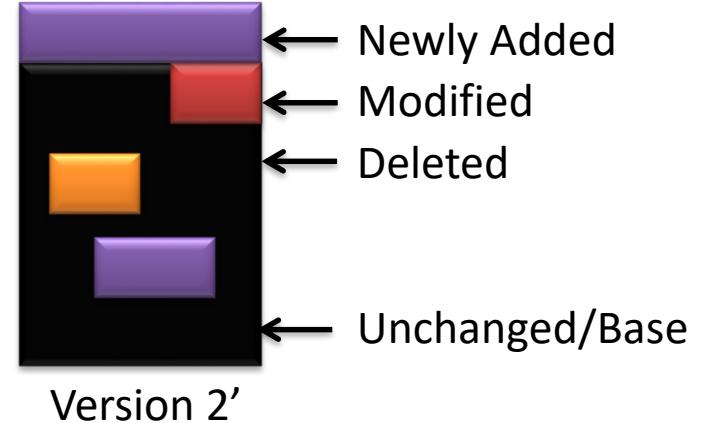
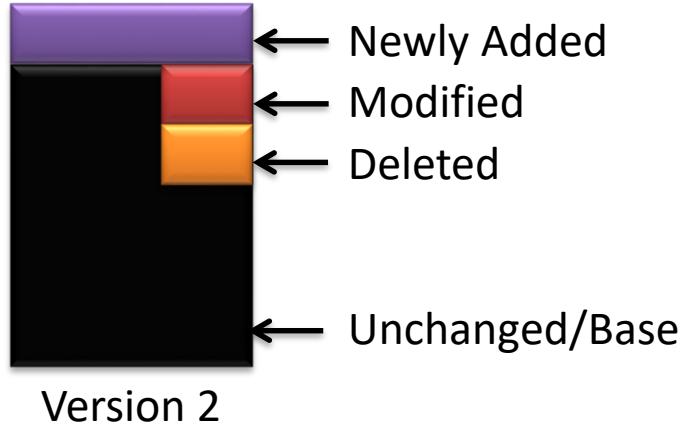
Topic 15.3: Regression Testing – Test Selection (Execution Trace/Dynamic Slicing)

Regression Testing

- A program that is modified for reasons such as correction, addition of features etc., is often retested to ensure that the unchanged parts of the program continue to work correctly. Such testing is commonly referred to as regression testing

Regress means to return to previous,
usually worst, state

A Minefield!



A
new
form!

Use of Execution Trace

- Obtain the Execution Trace
- Selection of Regression Tests

- CFG
- PDG
- DU Pairs

Use of Dynamic Slicing

- Ensuring that the affected variable is part of the execution
 - Limitations
 - Better than Execution Trace
-
- CFG
 - Execution Trace
 - Dependence Graphs

Test Minimisation

- The Example Code
- Review the Data Set
- Review the Test Set
- Only Control Flow and Data Flow may not be enough
- Redundancy removal with care! Utmost care!

Test Prioritisation

- Ranking
- No Discarding
 - Ranking by Context (Example: During development or Regression)
 - Absolute Ranking (At all times through out the product development, release & maintenance)

Test Design and Selection

- Pairwise Design – Binary Factors
- Pairwise Design – Multi-valued Factors
- Orthogonal Arrays

Pairwise – Binary

X1 Y1 Z1
X1 Y1 Z2
X1 Y2 Z1
X1 Y2 Z2
X2 Y1 Z1
X2 Y1 Z2
X2 Y2 Z1
X2 Y2 Z2

X1 Y1
X1 Y2
X1 Z1
X1 Z2
X2 Y1
X2 Y2
X2 Z1
X2 Z2
Y1 Z1
Y1 Z2
Y2 Z1
Y2 Z2

Pairwise – Mixed- Valued Factors

Block	Row	F1	F2	F3
1	1	1	1	1
	2	1	2	2
	3	1	3	3
2	1	2	1	2
	2	2	2	3
	3	2	3	1
3	1	3	1	3
	2	3	2	1
	3	3	3	2

Orthogonal Arrays

Run	F1	F2	F2
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Run	F1	F2	F2	F4
1	1	1	1	1
2	1	2	2	3
3	1	3	3	2
4	2	1	2	2
5	2	2	3	1
6	2	3	1	3
7	3	1	3	3
8	3	2	1	2
9	3	3	2	1

Examples

- Cross Platform Application
- Web Browser on various platforms

Module 16: Agenda

Module 16: Test Case Minimisation, Prioritisation & Optimisation

Topic 16.1

Minimisation, Prioritisation & Optimisation Techniques

Topic 6.2

Test Selection Algorithms

Topic 16.3

Examples



Topic 16.1: Minimisation, Prioritisation & Optimisation Techniques

Test Minimisation

- The Example Code
- Review the Data Set
- Review the Test Set
- Only Control Flow and Data Flow may not be enough
- Redundancy removal with care! Utmost care!

Test Prioritisation

- Ranking
- No Discarding
 - Ranking by Context (Example: During development or Regression)
 - Absolute Ranking (At all times through out the product development, release & maintenance)



Topic 16.2: Test Selection Algorithms

Test Design and Selection

- Pairwise Design – Binary Factors
- Pairwise Design – Multi-valued Factors
- Orthogonal Arrays

Pairwise – Binary

X1 Y1 Z1
X1 Y1 Z2
X1 Y2 Z1
X1 Y2 Z2
X2 Y1 Z1
X2 Y1 Z2
X2 Y2 Z1
X2 Y2 Z2

X1 Y1
X1 Y2
X1 Z1
X1 Z2
X2 Y1
X2 Y2
X2 Z1
X2 Z2
Y1 Z1
Y1 Z2
Y2 Z1
Y2 Z2

Pairwise – Mixed- Valued Factors



Block	Row	F1	F2	F3
1	1	1	1	1
	2	1	2	2
	3	1	3	3
2	1	2	1	2
	2	2	2	3
	3	2	3	1
3	1	3	1	3
	2	3	2	1
	3	3	3	2

Orthogonal Arrays

Run	F1	F2	F2
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Run	F1	F2	F2	F4
1	1	1	1	1
2	1	2	2	3
3	1	3	3	2
4	2	1	2	2
5	2	2	3	1
6	2	3	1	3
7	3	1	3	3
8	3	2	1	2
9	3	3	2	1



Topic 16.3: Examples

Examples

- Cross Platform Application
- Web Browser on various platforms



SS 16.1 Compare & Contrast – Combinatorial Techniques

16.1 Self Study

To explore:

- Compare & contrast use of combinatorial techniques for test reduction

Study Work:

- Identify and study various combinatorial test techniques
- Review the test techniques which enable reduction of test cases
- Compare and contrast those techniques
- Comment on the risk of reduction of test cases