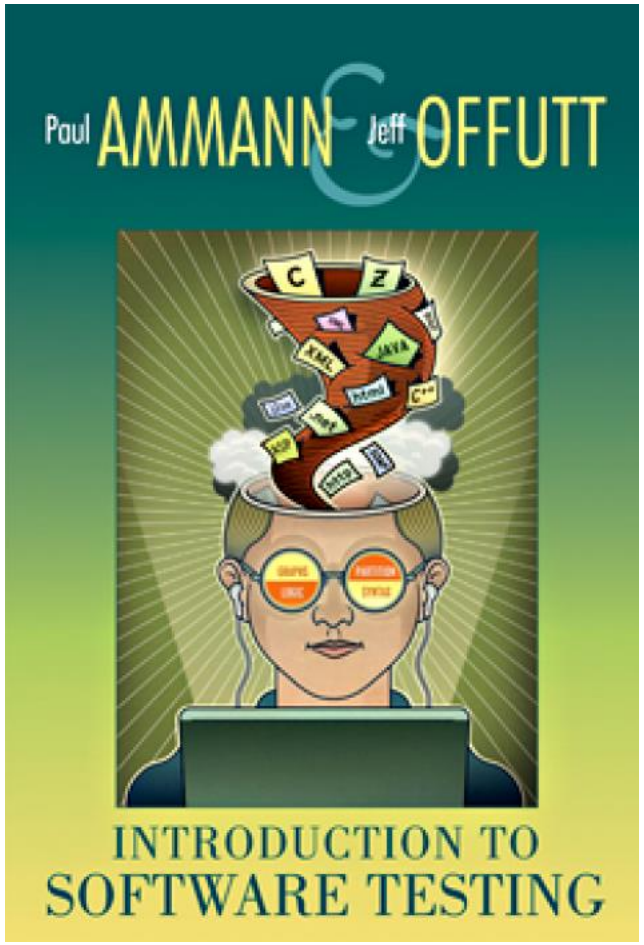
Software Testing

Introduction –

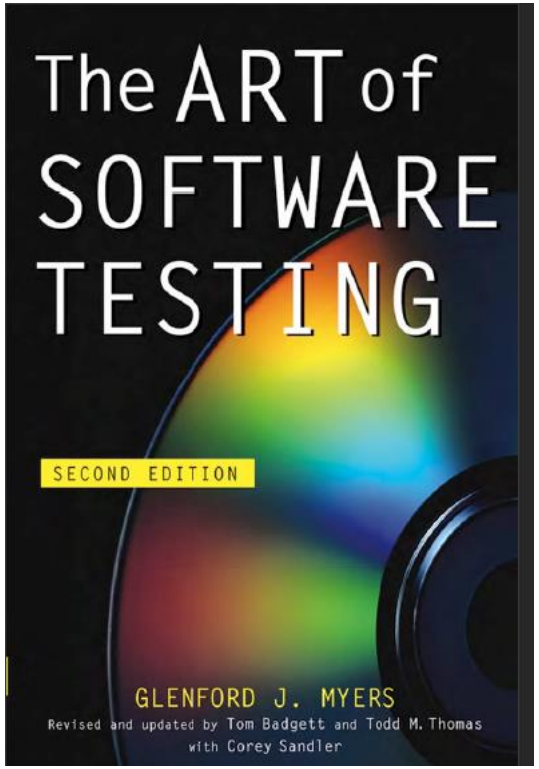
Lectures 1,2

One of the most important limitations of software testing is that testing can show only the presence of failures, not their absence

Course Main Reference



A balance of theory and practical application



Choose the Best Answer

- “Testing is the process of demonstrating that errors are not present.”

- “The purpose of testing is to show that a program performs its intended functions correctly.”

- “Testing is the process of establishing confidence that a program does what it is supposed to do.”

Choose the Best Answer

- "Testing is the process of demonstrating that errors are not present." [less important: no value]

- "The purpose of testing is to show that a program performs its intended functions correctly."

- "Testing is the process of establishing confidence that a program does what it is supposed to do."

Who should test?

- Every engineer involved in software development should realize that he or she sometimes wears the hat of a test engineer.
- The reason is that each software artifact produced over the course of a product's development has, or should have, an associated set of test cases, and **the person best positioned to define these test cases is often the designer of the artifact**.

Testing techniques typically are presented in the context of a particular software artifact (for example, a **requirements document or code**) or a particular phase of the lifecycle (for example, requirements analysis or implementation)

Formal Coverage Criterion

- Formal coverage criteria give test engineers ways to decide **what test inputs to use during testing**, making it more likely that the tester will find problems in the program and providing greater assurance that the software is of high quality and reliability

  - ❑We cannot test with all input
  - ❑Are used to decide which test inputs to use
  - ❑ Provide stopping rules for the test engineers

Activities of Test Engineer

- Every engineer involved in software development should realize that he or she sometimes wears the hat of a test engineer.
- The reason is that each software artifact produced over the course of a product's development has, or should have, an associated set of test cases, and the person best positioned to define these test cases is often the designer of the artifact

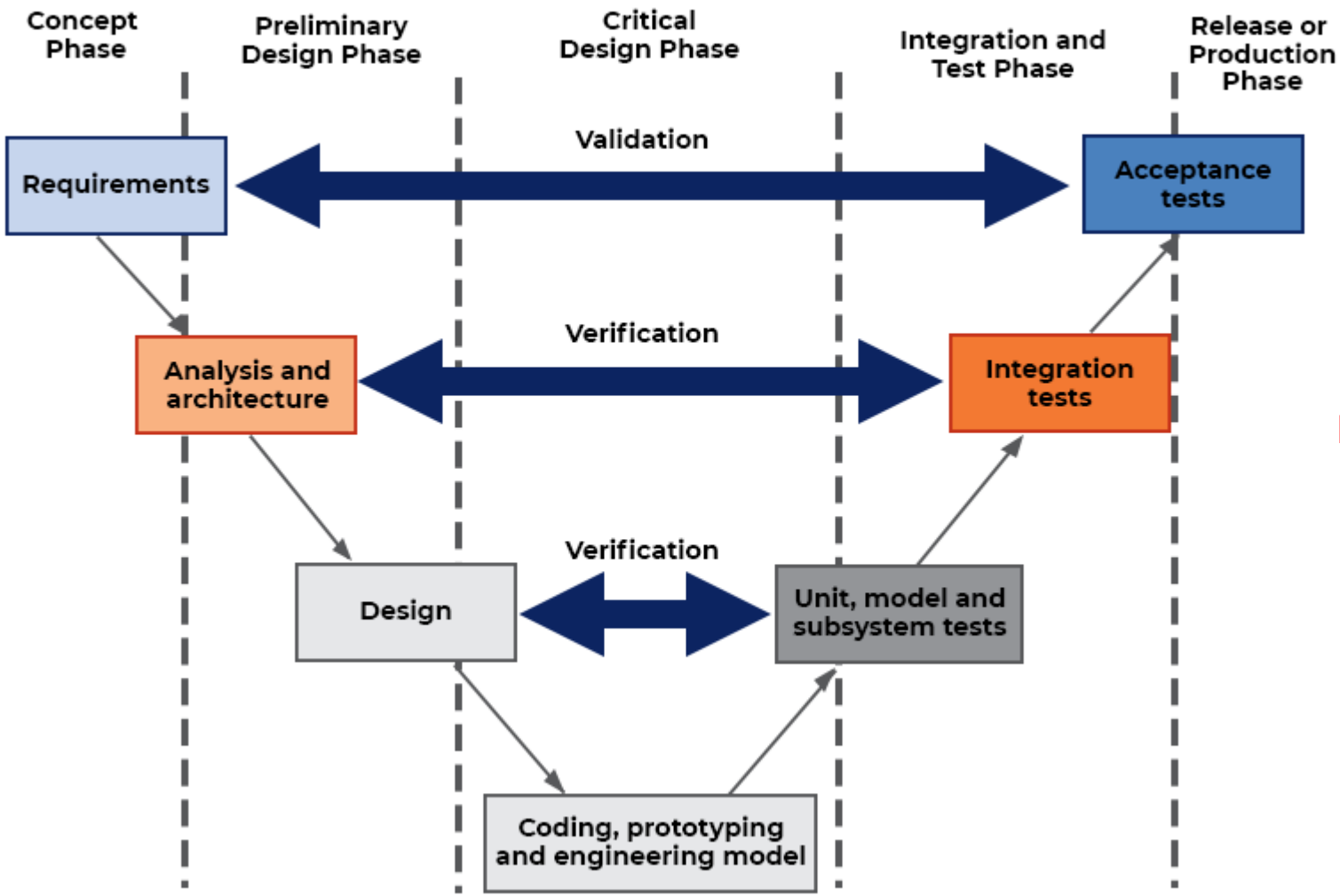
Software testing activities

- The most often used level categorization is **based** on traditional **software process steps.**
- The second-level categorization is **based on the attitude and thinking of the testers.**

Testing Levels Based on Software Activity

Tests can be derived from requirements and specifications, design artifacts, or the source code. A different level of testing accompanies each distinct software development activity:

- **Acceptance Testing** – assess software with respect to **requirements**:
  Acceptance testing must involve users or other individuals who have strong domain knowledge
- **System Testing –** assess software with respect to **architectural design**.
- **Integration Testing** – assess software with respect to **subsystem design**.
- **Module Testing –** assess software with respect to **detailed design**
- **Unit Testing** – assess software with respect to **implementation**.



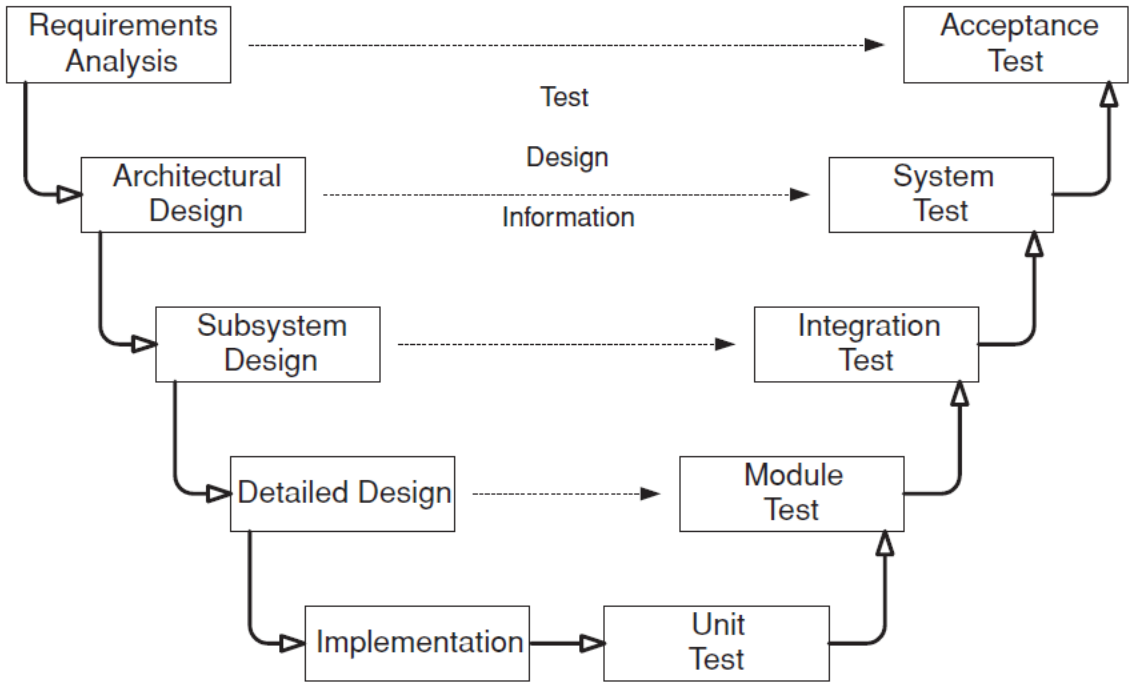
System Testing is Blackbox

End to End testing scenario.

https://kruschecompany.com/v-model-software-development-methodology/

The V-Model :Testing Levels Based on Software Activity

- Tests can be derived from requirements and specifications, design artifacts, or the source code. A different level of testing accompanies each distinc software development activity:
- **Acceptance Testing** – assess software with respect to requirements.
- **System Testing** – assess software with respect to architectural design.
- **Integration Testing** – assess software with respect to subsystem design.
- **Module Testing** – assess software with respect to detailed design.
- **Unit Testing** – assess software with respect to implementation.



What do you verify in System Testing?

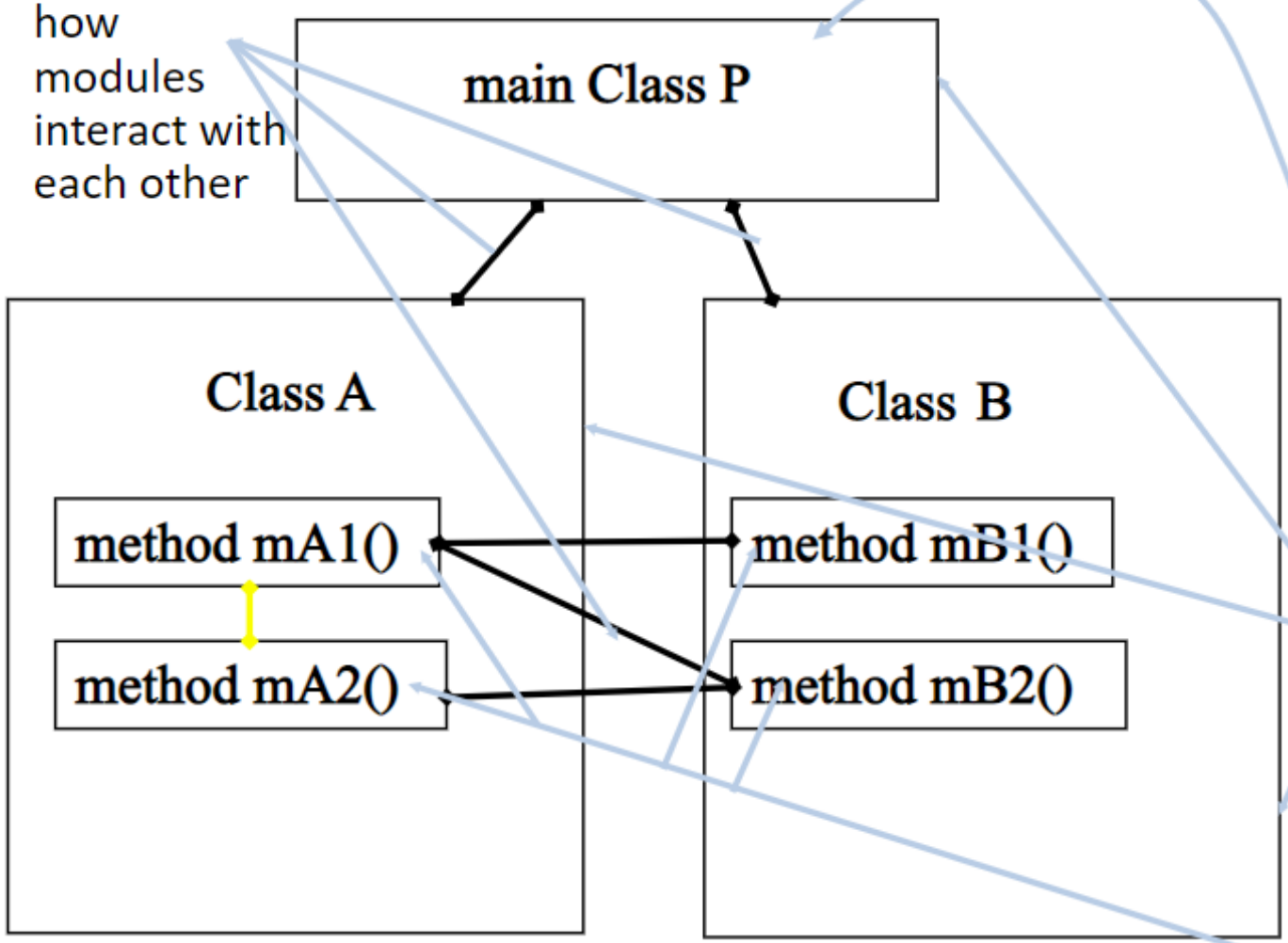
- Testing the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole. This is also called End to End testing scenario.
- Verify thorough testing of every input in the application to check for desired outputs.
- Testing of the user's experience with the application.

Integration Testing

- *Integration testing* is designed to assess whether the interfaces between modules (defined below) in a given subsystem have consistent assumptions and communicate correctly. Integration testing must assume that modules work correctly.
- Some testing literature uses the terms integration testing and system testing interchangeably; in this book, integration testing does **not** refer to testing the integrated system or subsystem. Integration testing is usually the responsibility of members of the development team.

Traditional Testing Levels

- Integration testing : Test how modules interact with each other



- System testing : Test the overall functionality of the system
- Module testing (developer testing) : Test each class, file, module or component
- Unit testing (developer testing) : Test each unit (method) individually

Testing Goals Based on Test Process Maturity

- **Level 0 :** There's no difference between **testing** and **debugging**
- **Level 1 :** The purpose of testing is to show correctness
- **Level 2 :** The purpose of testing is to show that the software doesn't work
- **Level 3 :** The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- **Level 4 :** Testing is a mental discipline that helps all IT professionals develop higher quality software

Level 0

- Testing is the same as debugging
- Does not distinguish between incorrect behavior and mistakes in the program
- Does not help develop software that is reliable or safe

Level 1

- In **Level 1** testing, the purpose is to show correctness. While a significant step up from the naive level 0, this has the unfortunate problem that in any but the most trivial of programs, correctness is virtually impossible to either achieve or demonstrate.
- **Suppose we run a collection of tests and find no failures. What do we know? Should we assume that we have good software or just bad tests?**
- Since the goal of correctness is impossible, test engineers usually **have no strict goal, real stopping rule, or formal test technique**. If a development manager asks how much testing remains to be done, the test manager has no way to answer the question. In fact, test managers are in a powerless position because they have no way to quantitatively express or evaluate their work.

Level 2

- In **Level 2** testing, the purpose is to show failures. Although looking for failures is certainly a valid goal, it is also a negative goal. Testers may enjoy finding the problem, but the developers never want to find problems – they want the software to work (level 1 thinking is natural for the developers).
- Thus, level 2 testing puts testers and developers into an adversarial relationship, which can be bad for team morale.
- Beyond that, when our primary goal is to look for failures, we are still left wondering what to do if no failures are found. Is our work done? Is our software very good, or is the testing weak? Having confidence in when testing is complete is an important goal for all testers.

Level 3

- The thinking that leads to **Level 3** testing starts with the realization that testing can show the presence, but not the absence, of failures. This lets us accept the fact that whenever we use software, we incur some risk. The risk may be small and the consequences unimportant, or the risk may be great and the consequences catastrophic, but risk is always there.
- This allows us to realize that the entire development team wants the same thing – to reduce the risk of using the software. In level 3 testing, **both testers and developers work together to reduce risk**

Level 4

- Level 4 thinking defines **testing as *a mental discipline that increases quality*.** Various ways exist to increase quality, of which creating tests that cause the software to fail is only one. Adopting this mindset, test engineers can become the technical leaders of the project (as is common in many other engineering disciplines). They have the primary responsibility of measuring and improving software quality, and their expertise should help the developers. An analogy that Beizer used is that of a spell checker.

Testing vs Debugging

- Testing is a process that confirms a program or system meets all its specifications.
- Debugging is finding and fixing errors in a program or system. ***Various methods of debugging code include:***
  - Using breakpoints to halt execution at fixed points in the program, like analyzing variables and their values.
  - Tracking program flow.
  - Examining memory contents and simulating possible inputs to detect strange behavior.

Exercise

- Write a Java method with the signature public static Vector union (Vector a, Vector b). The method should return a Vector of objects that are in either of thetwo argument Vectors.

a) Identify as many possible faults as you can. (*Note:* Vector *is a Java*

Collection *class. If you are using another language, interpret* Vector *as alist.*)

(c) Create a set of test cases that you think would have a reasonable chance of revealing the faults you identified above. Document a rationale for each test in your test set. If possible, characterize all of your rationales in some concise summary. Run your tests against your implementation.

(d) Rewrite the method signature to be precise enough to clarify the defects and ambiguities identified earlier. You might wish to illustrate your specification with examples drawn from your test cases.

Exercise :Consider the following Java method:

```
public static Vector union (Vector a, Vector b) {
   Vector result = new Vector (a); // get all of a's elements
   Iterator itr = b.iterator();
   while (itr.hasNext()) {
      Object obj = itr.next();
      if (!a.contains (obj)){
         result.add (obj);
      }
   }
   return result;
}
```

1. The method should return a Vector of objects that are in either of the two argument Vectors. Upon reflection, you may discover a variety of defects and ambiguities. In other words, ample opportunities for faults exist. Identify as many possible faults as you can.

2. Create a set of test cases that you think would have a reasonable chance of revealing the faults you identified above. Document a rationale for each test in your test set. If possible, characterize all of your rationales in some concise summary. Run your tests against your implementation.

3. Rewrite the method signature to be precise enough to clarify the defects and ambiguities identified earlier. You might wish to illustrate your specification with examples drawn from your test cases.

Vector Union

```
import java.util.*;
public class Main {
    public static Vector union(Vector a, Vector b) {
     Vector r = new Vector();
    for (int i = 0; i < a.size(); i++)
    r.add(a.elementAt(i));// ww w . j av a 2 s . co m
    for (int i = 0; i < b.size(); i++)
    r.add(b.elementAt(i)); return r;
    }
}
```

Are we doing the right thing?

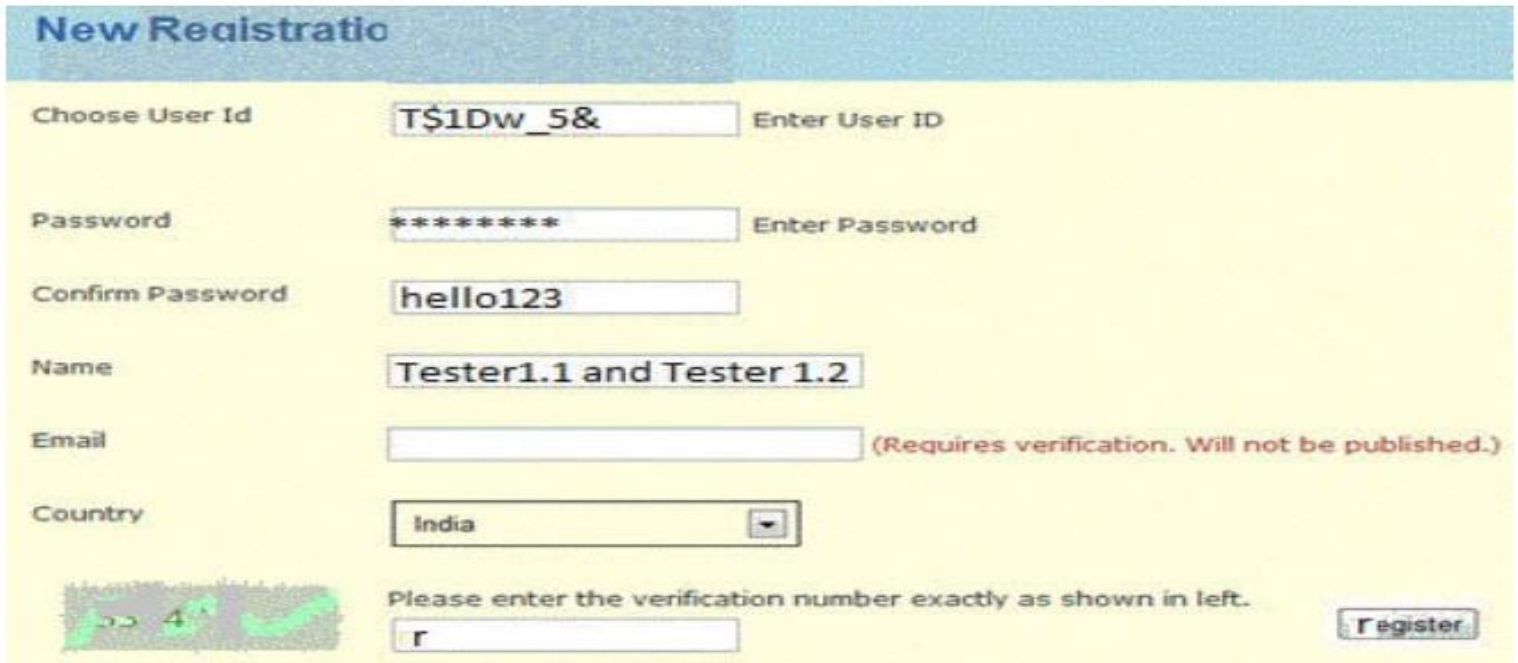
- Check if vectors are the best solution
- Are they obsolete or still used
- Cases where we need vectors [synchronization]
- Do we need to maintain the order
- Is duplicate allowed
- Understand how java iterator works [hasNext()]
- ---
- Options: Vector , List , Set

Solution

- Many potential ambiguities exist. The method name suggest set union, but the argument types (i.e. Vector) suggest otherwise.
- What happens if an input Vector is null?
- What happens if an input Vector contains duplicates?
- What happens if an input Vector contains null entries?
- Are the inputs modified?
- Is the return value always a new set, or might it be either a or b?
- The choice of argument types and return types as Vector is suspect; best practice is to use the least specific type consistent with the specier's goals. In particular, **if Vector is replaced with List**, this solves some, but not all of the problems above. **Replacing Vector with Set i**s probably an even better idea, assuming the arguments are really intended to be sets.

Exercise #1: Finding Defects

As a software tester, what do you do? Of course, testing the software, you would say.....Okay, can you find out defects on the page shown below?



Here is how you can judge yourself:

If you find:
0 – 4 defects => Poor
5 – 6 defects => Average
7 – 8 defects => Good
9 – 10 defects => Excellent
10+ defects => Best tester!

https://www.softwaretestinghelp.com/software-testing-exercises-to-test-your-skills/


New Registratio
Choose User Id
T$1Dw_5&
Enter User ID
Password
****
Enter Password
Confirm Password
Hello123
Name
Tester1.1 and Tester 1.2
Email
(Requires verification. Will not be published.)
Country
India
Please enter the verification number exactly as shown in left.
r
register

Answer: **Defects :**

1. *The user Id field accepts special characters.*
2. *Confirm password field does not show content in encrypted mode.*
3. *The name field does not seem to have any validation for number of characters.*
4. *Captcha is not at all readable.*
5. *There is no way user can reload the captcha.*
6. *Register button should be at bottom rather than on side.*
7. *Register button's label has "r" instead of "R".*
8. *There is no cancel button available if user wants to cancel the procedure..*
9. *There is no close button available if user wants to close the page.*
10. *The page title show wrong spelling of Registration.*
11. *Password selection guideline should be provided like the password should be alpha numeric or password strength factor should be present.*
12. *Page title should be New User Registration rather than New Registration.*
13. *Field length and labels should be same for the whole page / form.*
14. *The country field should by default show "Select" rather than selecting a value default.*

Software Testing Terminology

- *Verification*
- *Validation*
- *Software Fault:*
- *Software Error*
- *Software Failure*
- *Testing*
- *Test Failure*
- *Debugging:*
- *Expected Results*
- *Software Observability*
- *Software Controllability*

Prefix Values
Postfix Values
Verification Values
Exit Commands
Test Case
Test Set:
Executable Test Script
Test Requirement
Coverage Criterion
Coverage
Coverage Level
Criteria Subsumption

Older Software Testing Terminology

- *Black-box testing*
- *White-box testing*
- *Top-Down Testing*
- *Bottom-Up Testing*
- *Static Testing*
- *Dynamic Testing*

Verification vs Validation

Building the right system right

Are we building the right system (validation) in a right way (verification)?

Verification vs Validation

- ***Definition 1.1 Validation:*** The process of evaluating software at the end of software development to ensure compliance with intended usage.
- ***Definition 1.2 Verification:*** The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

Verification is usually a more technical activity that uses knowledge about the individual software artifacts, requirements, and specifications. Validation usually depends on domain knowledge; that is, knowledge of the application for which the software is written

Fault, Error and Failure

- *Definition 1.3* ***Software Fault***: A static defect in the software
- *Definition 1.4* ***Software Error***: An incorrect internal state that is the manifestation of some fault.
- *Definition 1.5* ***Software Failure:*** External, incorrect behavior with respect to the requirements or other description of the expected behavior

IEEE Definitions

- **Failure:** External behavior is incorrect
- **Fault:** Discrepancy in code that causes a failure.
- **Error:** Human mistake that caused fault

Note:

- **Error** is terminology of Developer.
- **Bug** is terminology of Tester

Fault : It is a condition that causes the software to fail to perform its required function.
Error : Refers to difference between Actual Output and Expected output.
Failure : It is the inability of a system or component to perform required function according to its specification.

Identify any Fault, error and failure?

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x

int count = 0;
for (int i = 1; i < x.length; i++)
  {

    if (x[i] == 0)     {

       count++;
     }
  }

  return count;
}
```

Try the following inputs

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x

int count = 0;
for (int i = 1; i < x.length; i++)
  {

    if (x[i] == 0)
     {

       count++;
     }
  }

  return count;
}
```

- Inputs
- numZero ([2, 7, 0])
- numZero ([0, 7, 2])

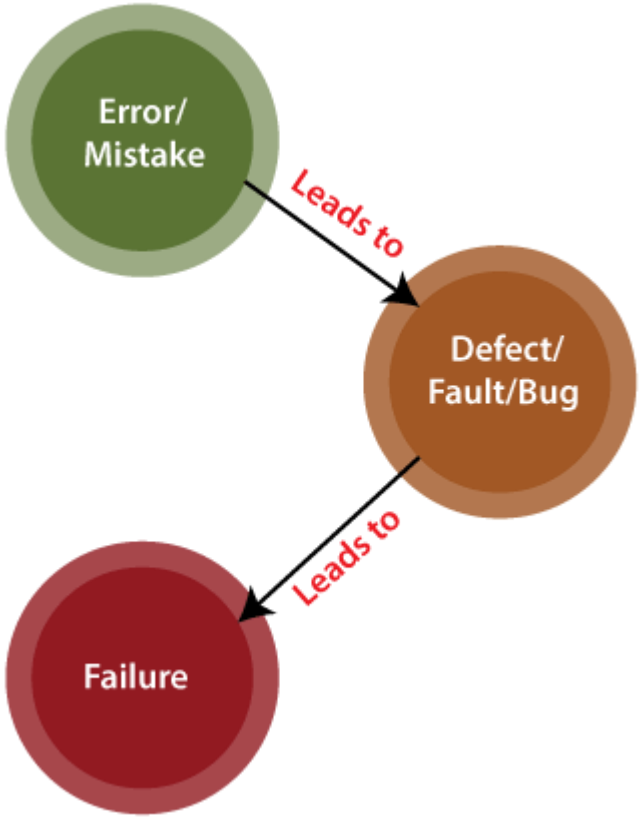
Error States

```
public static int numZero (int[] x) {

int count = 0;
for (int i = 1; i < x.length; i++)
  {

    if (x[i] == 0)
     {

       count++;
     }
  }

  return count;
}
```

x		Count	i	program counter - PC
Input-1 numZero ([2, 7, 0])		Count=0; i=1; PC=if		
Input-2 numZero ([0, 7, 2])		Count=0; i=1; PC=if		

The **fault i**n this program is that it starts looking for zeroes at index 1 instead of index 0, as is necessary for arrays in Java
For example, numZero ([2, 7, 0]) correctly evaluates to 1,
while numZero ([0, 7, 2]) incorrectly evaluates to 0.

Error vs Defect vs Failure



Error vs Defect vs Failure

Error

An error is a mistake made by a developer in the code. Its main causes are:

- **Negligence:** Carelessness often leads to errors
- **Miscommunication:** An unclear feature specification or its improper interpretation by developers could lead to slips.
- **Inexperience:** Inexperienced developers often miss out on essential details, leading to faults down the line.
- **Complexity:** Intricate algorithms can cause developers to make mistakes in their coding logic.

Error Example

```
class StackExample {
   public static void check(int i)
   {
      if (i == 0)
         return;
      else {
         check(i++);
      }
   }
}
public class Main {

   public static void main(String[] args)
   {
StackExample.check(5);
   }
}
```

What is Fault?

- The fault may occur in software because it has not added the code for fault tolerance, making an application act up.
- A fault may happen in a program because of the following reasons:
- Lack of resources
- An invalid step
- Inappropriate data definition

More Definitions

The definitions of fault and failure allow us to distinguish testing from debugging.

- *Definition 1.6 Testing:* Evaluating software by observing its execution.
- *Definition 1.7 Test Failure:* Execution that results in a failure.
- *Definition 1.8 Debugging:* The process of finding a fault given a failure

Fault/Failure Model :: RIP Model

For a given fault, not all inputs will "trigger" the fault into creating incorrect output (a failure). Also, it is often very difficult to relate a failure to the associated fault. Analyzing these ideas leads to the fault/failure model, which states that three conditions must be present for a failure to be observed.

1. The location or locations in the program that contain the fault must be reached (*Reachability*).
2. After executing the location, the state of the program must be incorrect (*Infection*).
3. The infected state must propagate to cause some output of the program to be incorrect (*Propagation*).

This "RIP" model is very important for coverage criteria such as mutation and for automatic test data generation.

Test Case Values

- *Definition 1.9 Test Case Values:* The input values necessary to complete some execution of the software under test.

Definition 1.10 Expected Results: The result that will be produced when executing the test if and only if the program satisfies its intended behavior. Two common practical problems associated with software testing are how to provide the right values to the software and observing details of the software's behavior. These two ideas are used to refine the definition of a test case.

Observability and Controllability

- ***Definition 1.11 Software Observability****:* How easy it is to observe the **behavior of a program in terms of its outputs**, effects on the environment, and other hardware and software components.
- ***Definition 1.12 Software Controllability****:* How **easy it is to provide a program with the needed inputs, in terms of values**, operations, and behaviors.

Test Case components

- *Definition 1.13* ***Prefix Values****:* Any inputs necessary to put the software into the appropriate state to receive the test case values.
- *Definition 1.14* ***Postfix Values:*** Any inputs that need to be sent to the software after the test case values are sent.
- Postfix values can be subdivided into two types.
  - *Definition 1.15* ***Verification Values****:* Values necessary to see the results of the test case values.
  - *Definition 1.16* ***Exit Commands****:* Values needed to terminate the program or otherwise return it to a stable state.

A test case is the combination of all these components (test case values, expected results, prefix values, and postfix values). When it is clear from context, however, we will follow tradition and use the term "**test case" in place of "test case values."**

Test Case

- ***Definition 1.17 Test Case:*** A test case is composed of the test case values, expected results, prefix values, and postfix values necessary for a complete execution and evaluation of the software under test.
- We provide an explicit definition for a test set to emphasize that coverage is a property of a set of test cases, rather than a property of a single test case.
- ***Definition 1.18 Test Set****:* A test set is simply a set of test cases.

- ***Definition 1.19 Executable Test Script****:* A test case that is prepared in a form to be executed automatically on the test software and produce a report

Project Name: Developing a Multi-Tenant and Integrated Xl System for- Ministry XXX			
Test Case: User manually enters an Assignment Decision			
Test case ID:	1.2 (success scenario)	**Test Designed by:**	Consultant
Test priority: (Low/Medium/High)	High	**Test Designed Date:**	6/27/2022
Module Name:	Assignment Module	**Test Executed by:**	
Test Title:	User manually enters an Assignment Decision	**Test Execution Date:**	
Description:	Assignment Decisions may be entered manually along with their attachments. If the decisions are approved, an Assignment Transaction will be initiated where further input is needed.		

Pre-Conditions:	The user is authorized.
Dependencies:	

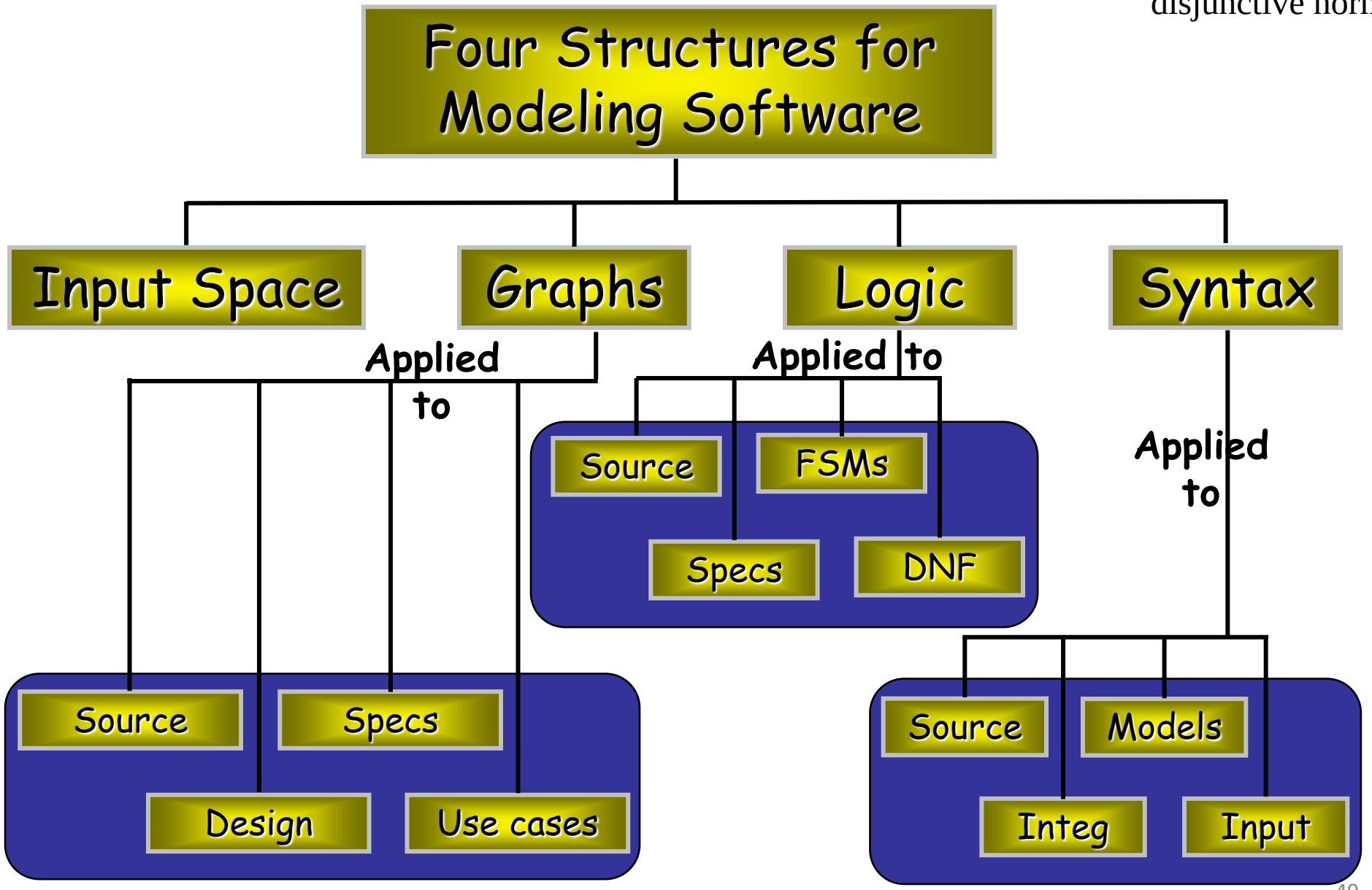
Step	Test Steps	Test Data	Expected Results	Actual Results	Status (Pass/Fail)	Notes
1	Navigate to the Assignment Decisions Screen.		The screen will display three tabs: "All Decisions", "Incoming Decisions", "Handled Decisions".			
2	Click on the "Incoming Decisions" Tab.		All the new Incoming Decisions will be displayed.			
3	Click on "Add Assignment Decision".		A popup appears with a form for the user to enter.			
14	Click on "Save".		The Assignment Decision will be added to the list of "Incoming Decisions" with the main information shown at the record.			

Validation and Business Rules:	Required fields must be entered. The system will check if the person already has an assignment and disallow it.
Post Conditions:	The Assignment Decision will be added to the list of "Incoming Decisions" with the main information shown at the record.

Structures for Criteria-Based Testing

finite state machine (FSM).

disjunctive normal form (DNF)



Advantages of Criteria-Based Test Design

- Criteria maximize the "bang for the buck"
  - Fewer tests that are more effective at finding faults
- Comprehensive test set with minimal overlap
- Traceability from software artifacts to tests
  - The "why" for each test is answered
  - Built-in support for regression testing
- A "stopping rule" for testing—advance knowledge of how many tests are needed
- Natural to automate

Criteria Summary

- Many companies still use "monkey testing"
  - A human sits at the keyboard, wiggles the mouse and bangs the keyboard
  - No automation
  - Minimal training required
- Some companies automate human-designed tests
- But companies that use both automation and criteria-based testing

Save money

Find more faults

Build better software

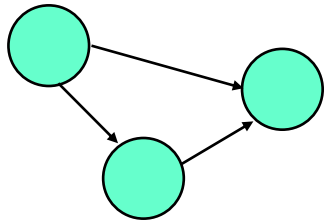
Criteria Based on Structures

Structures : Four ways to model software

1. Input Domain Characterization (sets)

```
A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}
```

2. Graphs



3. Logical Expressions

```
(not X or not Y) and A and B
```

4. Syntactic Structures (grammars)

```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

Graph Coverage

- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, each edge from one node to another

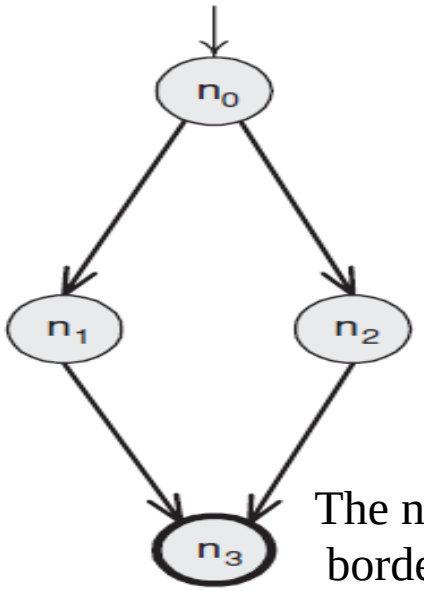
more than one initial node can be present; that is, N_0 is a set

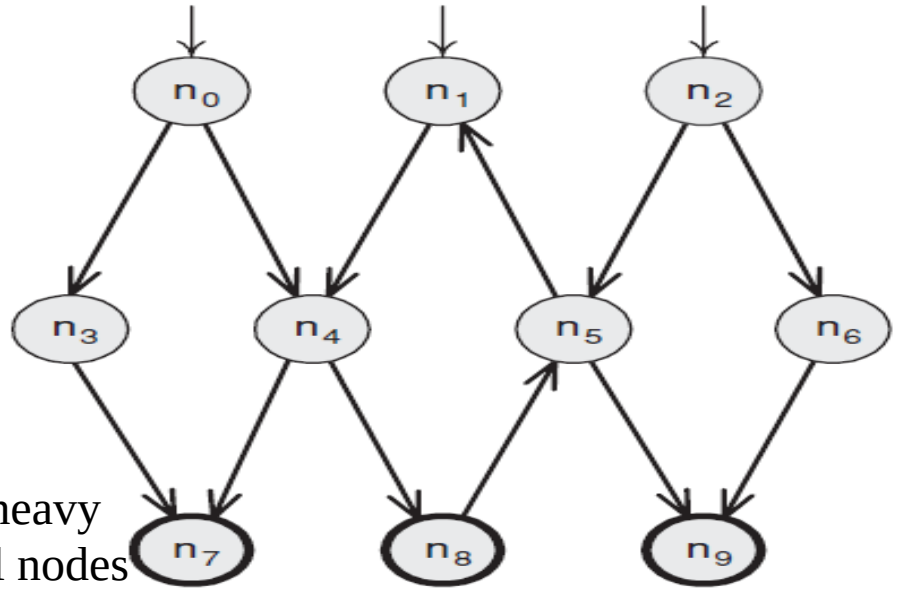
$N_0 = \{1\}$
$N_f = \{1\}$
E= {}

We always identify final nodes, and there must be at least one final node

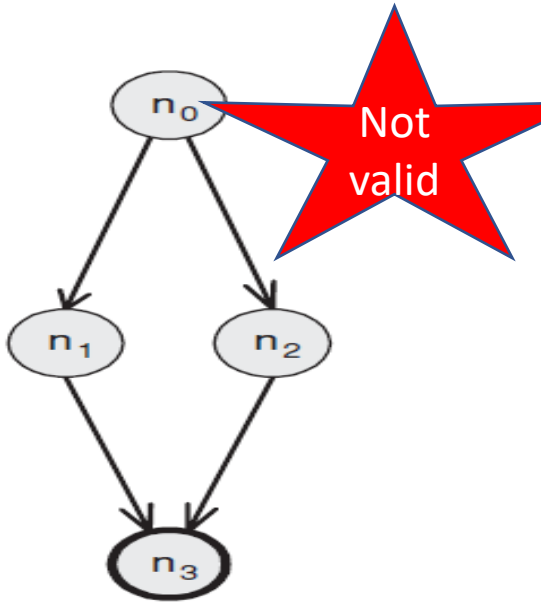
Example Graphs

Graph (a) has a single initial node, graph (b) multiple initial nodes, and graph (c) (rejected) with no initial nodes.





The nodes with heavy borders are final nodes

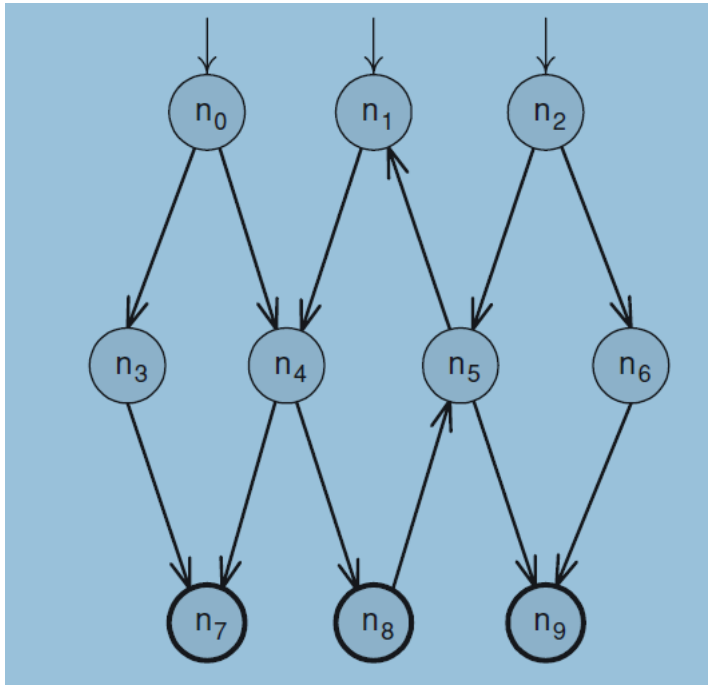


N0 = { 1 }
Nf = { 4 }
E = { (1, 2), (1,3), (2,4), (3,4) }

N0 = { 1, 2, 3 }
Nf = { 8, 9, 10 }
E = { (1,4), (1,5), (2,5), (3,6), (3, 7), (4, 8),
(5,8), (5,9), (6,2), (6,10), (7,10) (9,6) }

N0 = { }
Nf = { 4 }
E = { (1,2), (1,3), (2,4), (3,4) }

Paths in Graphs



Path Examples	
1	n_0, n_3, n_7
2	n_1, n_4, n_8, n_5, n_1
3	n_2, n_6, n_9

Invalid Path Examples	
1	n_0, n_7
2	n_3, n_4
3	n_2, n_6, n_8

(a) Path examples

Reachability Examples	
1	*reach* $(n_0) = N - \{n_2, n_6\}$
2	*reach* $(n_0, n_1, n_2) = N$
3	*reach* $(n_4) = \{n_1, n_4, n_5, n_7, n_8, n_9\}$
4	*reach* $([n_6, n_9]) = \{ n_9 \}$

(b) Reachability examples

the sequence $[n_0, n_7]$ is not a path because the two nodes are not connected by an edge.

A *path* is a sequence $[n_1, n_2, \ldots, n_M]$ of nodes, where each pair of adjacent nodes, (n_i, n_{i+1}), $1 \leq i < M$, is in the set E of edges. The length of a path is defined as the number of edges it contains.

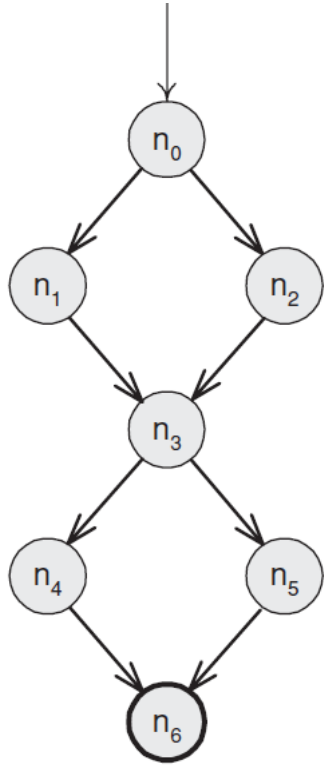
Test Path

- Basic graph algorithms, usually given in standard data structures texts, can be used to compute syntactic reachability.
- A test path represents the execution of a test case. The reason test paths must start in $N0$ is that test cases always begin from an initial node. It is important to note that a single test path may correspond to a very large number of test cases on the software. It is also possible that a test path may correspond to zero test cases if the test path is infeasible

Definition 2.31 Test path: **A path *p*, possibly of length zero, that starts at some node in N_0 and ends at some node in N_f.**

How Many Test Paths

Double-diamond graph Four test paths
[1, 2, 4, 5, 7]
[1, 2, 4, 6, 7]
[1, 3, 4, 5, 7]
[1, 3, 4, 6, 7]



single entry/single exit or *SESE* graphs

Syntactic vs Semantic Reach

- We say that a node *n* (or an edge *e*) is ***syntactically reachable*** from node *ni* if there exists a path from node *ni* to *n* (or edge *e*).
- A node *n* (or edge *e*) is also ***semantically reachable*** **if it is possible to execute at least one of the paths with some input**.
- We can define the function *reachG*(*x*) as the portion of a graph that is syntactically reachable from the parameter *x*. The parameter for *reachG*() can be a node, an edge, or a set of nodes or edges.

Testing and Covering Graphs

We use graphs in testing as follows :

- Develop a model of the software as a graph
- Require tests to visit/tour sets of nodes, edges or sub-paths

Visiting and Touring

Visit :

- A test path p visits node n if n is in p
- A test path p visits edge e if e is in p

Tour :

- A test path p tours subpath q if q is a subpath of p

, each edge is technically a subpath

For the Test path [1, 2, 4, 5, 7]

Visits nodes ? 1, 2, 4, 5, 7
Visits edges ? (1,2), (2,4), (4, 5), (5, 7)
Tours subpaths ? [1,2,4], [2,4,5], [4,5,7], [1,2,4,5], [2,4,5,7], [1,2,4,5,7]

Example

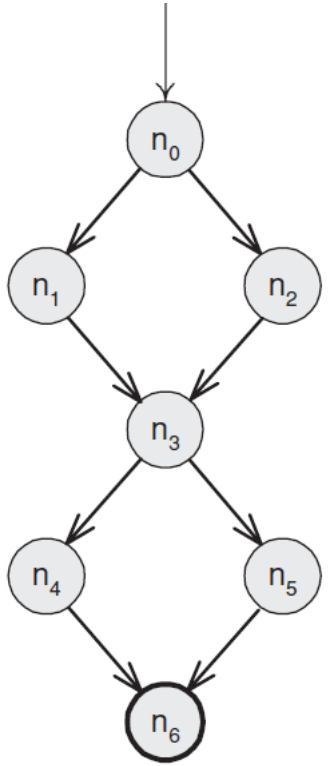
For the test path [*n*0, *n*1, *n*3, *n*4, *n*6]

visits

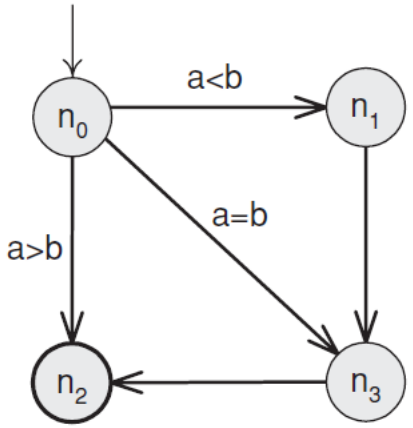
- nodes *n*0 and *n*1,
- visits edges (*n*0, *n*1) and (*n*3, *n*4)

Tours

the subpath [*n*1, *n*3, *n*4]



Mapping between test cases and test paths



(a) Graph for testing the case with input integers a, b and output (a+b)

Map to

Test case t_1 : (a=0, b=1) → [Test path p_1 : n_0, n_1, n_3, n_2]

Test case t_2 : (a=1, b=1) → [Test path p_2 : n_0, n_3, n_2]

Test case t_3 : (a=2, b=1) → [Test path p_3 : n_0, n_2]

Graph Coverage Criteria

- *Definition 2.32* ***Graph Coverage****:* Given a set *TR* of test requirements for a graph criterion *C*, a test set *T* satisfies *C* on graph *G* if and only if for every test requirement *tr* in *TR*, there is at least one test path *p* in *path*(*T*) such that *p* meets *tr*
  .

Example : Jelly Bean Coverage

Flavors :

1. Lemon
2. Pistachio
3. Cantaloupe
4. Pear
5. Tangerine
6. Apricot

Colors :

1. Yellow (Lemon, Apricot)
2. Green (Pistachio)
3. Orange (Cantaloupe, Tangerine)
4. White (Pear)

- Possible coverage criteria :
  1. Taste one jelly bean of each flavor
     - Deciding if yellow jelly bean is Lemon or Apricot is a controllability problem
  2. Taste one jelly bean of each color

Coverage Level

- T1 = { three Lemons, one Pistachio, two Cantaloupes, one Pear, one Tangerine, four Apricots }
- T2 = { One Lemon, two Pistachios, one Pear, three Tangerines }
- Does test set T1 satisfy the flavor criterion?
- Does test set T2 satisfy the flavor criterion?
- Does test set T2 satisfy the color criterion?

The ratio of the number of test requirements satisfied by T to the size of TR
T2 on the previous slide satisfies 4 of 6 test requirements

Coverage

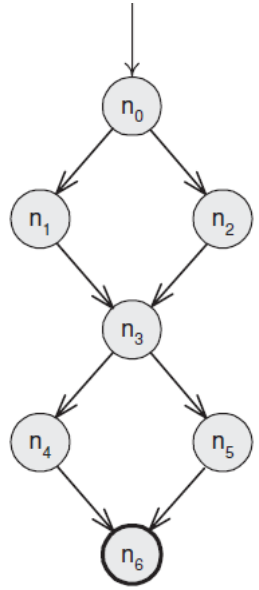
Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr*

- Infeasible test requirements : test requirements that cannot be satisfied
  - No test case values exist that meet the test requirements
  - Example: Dead code
  - Detection of infeasible test requirements is formally undecidable for most test criteria
- Thus, 100% coverage is impossible in practice

Structural Coverage Criteria

Node Coverage **(NC)**

- Criterion **2.1 Node Coverage (NC):** *TR contains each reachable node in G.*



$TR = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$.
Test path $p_1 = [n_0, n_1, n_3, n_4, n_6]$ meets the first, second,fourth, fifth, and seventh test requirements, and test path $p_2 = [n_0, n_2, n_3, n_5, n_6]$ meets the first, third, fourth, sixth, and seventh. Therefore, if a test set T contains $\{t_1, t_2\}$, where $path(t_1) = p_1$ and $path(t_2) = p_2$, then T satisfies node coverage on G.

Definition 2.34 Node Coverage (NC) *(Standard Definition):* Test set T satisfies node coverage on graph Gif and only if for every syntactically reachable node n in N, there is some path p in $path(T)$ such that p visits n.

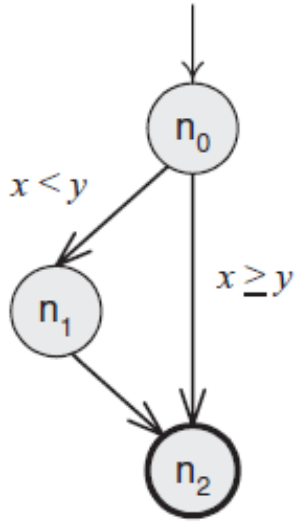
Edge Coverage (EC)

- Criterion **2.2 Edge Coverage (EC):** *TR contains each reachable path of length up to 1, inclusive, in G.*

$T_1 = \{ t_1 \}$
T_1 satisfies node coverage on the graph
(a) Node Coverage

$T_2 = \{ t_1 , t_2 \}$
T_2 satisfies edge coverage on the graph
(b) Edge Coverage

"if-else" structure.



path $(t_1) = [n_0, n_1, n_2]$
path $(t_2) = [n_0, n_2]$

Prime Path

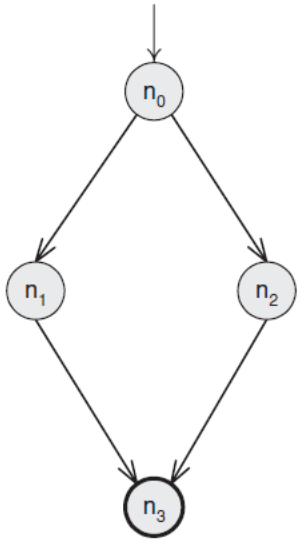
- ***Definition 2.35 Prime Path**:* A path from *ni* to *nj* is a prime path if it is a simple path and it does not appear as a proper subpath of any other simple path.

Simple Path :: Path Length

- Path length 0 : n0
- Path length 1: [n0,n1]
- What is the max path length in
- Figure a: max =2
- Graph (b) have infinite length
- Simple path when a visited node

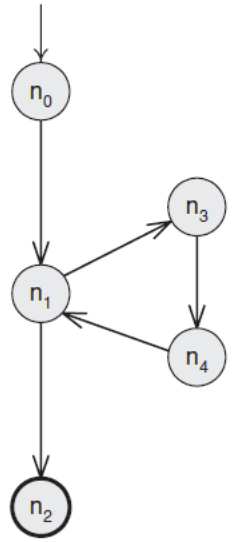
never been visited again

What about the path [n1,n3,n4]



Prime Paths = { $[n_0, n_1, n_3]$, $[n_0, n_2, n_3]$ }
path (t_1) = $[n_0, n_1, n_3]$
path (t_2) = $[n_0, n_2, n_3]$
$T_1 = \{t_1, t_2\}$
T_1 satisfies prime path coverage on the graph

(a) Prime Path Coverage on a Graph with No Loops



Prime Paths = { $[n_0, n_1, n_2]$,
$[n_0, n_1, n_3, n_4]$, $[n_1, n_3, n_4, n_1]$,
$[n_3, n_4, n_1, n_3]$, $[n_4, n_1, n_3, n_4]$,
$[n_3, n_4, n_1, n_2]$ }
path (t_3) = $[n_0, n_1, n_2]$
path (t_4) = $[n_0, n_1, n_3, n_4, n_1, n_3, n_4, n_1, n_2]$
$T_2 = \{t_3, t_4\}$
T_2 satisfies prime path coverage on the graph

(b) Prime Path Coverage on a Graph with Loops

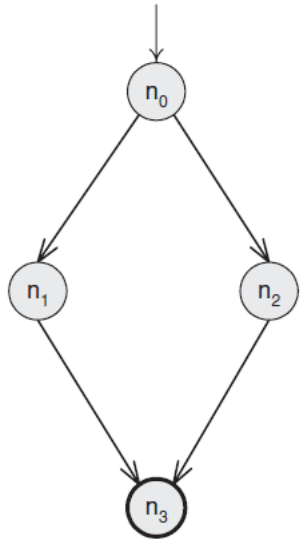
Find Simple Path in (a) and (b)

- A:
- [n0,n1,n3], [n0,n2,n3]
- B:
- [n0,n1,n3], [n0,n1,n3,n4]
- [n1,n3,n4,n1] simple because first node equals last node

[n3,n4,n1,n3], [n4,n1,n3,n4]

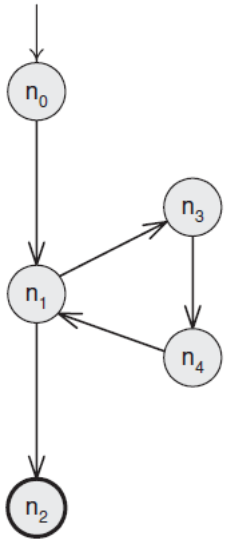
[n0,n1,n3]

[n1,n3,n4]??? {repeated}



Prime Paths = { $[n_0, n_1, n_3]$, $[n_0, n_2, n_3]$ }
path (t_1) = $[n_0, n_1, n_3]$
path (t_2) = $[n_0, n_2, n_3]$
$T_1 = \{t_1, t_2\}$
T_1 satisfies prime path coverage on the graph

(a) Prime Path Coverage on a Graph with No Loops



Prime Paths = { $[n_0, n_1, n_2]$,
$[n_0, n_1, n_3, n_4]$, $[n_1, n_3, n_4, n_1]$,
$[n_3, n_4, n_1, n_3]$, $[n_4, n_1, n_3, n_4]$,
$[n_3, n_4, n_1, n_2]$ }
path (t_3) = $[n_0, n_1, n_2]$
path (t_4) = $[n_0, n_1, n_3, n_4, n_1, n_3, n_4, n_1, n_2]$
$T_2 = \{t_3, t_4\}$
T_2 satisfies prime path coverage on the graph

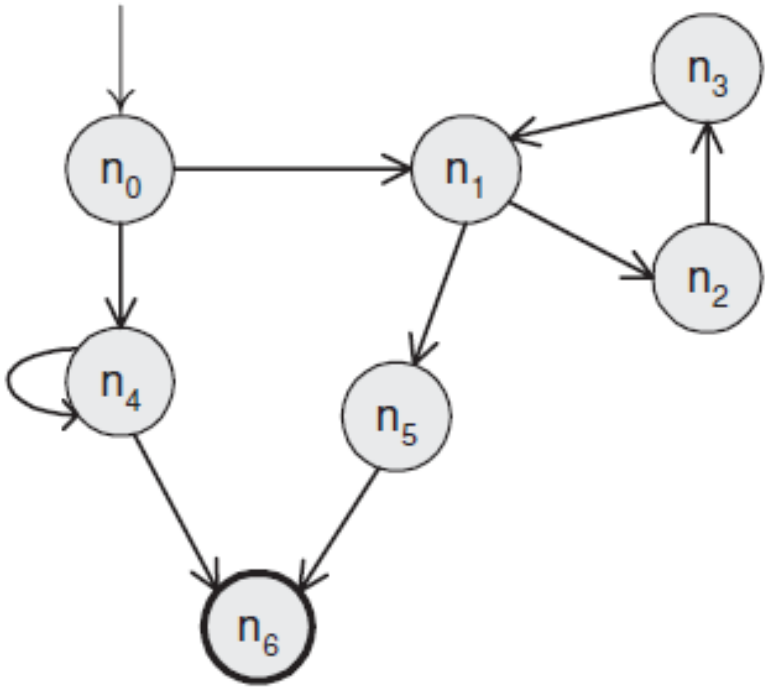
(b) Prime Path Coverage on a Graph with Loops

Prime Path Coverage (PPC)

- Criterion **2.4 Prime Path Coverage (PPC):** *TR contains each prime path in G.*
- A ***round trip*** **path is a prime** path of nonzero length that starts and ends at the same node. One type of round trip test coverage requires at least one round trip path to be taken for each node, and another requires all possible round trip paths
- Criterion **2.5 Simple Round Trip Coverage (SRTC):** *TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*
- Criterion **2.6 Complete Round Trip Coverage (CRTC):** *TR contains all roundtrip paths for each reachable node in G.*

Exercise: Finding Prime Test Paths

- Consider the example graph in Figure It has seven nodes and nine edges, including a loop and an edge from node *n*4 to itself (sometimes called a “self-loop.”) .Prime paths can be found b starting with paths of length 0, then extending to length 1, an so on. Such an algorithm collects all simple paths, whether prime or not. The prime paths can be easily screened from th set.



Solution…

Simple paths of length 0 (7):
1) [0]
2) [1]
3) [2]
4) [3]
5) [4]
6) [5]
7) [6] !

Simple paths of length 1 (9):
8) [0, 1]
9) [0, 4]
10) [1, 2]
11) [1, 5]
12) [2, 3]
13) [3, 1]
14) [4, 4] *
15) [4, 6] !
16) [5, 6] !

Simple paths of length 2 (8):
17) [0, 1, 2]
18) [0, 1, 5]
19) [0, 4, 6] !
20) [1, 2, 3]
21) [1, 5, 6] !
22) [2, 3, 1]
23) [3, 1, 2]
24) [3, 1, 5]

Simple paths of length 3 (7):
25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
29) [2, 3, 1, 5]
30) [3, 1, 2, 3] *
31) [3, 1, 5, 6] !

Prime paths of length 4 (1):
32) [2, 3, 1, 5, 6]!

The prime paths can be computed by eliminating any path that is a (proper) subpath of some other simple path. Note that every simple path without an exclamation mark or asterisk is eliminated as it can be extended and is thus a proper subpath of some other simple path. There are eight prime paths:

Solution…continued

There are eight prime paths:
14) [4, 4] *
19) [0, 4, 6] !
25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
30) [3, 1, 2, 3] *
32) [2, 3, 1, 5, 6]!

Reference

- INTRODUCTION TO SOFTWARE TESTING by Paul Ammann- George Mason University and Jeff Offutt- George Mason University
- (http://www.cs.gmu.edu/~offutt/softwaretest)