

UNIT-IV
KCS-601
SOFTWARE ENGINEERING
PART-I

Software Engineering

Syllabus

Unit-IV: Software Testing

Testing Objectives, Unit Testing, Integration Testing, Acceptance Testing, Regression Testing, Testing for Functionality and Testing for Performance, Top-Down and Bottom-Up Testing Strategies: Test Drivers and Test Stubs, Structural Testing (White Box Testing), Functional Testing (Black Box Testing), Test Data Suit Preparation, Alpha and Beta Testing of Products.

Static Testing Strategies: Formal Technical Reviews (Peer Reviews), Walk Through, Code Inspection, Compliance with Design and Coding Standards.

INTRODUCTION To TESTING SOFTWARE TESTING

- ❑ Software Testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free.
- ❑ The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

INTRODUCTION To TESTING SOFTWARE TESTING

□ What is it?

- ▣ Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer.
- ▣ Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter in to the picture.

□ Who does it?

- ▣ During early stages of testing, a soft-ware engineer performs all tests.
- ▣ However, as the testing process progresses, testing specialists may become involved.

INTRODUCTION To TESTING SOFTWARE TESTING

□ Why is it important?

- ▣ Reviews and other SQA activities can and do uncover errors, but they are not sufficient.
- ▣ **Software Testing is Important** because if there are any bugs or errors in the software, it can be identified early and can be solved before delivery of the software product. Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.

BENEFITS OF SOFTWARE TESTING

- ❑ **Cost-Effective:** It is one of the important advantages of software testing. Testing any IT project on time helps you to save your money for the long term. In case if the bugs caught in the earlier stage of software testing, it costs less to fix.
- ❑ **Security:** It is the most vulnerable and sensitive benefit of software testing. People are looking for trusted products. It helps in removing risks and problems earlier.
- ❑ **Product quality:** It is an essential requirement of any software product. Testing ensures a quality product is delivered to customers.
- ❑ **Customer Satisfaction:** The main aim of any product is to give satisfaction to their customers. UI/UX Testing ensures the best user experience

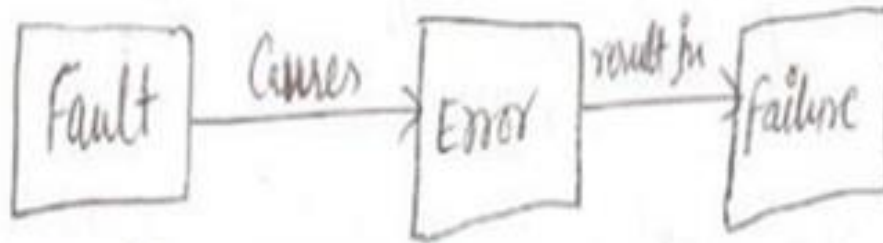
INTRODUCTION To TESTING SOFTWARE TESTING

- ❑ **Software testing has been defined as-**
 - ❑ Testing is the process of executing a program with the aim of finding errors. To make our software perform well it should be error-free. If testing is done successfully it will remove all the errors from the software.
 - ❑ **“The Process of analyzing a program with the intent of finding errors”.**

OR

- ❑ **“Testing is the Process of executing a program with the intent of finding errors”.**

Basic Terminology Related to s/w Testing



Fault → is a defect within a system

Error → is observed by a deviation from the expected behaviour of the system.

failure → occurs when the system can no longer perform as required
(does not meet specification)

Basic Terminology Related to s/w Testing

- ❑ **Bug**: Bug is the informal name of defects, which means that software or application is not working as per the requirement.
- ❑ **Test**: Act of exercising software with test cases.
- ❑ **Test-Case**: A set of input values, execution, preconditions, expected results and execution, post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.
- ❑ It describes an input description and an expected output description.
- ❑ **Test-Suite**: The set of test cases is called a test suite. we may have suite of all possible test case.

Basic Terminology Related to s/w Testing

- **Bug:** if Fault is a source code, we call it a bug, these are mistakes done by developers during coding.
- **Test:** Act of exercising software with test cases.
- **Test-Case:** A set of input values ,execution, preconditions, expected results and execution, post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.
- It describes an input description and an expected output description.
- **Test-Suite:** The set of test cases is called a test suite. we may have suite of all possible test case.

TESTING PRINCIPLES

- There are many principles that guide software testing. Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing.
- The following are the main principles for testing:
 1. **All tests should be traceable to customer requirements**-This is in order to uncover any defects that might cause the program or system to fail to meet the client's requirements.

TESTING PRINCIPLES

2. **Tests should be planned long before testing begins-**
Soon after the requirements model is completed, test planning can begin. Detailed test cases can begin as soon as the design model is designed.
3. **Testing should begin “in the small” and progress toward testing “in the large- ”**The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

TESTING PRINCIPLES

5. **Exhaustive testing is not possible-** The number of path permutations for even a moderately-sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
6. **To be most effective, testing should be conducted by an independent third party-** The software engineer who has created the system is not the best person to conduct all tests for the software.
7. **Testing Time And Resource are Limited-** So we should avoid redundant test.

TESTING OBJECTIVES

- The testing objective is to test the code, whereby there is a high probability of discovering all errors.
- This objective also demonstrates that the software functions are working according to software requirements specification (SRS) with regard to functionality, features, facilities, and performance.
- Testing objectives are-
 1. Testing is a process of executing a program with the intent of finding an error.
 2. A good test case is one that has a high probability of finding an as-yet-undiscovered error.
 3. A successful test is one that uncovers an as-yet-undiscovered error.

Verification	Validation
Are we building the system right?	Are we building the right system?
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	Validation is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.	The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
Following activities are involved in Verification: Reviews, Meetings and Inspections.	Following activities are involved in Validation: Testing like black box testing, white box testing, gray box testing etc.
Verification is carried out by QA team to check whether implementation software is as per specification document or not.	Validation is carried out by testing team.

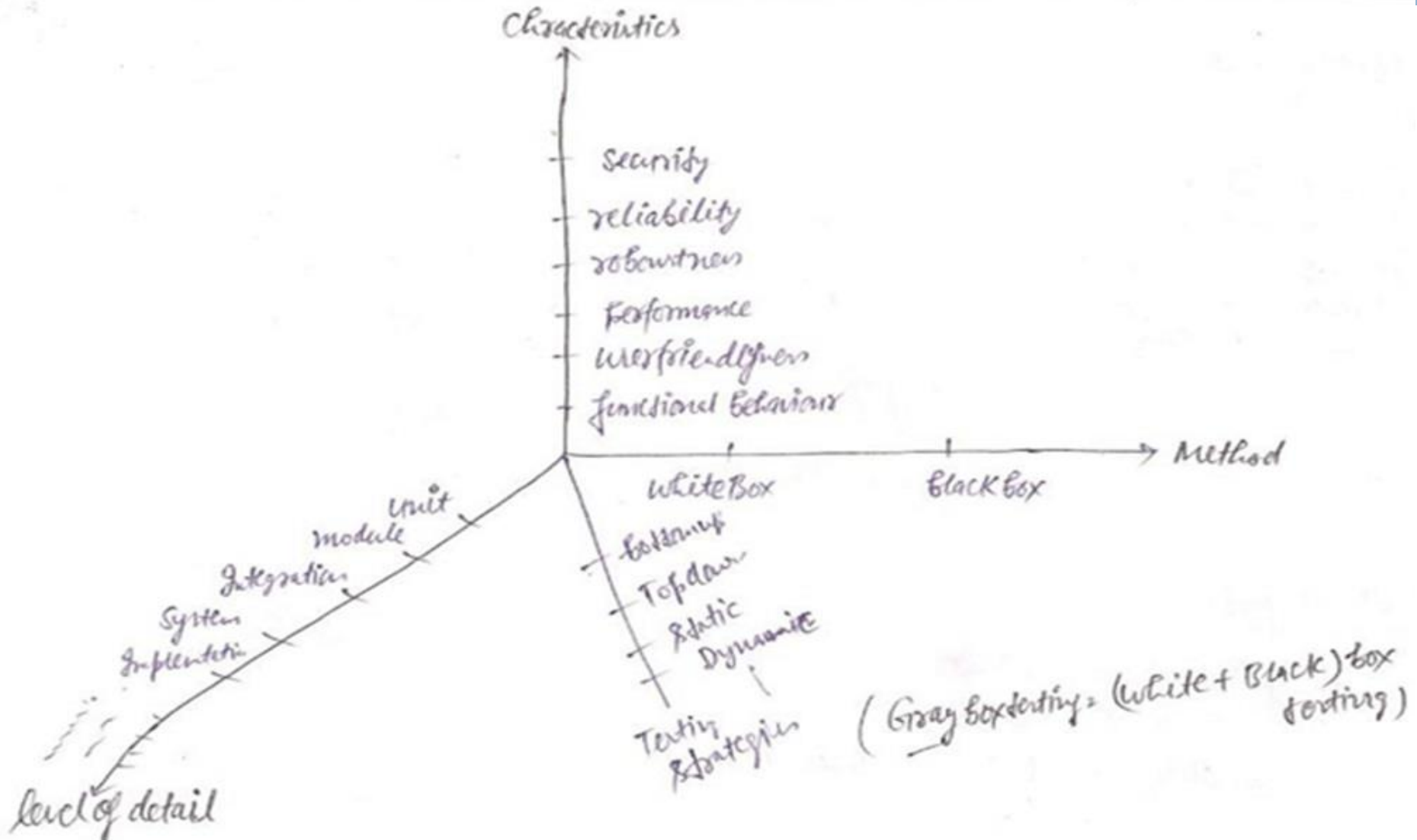
Execution of code is not comes under Verification.	Execution of code is comes under Validation.
Verification process explains whether the outputs are according to inputs or not.	Validation process describes whether the software is accepted by the user or not.
Verification is carried out before the Validation.	Validation activity is carried out just after the Verification.
Following items are evaluated during Verification: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,	Following item is evaluated during Validation: Actual product or Software under test.
Cost of errors caught in Verification is less than errors found in Validation.	Cost of errors caught in Validation is more than errors found in Verification.
It is basically manually checking the of documents and files like requirement specifications etc.	It is basically checking of developed program based on the requirement specifications documents & files.

Conclusion on difference of *Verification and Validation in software testing*:

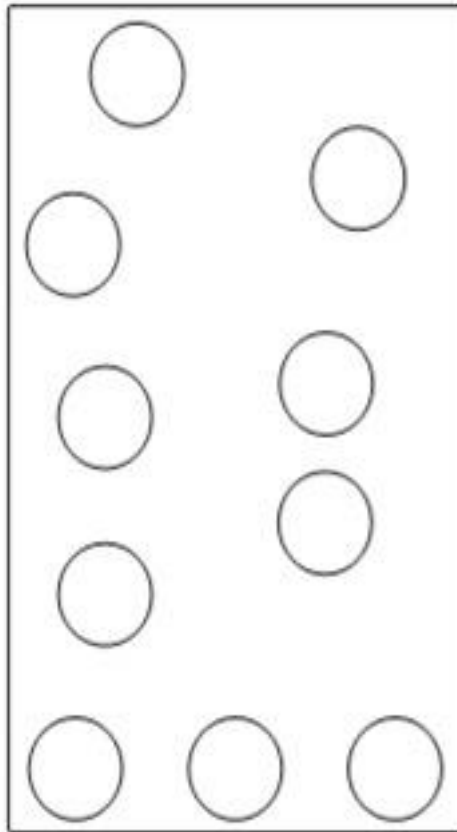
1. Both Verification and Validation are essential and balancing to each other.
2. Different error filters are provided by each of them.
3. Both are used to find a defect in different way, Verification is used to identify the errors in requirement specifications & validation is find the defects in the implemented Software application.

**Error detection techniques= Verification techniques
+
Validation techniques**

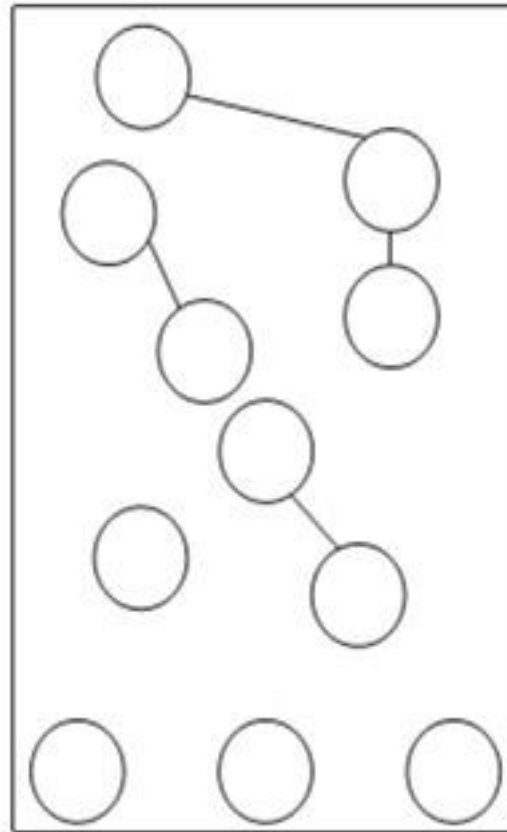
Testing Types



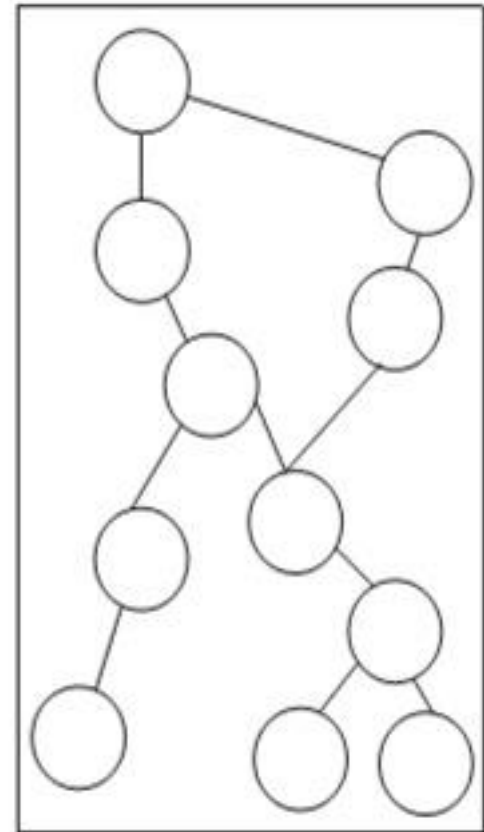
Level of Testing



UNIT TESTING



INTEGRATION TESTING



SYSTEM TESTING

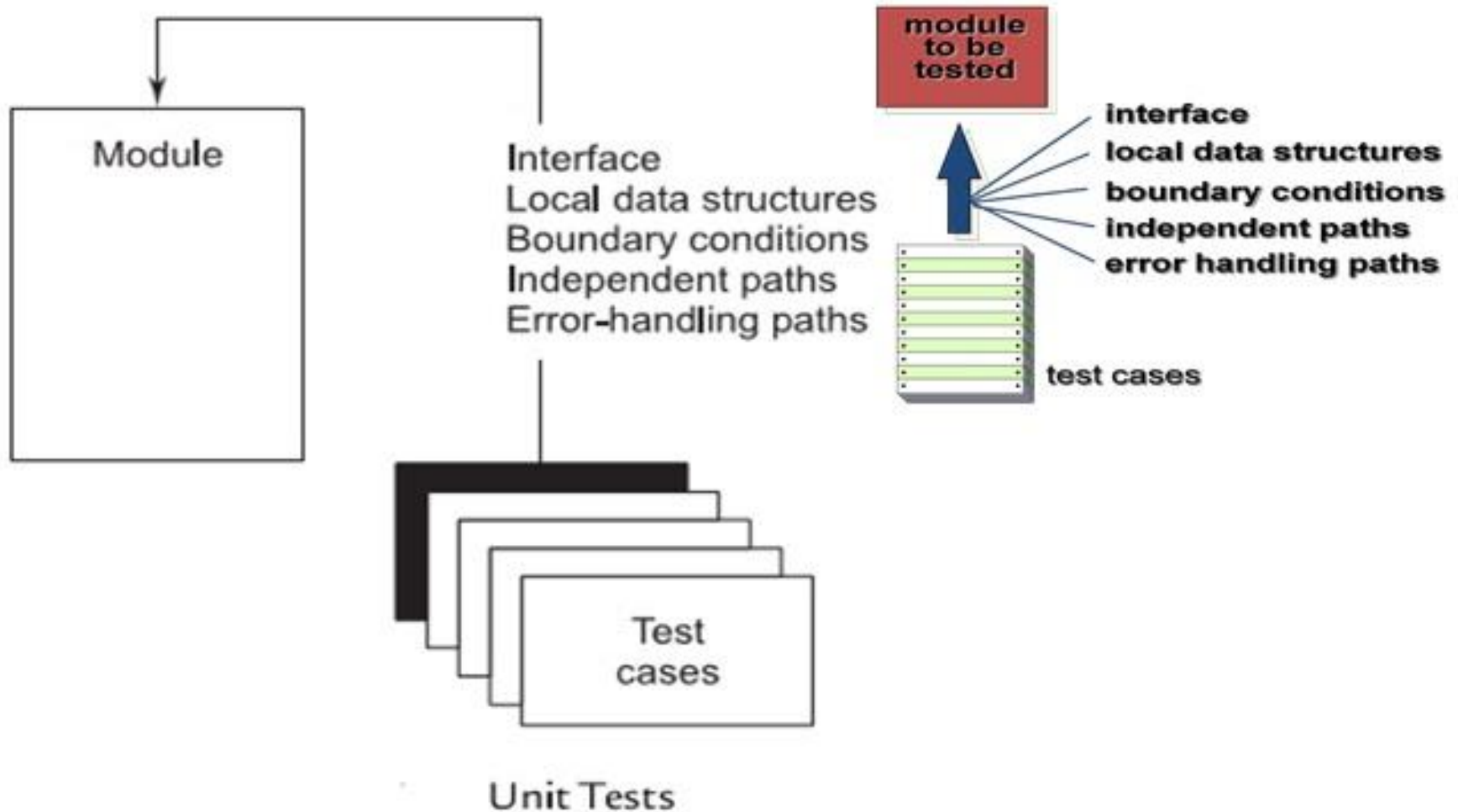
Unit Testing

- In unit testing individual components are tested to ensure that they operate correctly.
- It focuses on verification effort. On the smallest unit of software design, each component is tested independently without other system components.
- There are a number of reasons to do unit testing rather than testing the entire product-
 - The size of a single module is small enough that we can locate an error fairly easily.
 - The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
 - Confusing interactions of multiple errors in widely different parts of the software are eliminated.

Unit Test Consideration

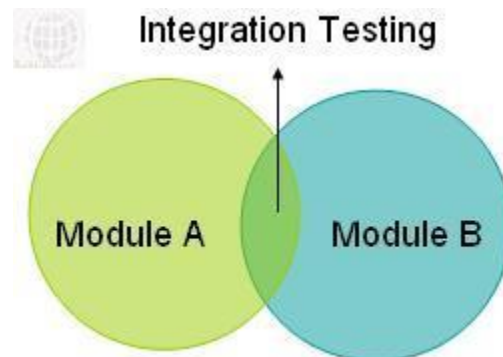
- The module interface is tested to ensure that information properly flows into and out of the program unit under testing.
- The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- And finally, all error-handling paths are tested.

Unit Test Consideration



Integration Testing

- Integration testing is the second level of the software testing process comes after unit testing. In this testing, units or individual components of the software are tested in a group.
- **INTEGRATION TESTING** is defined as a type of testing where software modules are integrated logically and tested as a group. A typical software project consists of multiple software modules, coded by different programmers.



Integration Testing

- Software Engineering defines variety of strategies to execute Integration testing, viz.
- Big Bang Approach :
- Incremental Approach: which is further divided into the following
 - ▣ Top Down Approach
 - ▣ Bottom Up Approach
 - ▣ Sandwich Approach – Combination of Top Down and Bottom Up

Integration Testing

- **Integration test approaches –**

There are four types of integration testing approaches. Those approaches are the following:

- 1. Big-Bang Integration Testing –**

It is the simplest integration testing approach, where all the modules are combining and verifying the functionality after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If once an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. So, debugging errors reported during big bang integration testing are very expensive to fix.

- **Advantages:**

- It is convenient for small systems.

- **Disadvantages:**

- There will be quite a lot of delay because you would have to wait for all the modules to be integrated.
- High risk critical modules are not isolated and tested on priority since all modules are tested at once.

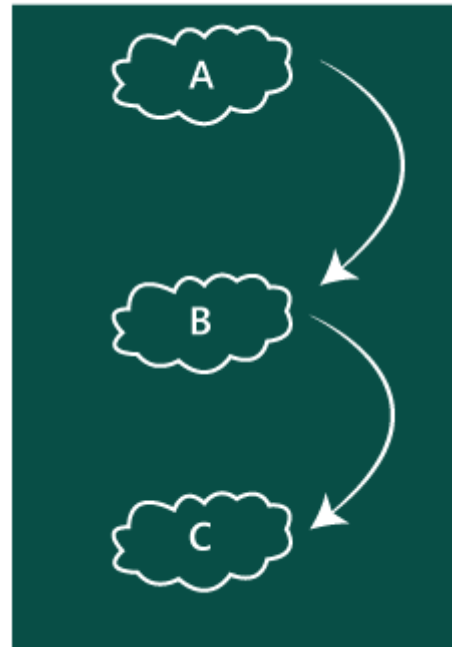
For example: Suppose we have a Flipkart application, we will perform incremental integration testing, and the flow of the application would like this:

Flipkart → Login → Home → Search → Add cart → Payment → Logout

Incremental integration testing is carried out by further methods:

Top-Down approach

Top-Down Approach



Bottom Up approach

Integration Testing

- **Bottom-Up Integration Testing –**
The bottom to up testing strategy deals with the process in which lower level modules are tested with higher level modules until the successful completion of testing of all the modules.
- Top level critical modules are tested at last, so it may cause a defect. Or we can say that we will be adding the modules from **bottom to the top** and check the data flow in the same order.
- **Advantages**
- Identification of defect is easy.
- Do not need to wait for the development of all the modules as it saves time.
- **Disadvantages**
- Critical modules are tested last due to which the defects can occur.
- There is no possibility of an early prototype.

Integration Testing

- **Top-Down Integration Testing –**
The top-down testing strategy deals with the process in which higher level modules are tested with lower level modules until the successful completion of testing of all the modules.
- **Advantages:**
- Identification of defect is difficult.
- An early prototype is possible.
- **Disadvantages:**
- Due to the high number of stubs, it gets quite complicated.
- Lower level modules are tested inadequately.
- Critical Modules are tested first so that fewer chances of defects.
- .

Integration Testing

Mixed Integration Testing –

A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. A mixed integration testing is also called sandwiched integration testing.

Advantages:

Mixed approach is useful for very large projects having several sub projects. This Sandwich approach overcomes this shortcoming of the top-down and bottom-up approaches.

Disadvantages:

For mixed integration testing, require very high cost because one part has Top-down approach while another part has bottom-up approach. This integration testing cannot be used for smaller system with huge interdependence between different modules.

Top-Down & Bottom Up Testing Approach Using Stubs And Drivers[Important]

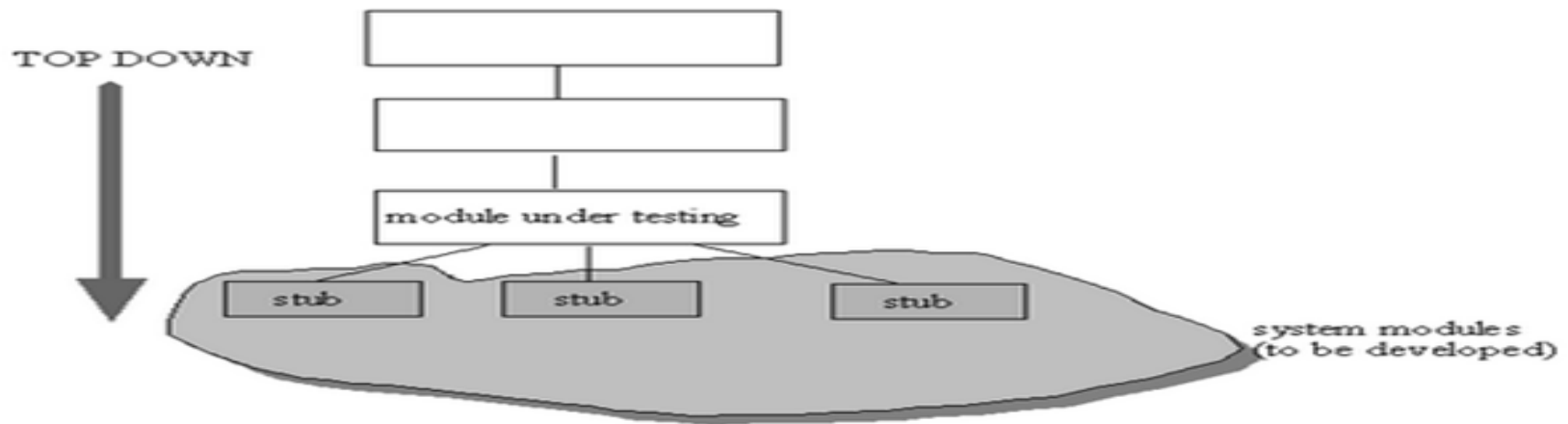


Fig. Top-Down System Testing

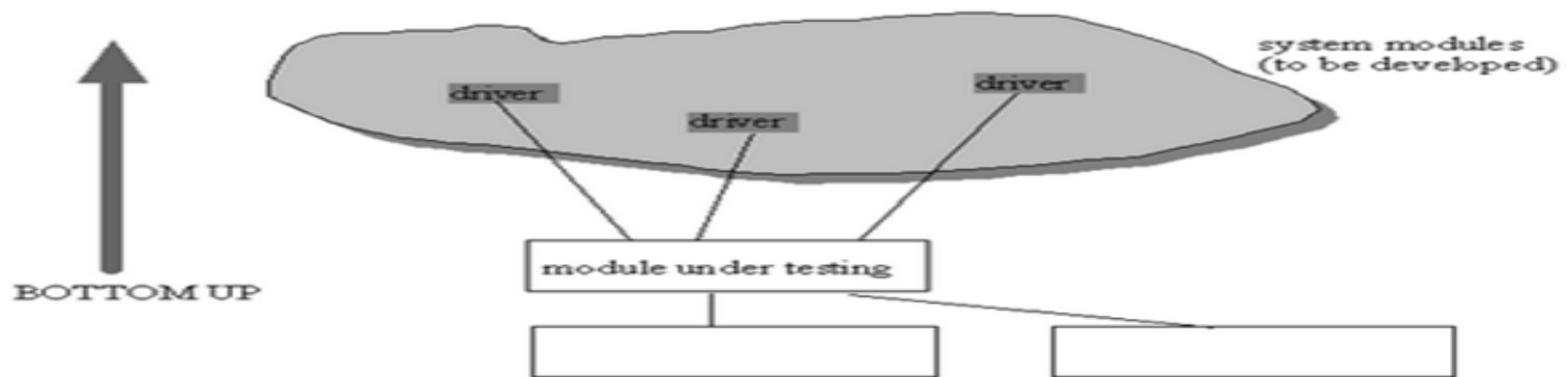


Fig. Bottom-Up System Testing

Stubs and Drivers

Stubs and Drivers are the dummy programs in Integration testing used to facilitate the software testing activity. These programs act as a substitutes for the missing models in the testing. They do not implement the entire programming logic of the software module but they simulate data communication with the calling module while testing.

Stub: Is called by the Module under Test.

Driver: Calls the Module to be tested.

System Testing

- Subsystems are integrated to make up the entire system.
- The testing process is concerned with finding errors that result from unanticipated interactions between subsystems and system components.
- It is also concerned with validating that the system meets its functional and non-functional requirements.
- There are essentially three main kinds of system testing:
 - ▣ **Alpha testing**
 - ▣ **Beta testing**
 - ▣ **Acceptance testing**

1-Alpha Testing

Alpha Testing. Alpha testing refers to the system testing carried out by the test team within the development organization.

The alpha test is conducted at the developer's site by the customer under the project team's guidance. In this test, users test the software on the development platform and point out errors for correction. However, the alpha test, because a few users on the development platform conduct it, has limited ability to expose errors and correct them. Alpha tests are conducted in a controlled environment. It is a simulation of real-life usage. Once the alpha test is complete, the software product is ready for transition to the customer site for implementation and development.

2-Beta Testing

***Beta Testing.** Beta testing is the system testing performed by a selected group of friendly customers.*

If the system is complex, the software is not taken for implementation directly. It is installed and all users are asked to use the software in testing mode; this is not live usage. This is called the beta test. Beta tests are conducted at the customer site in an environment where the software is exposed to a number of users. The developer may or may not be present while the software is in use. So, beta testing is a real-life software experience without actual implementation. In this test, end users record their observations, mistakes, errors, and so on and report them periodically.

In a beta test, the user may suggest a modification, a major change, or a deviation. The development has to examine the proposed change and put it into the change management system for a smooth change from just developed software to a revised, better software. It is standard practice to put all such changes in subsequent version releases.

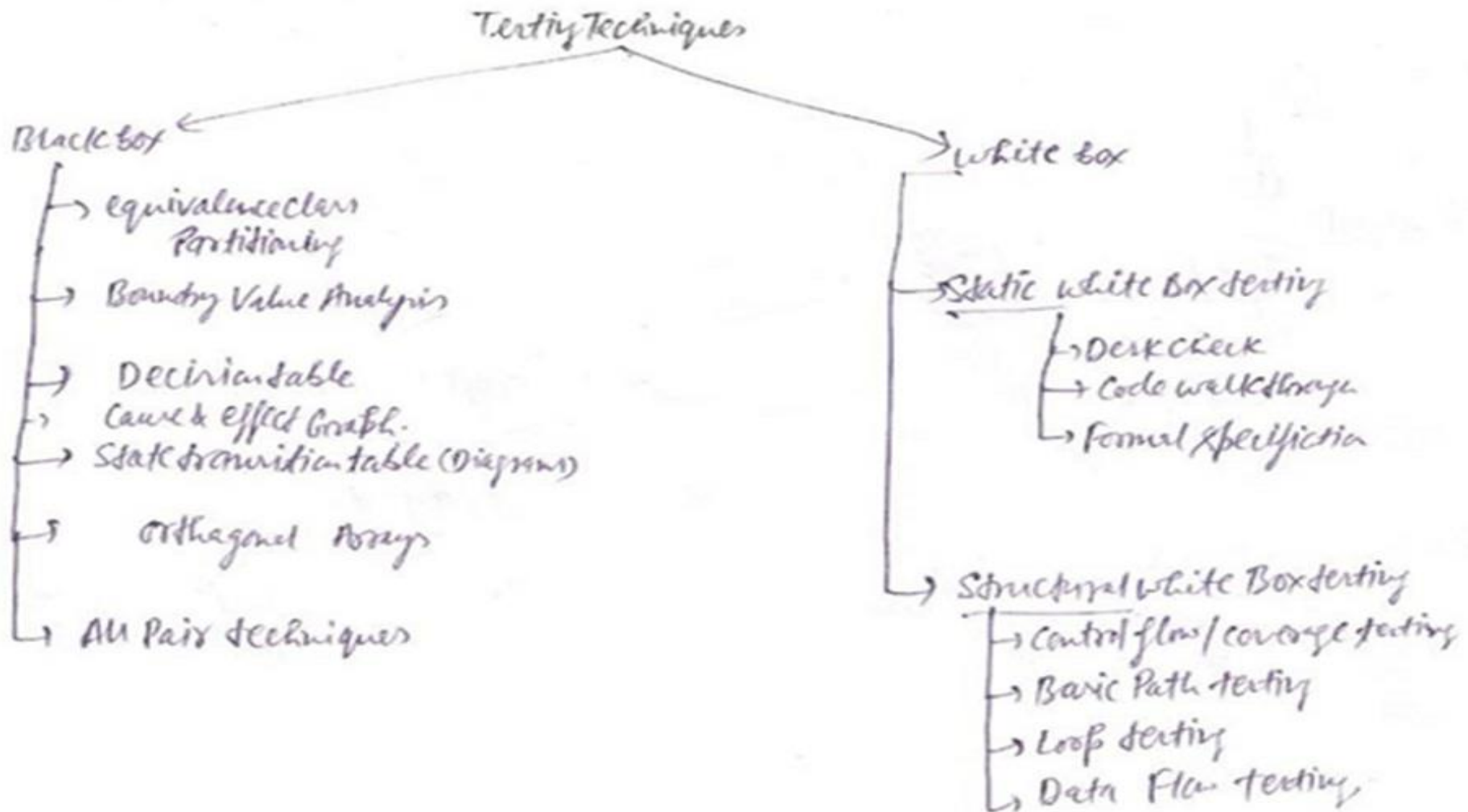
3-Acceptance Testing

Acceptance Testing. *Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system.*

When customer software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than the software engineers, an acceptance test can range from an informal 'test drive' to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

- It is conducted by business owners, the purpose of acceptance testing is to test whether the system does in fact, meet their business requirements.
- It is a level of the testing where a system is tested for acceptability.

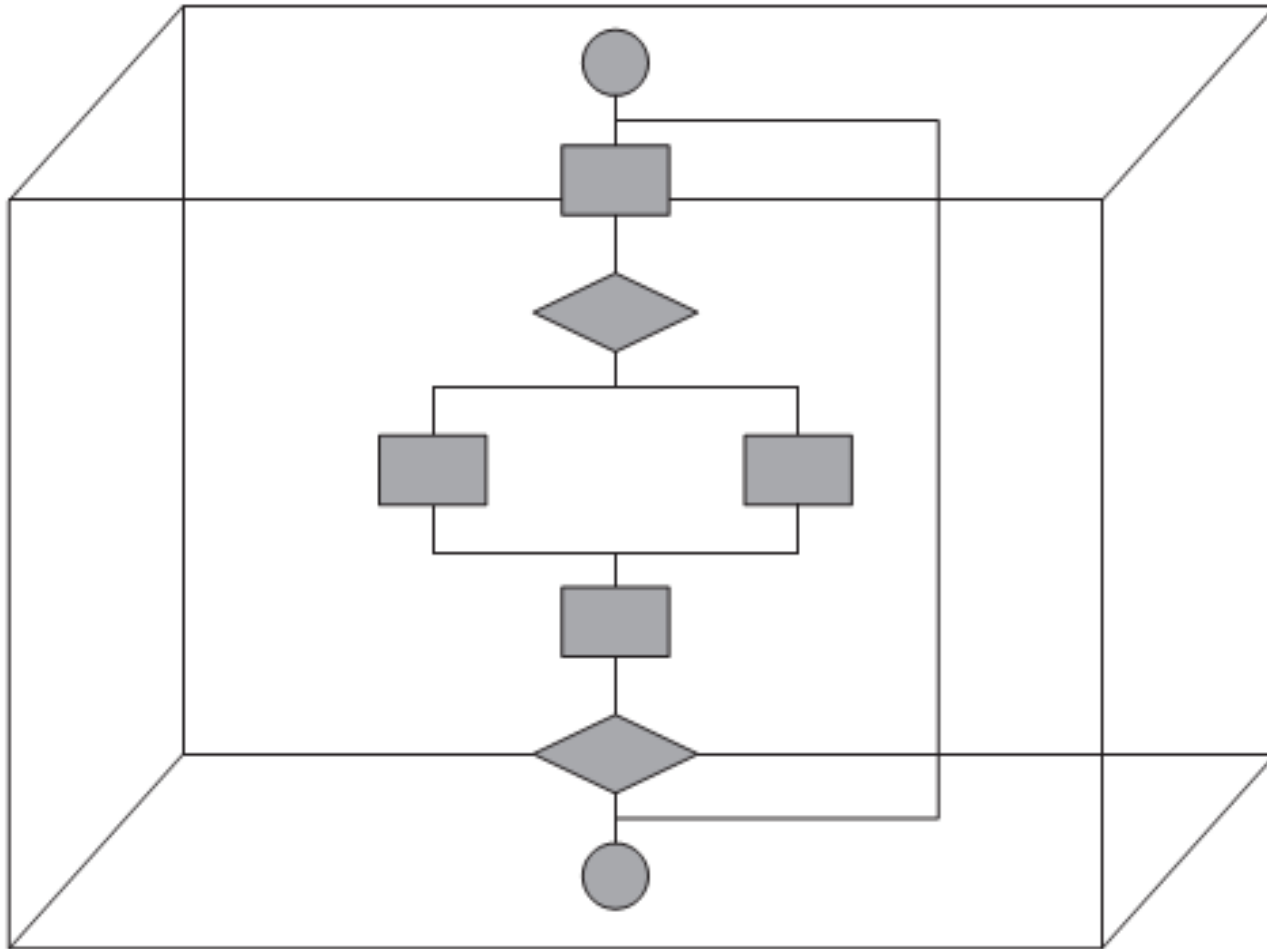
Testing Techniques



White Box Testing(Structural Testing)

- A complementary approach to functional or black-box testing is called structural or white-box testing.
- In this approach, test groups must have complete knowledge of the internal structure of the software.
- We can say structural testing is an approach to testing where the tests are derived from knowledge of the software's structure and implementation.
- The analysis of the code can be used to find out how many test cases are needed to guarantee that all of the statements in the program are executed at least once during the testing process.

White Box Testing



White Box Testing

Using white-box testing methods the software engineer can test cases that:

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decision on their true and false sides.
- Exercise all loops at their boundaries.
- Exercise internal data structures to ensure their validity.

Advantages of Structural/White-box Testing

The various advantages of white-box testing include:

- Forces test developer to reason carefully about implementation.
- Approximates the partitioning done by execution equivalence.
- Reveals errors in hidden code.

1-Static White box Testing

Static testing is a type of testing which requires only the source code of the product, not the binaries or executables. Static testing does not involve executing the programs on computers but involves select people going through the code to find out whether

- The code works according to the functional requirement;
- The code has been written in accordance with the design developed earlier in the project life cycle;
- The code for any functionality has been missed out;
- The code handles errors properly.

Static testing can be done by humans or with the help of specialized tools.

Static Testing by Humans

These methods rely on the principle of humans reading the program code to detect errors rather than computers executing the code to find errors. There are multiple methods to achieve static testing by humans. They are as follows:

1. **Desk checking:** It is done manually by the author of the code, desk checking is a method to verify the portions of the code for correctness. Such verification is done by comparing the code with the design or specifications to make sure that the code does what it is supposed to do and effectively.
2. **Code walkthrough:** This method and formal inspection (described in the next section) are group-oriented methods. Walkthroughs are less formal than inspections. The line drawn in formalism between walkthroughs and inspections is very thin and varies from organization to organization. The advantage that walkthrough has over desk checking is that it brings multiple perspectives. In walkthroughs, a set of people look at the program code and raise questions for the author. The author explains the logic of the code, and answers the questions. If the author is unable to answer some questions, he or she then takes those questions and finds their answers.

Static Testing by Humans

- 3. Formal inspection:** Code inspection-also called Fagan Inspection (named after the original formulator) - is a method, normally with a high degree of formalism. The focus of this method is to detect all faults, violations, and other side-effects. This method increases the number of defects detected by demanding thorough preparation before an inspection/review;
 - Enlisting multiple diverse views;
 - Assigning specific roles to the multiple participants; and
 - Going sequentially through the code in a structured manner.

Roles in Inspection

- 1. Author of the code:** Basic goal should be to learn as much as possible with regard to improving the quality of the document
- 2. Moderator:** The person who is expected to formally run the inspection according to the process.
- 3. Inspectors:** These are the people who actually provides, review comments for the code. There are typically multiple inspectors.
- 4. Scribe:** The person who takes detailed notes during the inspection meeting and circulates them to the inspection team after the meeting.

2-Structural testing

Structural testing takes into account the code, code structure, internal design, and how they are coded. The fundamental difference between structural testing and static testing is that in structural testing tests are actually run by the computer on the built product, where as in static testing, the product is tested by humans using just the source code and not the executables or binaries.

Structural testing entails running the actual product against some predesigned test cases to exercise as much of the code as possible or necessary. A given portion of the code is exercised if a test case causes the program to execute that portion of the code when running the test.

2.1-Code Coverage Testing

Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by testing. The percentage of code covered by a test is found by adopting a technique called instrumentation of code. There are specialized tools available to achieve instrumentation. Instrumentation rebuilds the product, linking the product with a set of libraries provided by the tool vendors. Code coverage testing is made up of the following types of coverage.

- Statement coverage
- Branch Coverage
- Condition coverage
- Path coverage
- Function coverage

A-Statement/Line Coverage

Statement coverage refers to writing test cases that execute each of the program statements. One can start with the assumption that more the code covered, the better is the testing of the functionality. It is called as Line Coverage.

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

Consider code sample 1:

READ X

READ Y

IF X>Y THEN Z = 0

ENDIF

To achieve 100% statement coverage of this code segment just one test case is required, one which ensures that variable A contains a value that is greater than the value of variable Y, for example, X = 12 and Y = 10

Consider code sample 2:

```
1 READ X
2 READ Y
3 Z =X + 2*Y
4 IF Z> 50 THEN
5 PRINT large Z
6 ENDIF
```

Let's analyze the coverage of a set of tests on our six-statement program:

TEST SET 1

- Test 1_1: X= 2, Y = 3
- Test 1_2: X =0, Y = 25
- Test 1_3: X =47, Y = 1

In Test 1_1, 2, 3, the value of Z will be 8, 50, and 49. So we will cover the statements on lines 1 to 4 and line 6. Since we have covered five out of six statements, we have 83% statement coverage. How about this one:

Test 1_4: X = 20, Y = 25

This time the value of Z is 70, so we will print 'Large Z' and we will have exercised all six of the statements, so now statement coverage = 100%. Notice that we measured coverage first, and then designed a test to cover the statement that we had not yet covered.

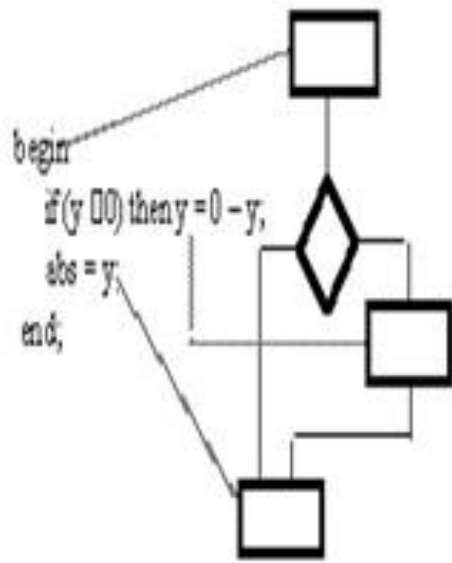


Fig. Statement coverage testing

B-Branch /Decision Coverage

Whenever there are two or more possible exits from the statement like an IF statement, a DO-WHILE or a CASE statement it is known as decision because in all these statements there are two outcomes, either TRUE or FALSE.

With the loop control statement like DO-WHILE or IF statement the outcome is either TRUE or FALSE and decision coverage ensures that each outcome (i.e TRUE and FALSE) of control statement has been executed at least once.

The formula to calculate decision coverage is:

Decision Coverage = (Number of decision outcomes executed/Total number of decision outcomes)*100%

B-Branch /Decision Coverage

Let us take one example to explain decision coverage:

READ X

READ Y

IF "X > Y"

PRINT X is greater than Y

ENDIF

To get 100% statement coverage only one test case is sufficient for this pseudo-code.

TEST CASE 1: X=10 Y=5

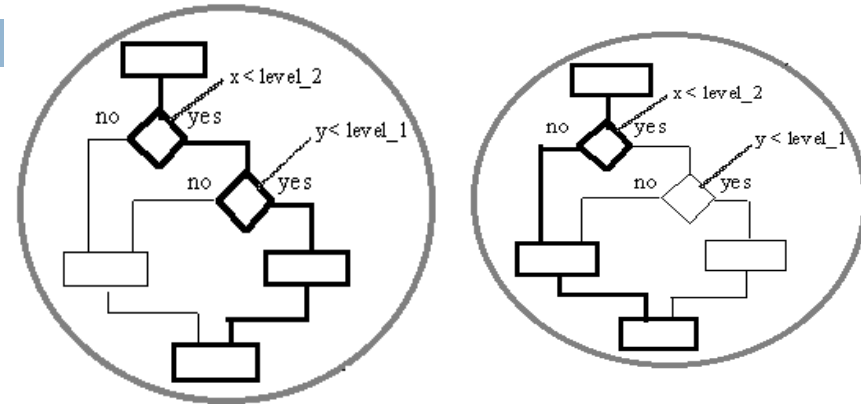
However this test case won't give you 100% decision coverage as the FALSE condition of the IF statement is not exercised.

In order to achieve 100% decision coverage we need to exercise the FALSE condition of the IF statement which will be covered when X is less than Y.

So the final TEST SET for 100% decision coverage will be:

TEST CASE 1: X=10, Y=5

TEST CASE 2: X=2, Y=10



C-Condition/Predicate Coverage

Condition coverage is seen for Boolean expression, condition coverage ensures whether all the Boolean expressions have been evaluated to both TRUE and FALSE.

Let us take an example to explain Condition Coverage

IF ("X && Y")

In order to suffice valid condition coverage for this pseudo-code following tests will be sufficient.

TEST 1: X=TRUE, Y=FALSE

TEST 2: X=FALSE, Y=TRUE

Note: 100% condition coverage does not guarantee 100% decision coverage.

D-Path Coverage

- In this the test case is executed in such a way that every path is executed at least once.

E-Function call Coverage-

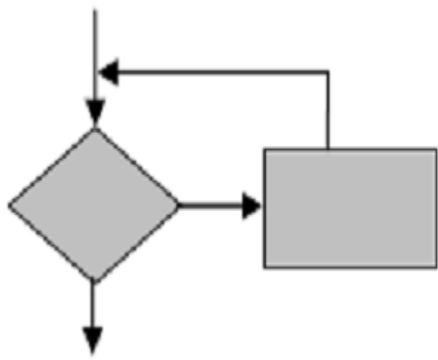
Function call coverage measures how many of the available function calls have actually been made. For example, a code section might make many function calls. To reach 100% Function call coverage, every function call in the code section must be executed at least one time.

Function Coverage = (total no. of functions exercised/ total no. of functions in a program)*100

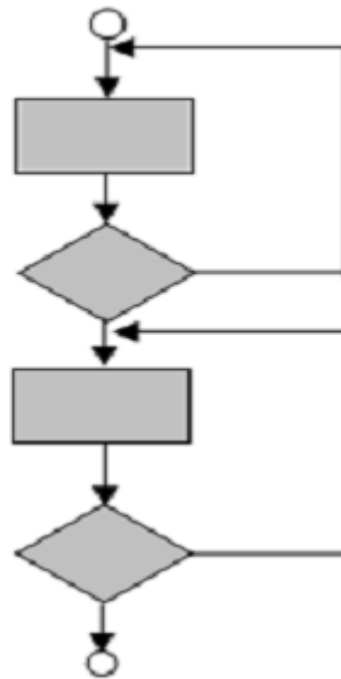
2.2-Loop Testing

Loop testing is another type of white box testing which exclusively focuses on the validity of loop construct. Loops are simple to test unless dependencies exist between the loops or among the loop and the code it contain. There are four classes of loops:

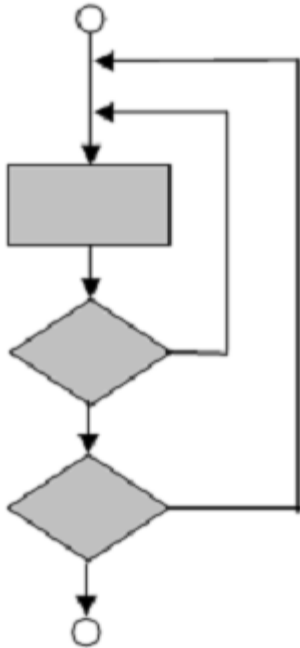
- a) Simple Loop
- b) Nested Loop
- c) Concatenated Loop
- d) Unstructured Loop



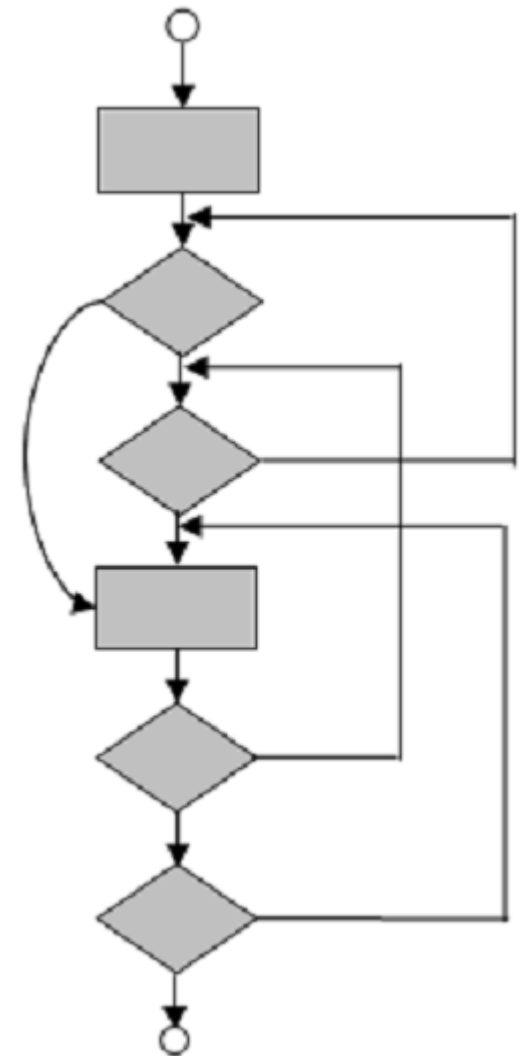
Represent Simple Loop



Represent Concatenated Loop



Represent Nested Loop

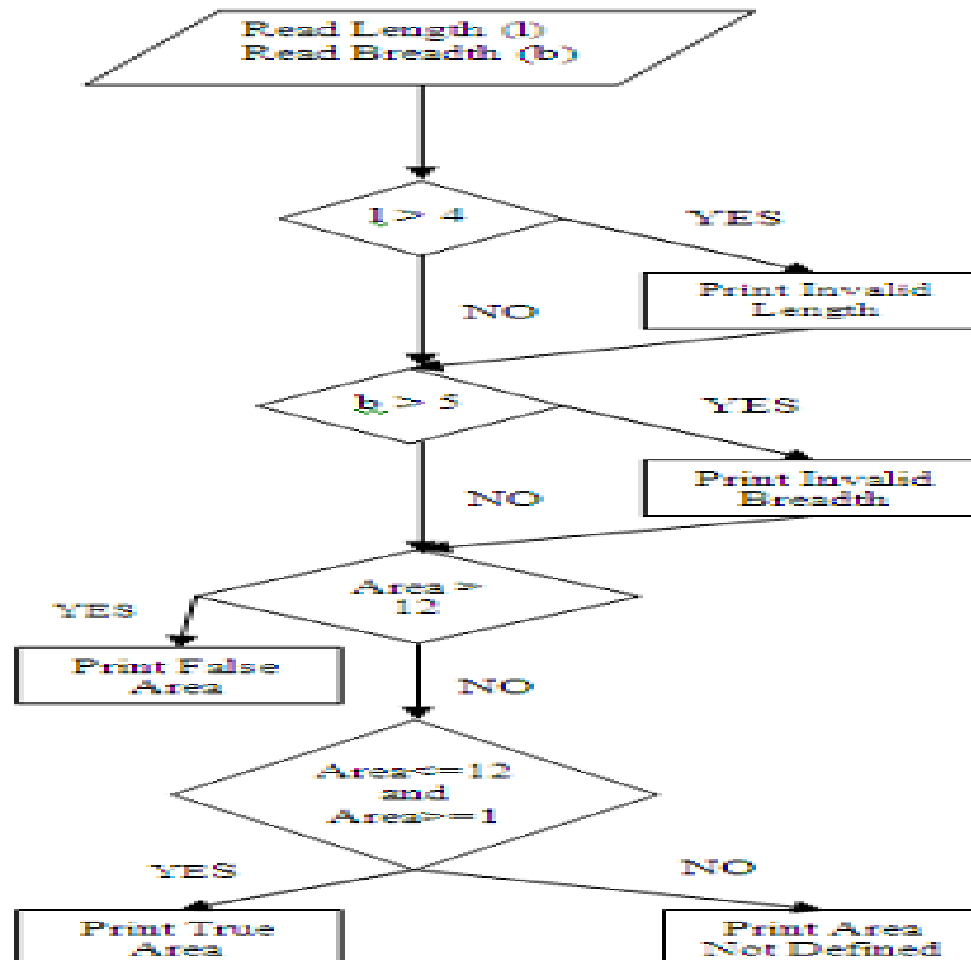


Represent Unstructured Loop

2.3-Branch Testing

- The other synonym of branch testing are conditional testing or decision testing and comes under white box testing technique; it makes sure that each possible outcome from the condition is tested at least once.
- Branch testing however, has the objective to test every option (either true or false) on every control statement which also includes compound decision (when the second decision depends upon the previous decision).
- In branch testing, test cases are designed to exercise control flow branches or decision points in a unit.
- All branches within the branch are tested at least once.

2.3-Branch Testing



2.3-Branch Testing

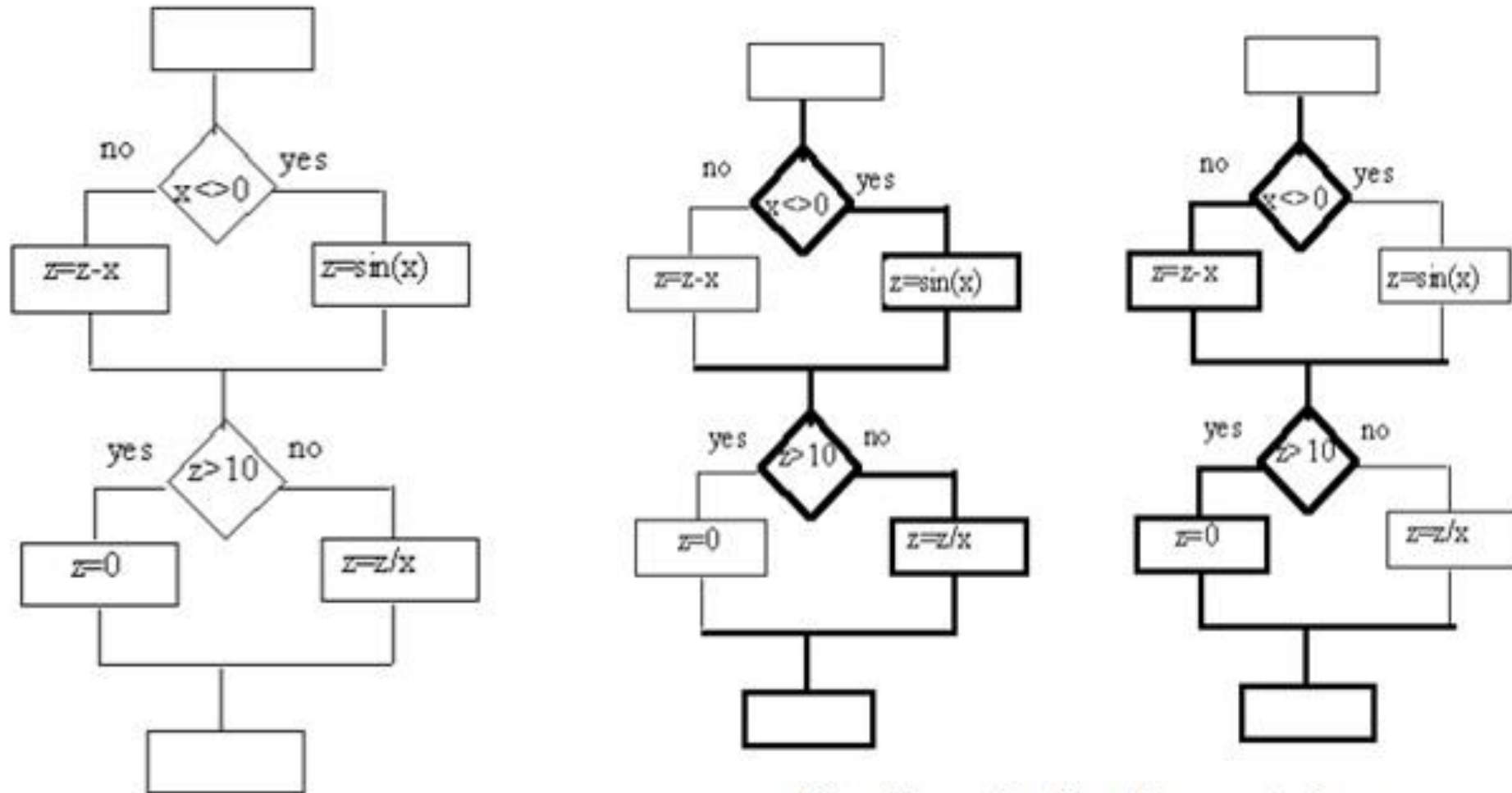
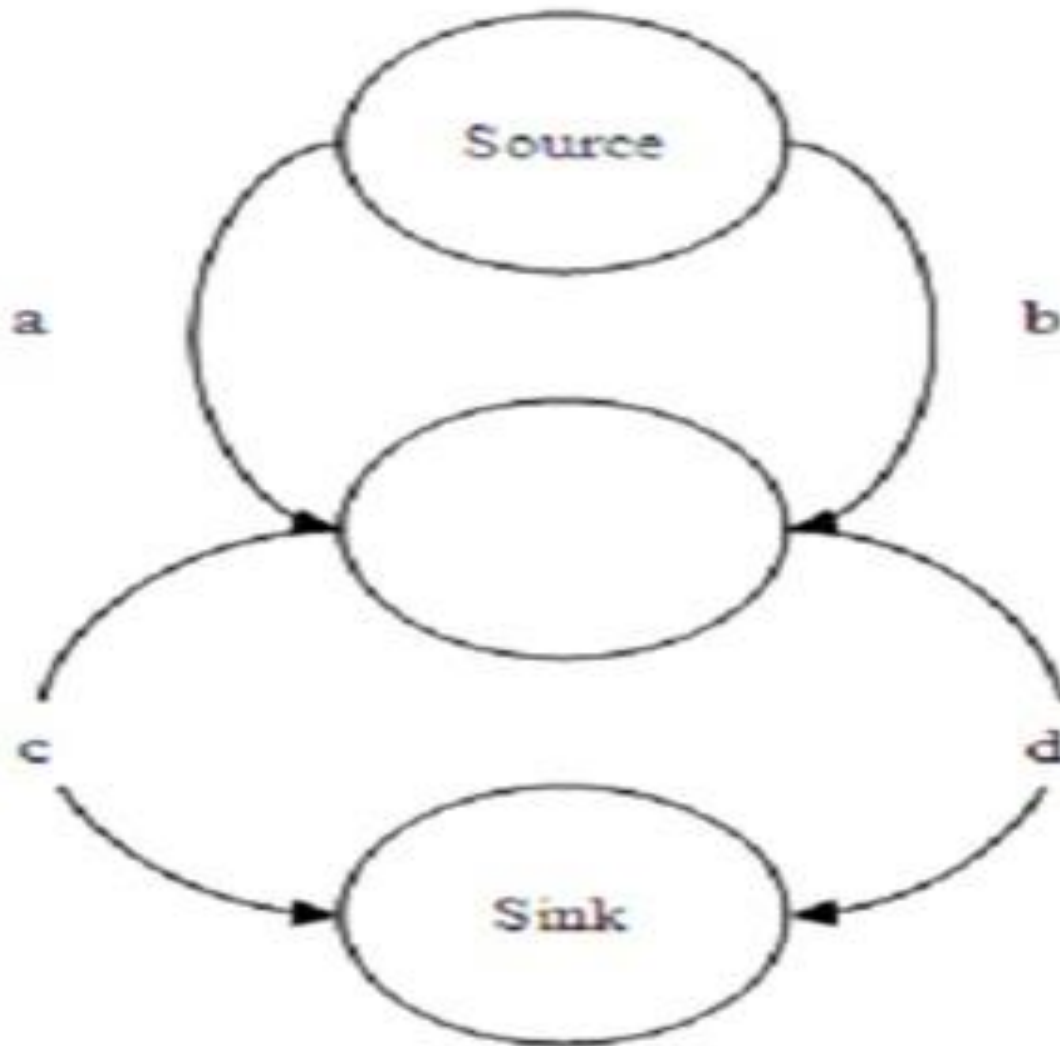


Fig. Enumeration of branch coverage testing

2.3-Data Flow Testing

- Data flow testing is another type of white box testing which looks at how data moves within a program.
- In data flow testing the control flow graph is annotated with the information about how the program variables are defined and used.
- Control flow graph is a diagrammatic representation of a program and its execution.
- The figure below represents Simplified Control Flow Graph.

2.3-Data Flow Testing



2.4-Basis Path Testing

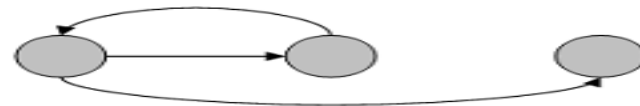
- Basis path is a white box testing techniques which is first proposed by Tom McCabe.
- It allows the test case designer to produce a logical complexity measure of procedural design and use this measure as an approach for outlining a basic set of execution path (basic set is the set of all the execution of a procedure).
- These are test cases that exercise basic set will execute every statement at least once.
- Basic path testing makes sure that each independent path through the code is taken in a predetermined order.

2.4-Basis Path Testing

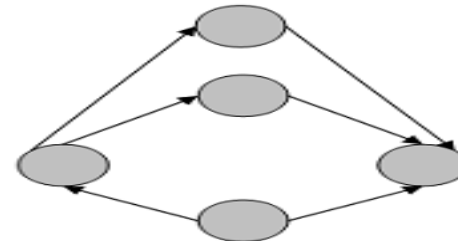
□ Flow Graph Notation-



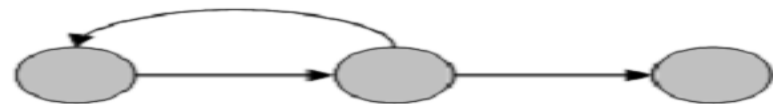
Sequence



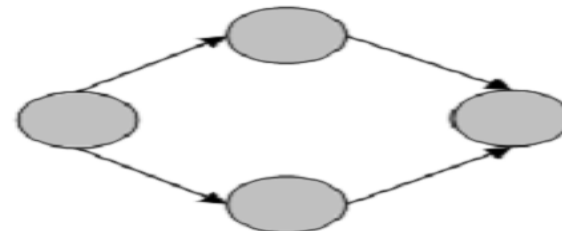
While



Case



Until



If

2.4-Basis Path Testing

□ Cyclomatic Complexity-

A cyclomatic complexity is a software metric which gives a quantitative measure of logical complexity. When used in the context of the basis path testing method, it defines the number of independent paths (any path through the program that introduces atleast one new set of processing statement or a new condition) in the basis set of program and provides upper bound for number of tests required to guarantee coverage of all program statements.

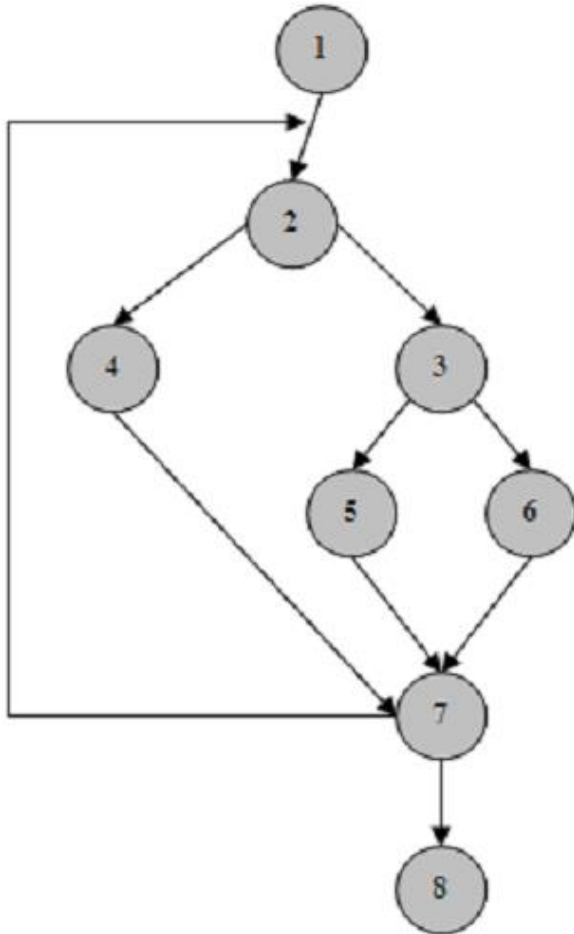
We can compute cyclomatic complexity for a flow graph in any of the given ways.

1. Cyclomatic complexity, $V(G)$, for a graph flow G is defined as
$$V(G) = P + 1$$

Where P is the number of predicate nodes
2. The number of regions of the flow graph corresponds to the cyclomatic complexity.
3. Cyclomatic Complexity, $V(G)$, for a flow graph G is defined as
$$V(G) = E - N + 2$$

Where N is the number of nodes in a flow graph and E is the number of edges.

2.4-Basis Path Testing(Cyclomatic Complexity)



As set of independent paths for flow graph illustrated in Figure

Path 1: 1-2-4-7-8

Path 2: 1-2-3-5-7-8

Path 3: 1-2-3-6-7-8

Path 4: 1-2-4-7-2-4.....-7-8

To measure cyclomatic complexity

Region, $R = 4$

Number of Nodes = 8

Number of edges = 10

Number of Predicate Nodes = 3

Cyclomatic Complexity

$V(G) = R = 4$

Or

$V(G) = \text{Predicate Nodes} + 1$

$= 3 + 1 = 4$

Or

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

$= 10 - 8 + 2$

$= 4$

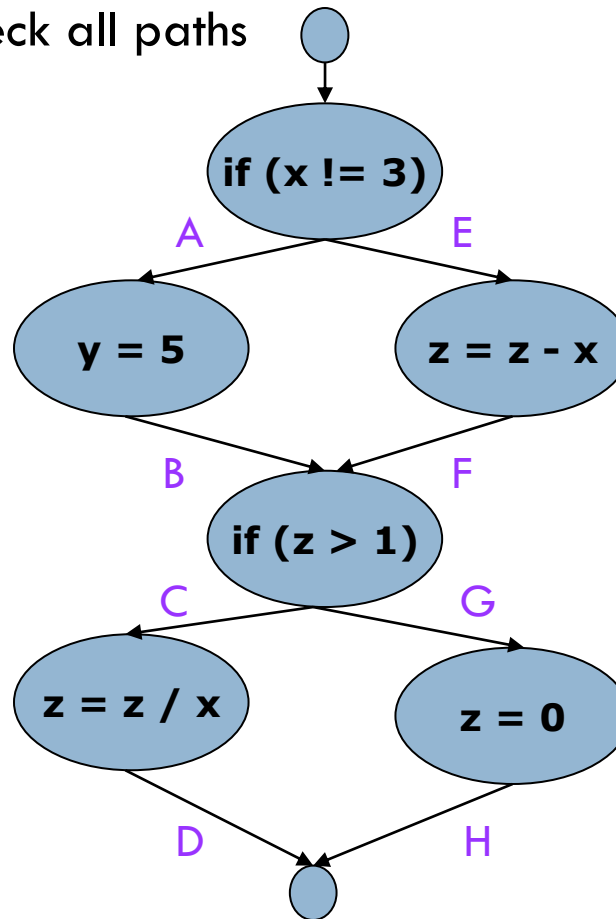
2.4-Path Testing

■ Paths: different routes that your program can take

□ Design test data to check all paths

□ Example

```
if (x != 3) {  
    y = 5;  
}  
else {  
    z = z - x;  
}  
  
if (z > 1) {  
    z = z / x;  
}  
else {  
    z = 0;  
}
```



<x=0, z=1>

Paths A, B, G, H.

<x=3, z=3>

Paths E, F, C, D.

2.4-Basis Path Testing(Cyclomatic Complexity) (Total no of Independent Path)

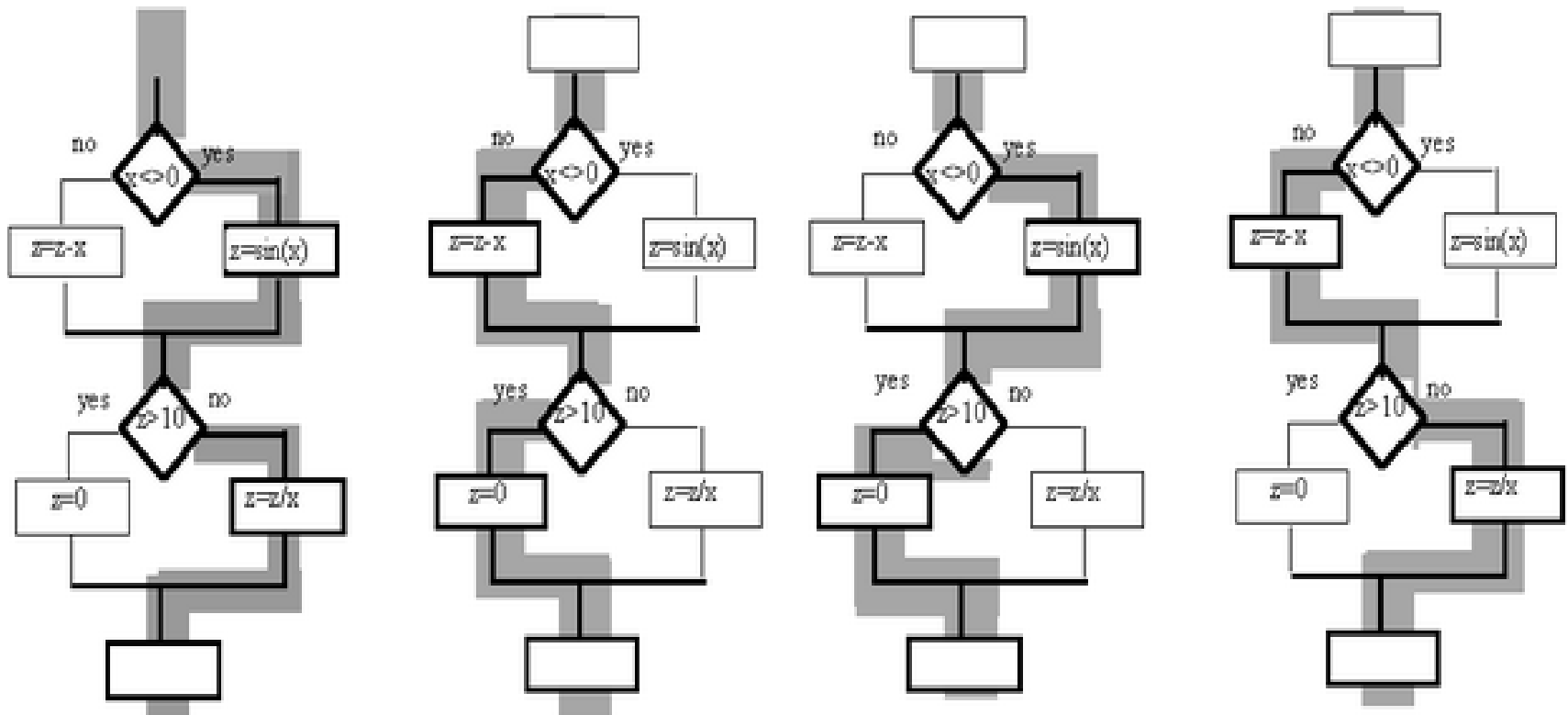


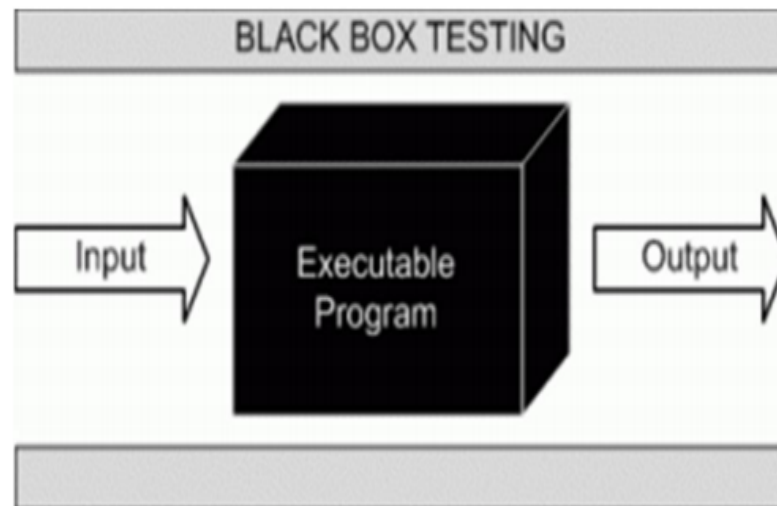
Fig. Enumeration of path coverage testing

Black-Box Testing[Functional Testing]

Black box testing is "**the verification of a software system based solely on requirements with no reference to its internal workings.**" In other words, focus on what you can put in to the box and what comes out of the box and not what happens inside of the box.

Or

Also known as Behavioral Testing is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.



This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see.

Black-Box Testing

Types of Errors, we can find out through Black Box Testing:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

Advantages

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications
- Tester need not know programming languages or how the software has been implemented
- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias
- Test cases can be designed as soon as the specifications are complete

Black-Box Testing-Approach

1-Boundary Value Analysis

Boundary Value Analysis:-

This is simple but popular functional testing technique. In this we concentrate on input values and design test cases with input values that are on or close to boundary values.

Experience has shown that such test cases have a higher probability of detecting a fault in the software.

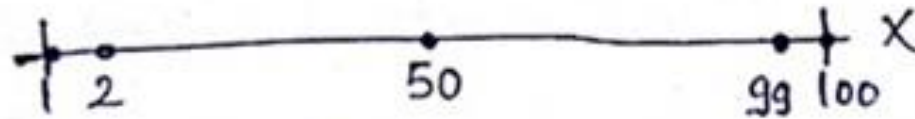
1-Boundary Value Analysis

Suppose there is a program 'square' which takes 'x' as input and prints the square of 'x' as output. The range of x is from 1 to 100. So, In boundary value analysis, we select values on or close to boundaries and all input values may have one of the following;

- (i) Minimum Value
- (ii) Just above minimum value
- (iii) Maximum Value
- (iv) Just below maximum value
- (v) Nominal (Average) Value

1-Boundary Value Analysis

These values may be shown as, for the program 'square'.



These five values (1, 2, 50, 99, 100) are selected on the basis of boundary value analysis, so there is no need to select all 100 inputs and execute the program one by one for all 100 inputs.

The number of inputs selected by this technique is $4n+1$ Where n is the no. of inputs.

1-Boundary Value Analysis

Test cases for 'square' program are given in Table

Test Case	Input x	Expected O/P
1.	1	1
2.	2	4
3.	50	2500
4.	99	9801
5.	100	10000

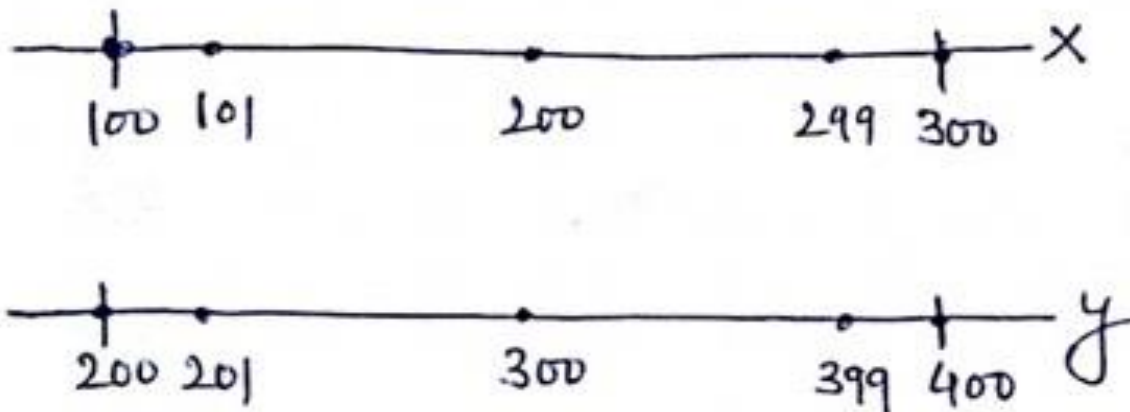
1-Boundary Value Analysis[Example-2]

Consider a program 'Addition' with two input values x & y and it gives the addition of x and y as an output. The range of both input values are given as:

$$100 \leq x \leq 300$$

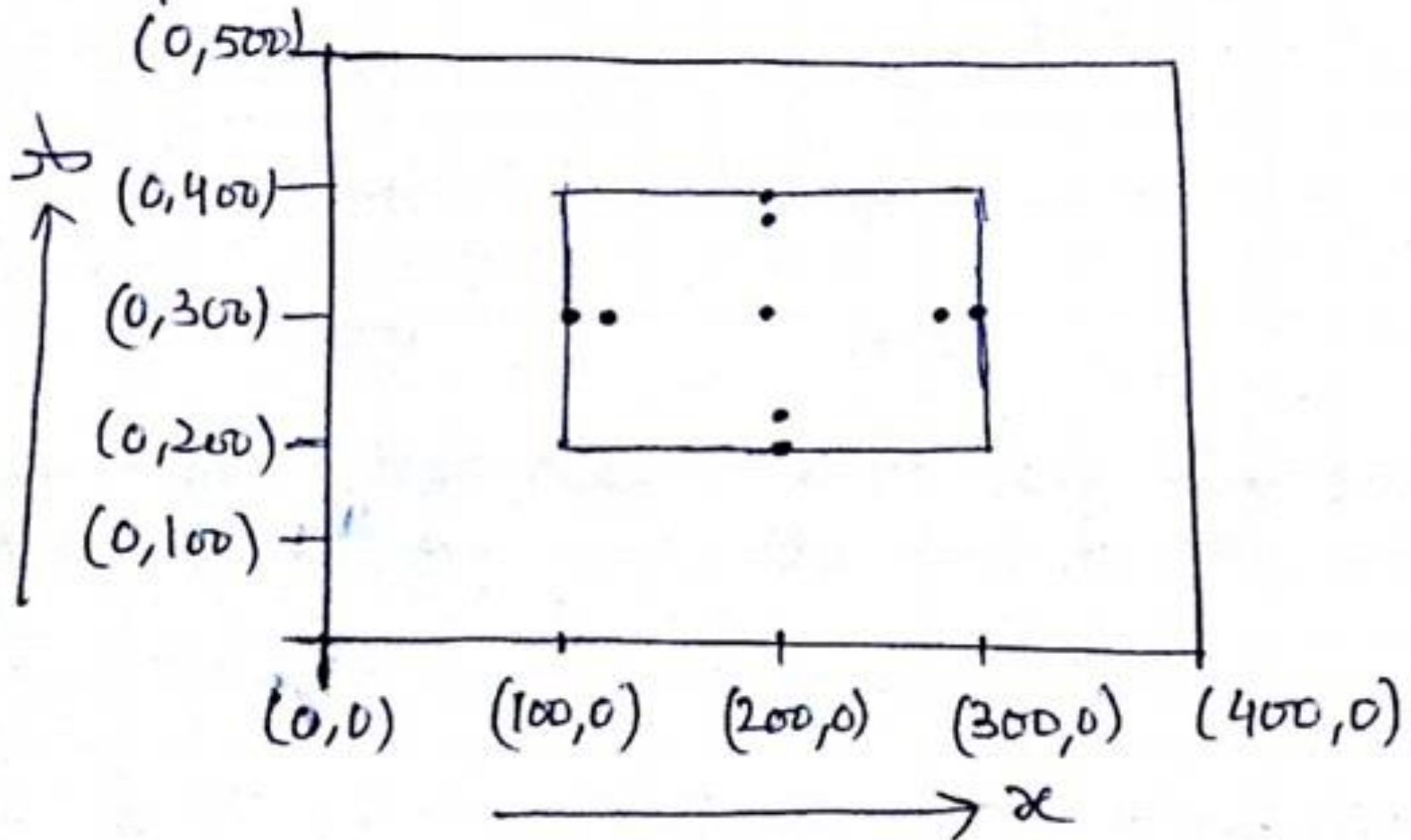
$$200 \leq y \leq 400$$

The Selected values for x and y are given in figure:



1-Boundary Value Analysis[Example-2]

The input domain is shown as:



1-Boundary Value Analysis[Example-2]

So, test cases for program 'Addition'

$$\underline{4n+1 = 4 \times 2 + 1 = 9}$$

Test Case	x	y	Expected o/p
1	100	300	400
2	101	300	401
3	200	300	500
4	299	300	599
5	300	300	600
6	200	200	500
7	200	201	401
8	200	300	500
9	200	399	599
10	200	400	600

In above table, test case 3 & 8 are similar, hence we select only one. So, the no. of test cases = 9

2-Equivalence Class Testing

Equivalence Class Testing

We know that a large number of test cases are generated for every program. It is neither feasible nor desirable to execute all such test cases, because some test cases do the same thing.

So, we may divide input domain into various categories with some relationship and expects that every test case from a category exhibits the same behaviour. If categories are well selected, we may assume that if one representative test case works correctly, others may also give the same results. This assumption allows us to select exactly one test case from each category. and if there are four categories, four test cases may be selected. Each category is called an equivalence class and this type of testing is known as equivalence class testing.

2-Equivalence Class Testing

Creation of Equivalence Classes:-

The entire input domain can be divided into at least two equivalence classes:-

- (i) Containing all valid inputs.
- (ii) Containing all invalid inputs.

Each class can further be subdivided into equivalence class on which the program is required to behave differently.

Eg. 1:- Consider the program 'square' which takes 'x' as input (range 1-100) and prints the square of 'x'.

2-Equivalence Class Testing

The input domain equivalence classes for program are; -

- (i) $I_1 = \{ 1 \leq x \leq 100 \}$ (Valid input range from 1 to 100)
- (ii) $I_2 = \{ x < 1 \}$ (Any invalid input where x is less than 1)
- (iii) $I_3 = \{ x > 100 \}$ (Any invalid input where x is greater than 100)

So, Test Cases for program 'square' based on input domain

<u>Test Case</u>	<u>Input x</u>	<u>Expected O/P</u>
I_1	50	2500
I_2	0	Invalid O/P
I_3	101	Invalid O/P.

3-Decision Table Based testing

Decision Tables Based Testing

Decision tables are precise and compact way to model complicated logic. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions.

It is another popular black box testing. A decision table has following four portions

- Stub portion
- Entry portion
- Condition portion, and
- Action portion

Structure of decision table: Please refer following decision table

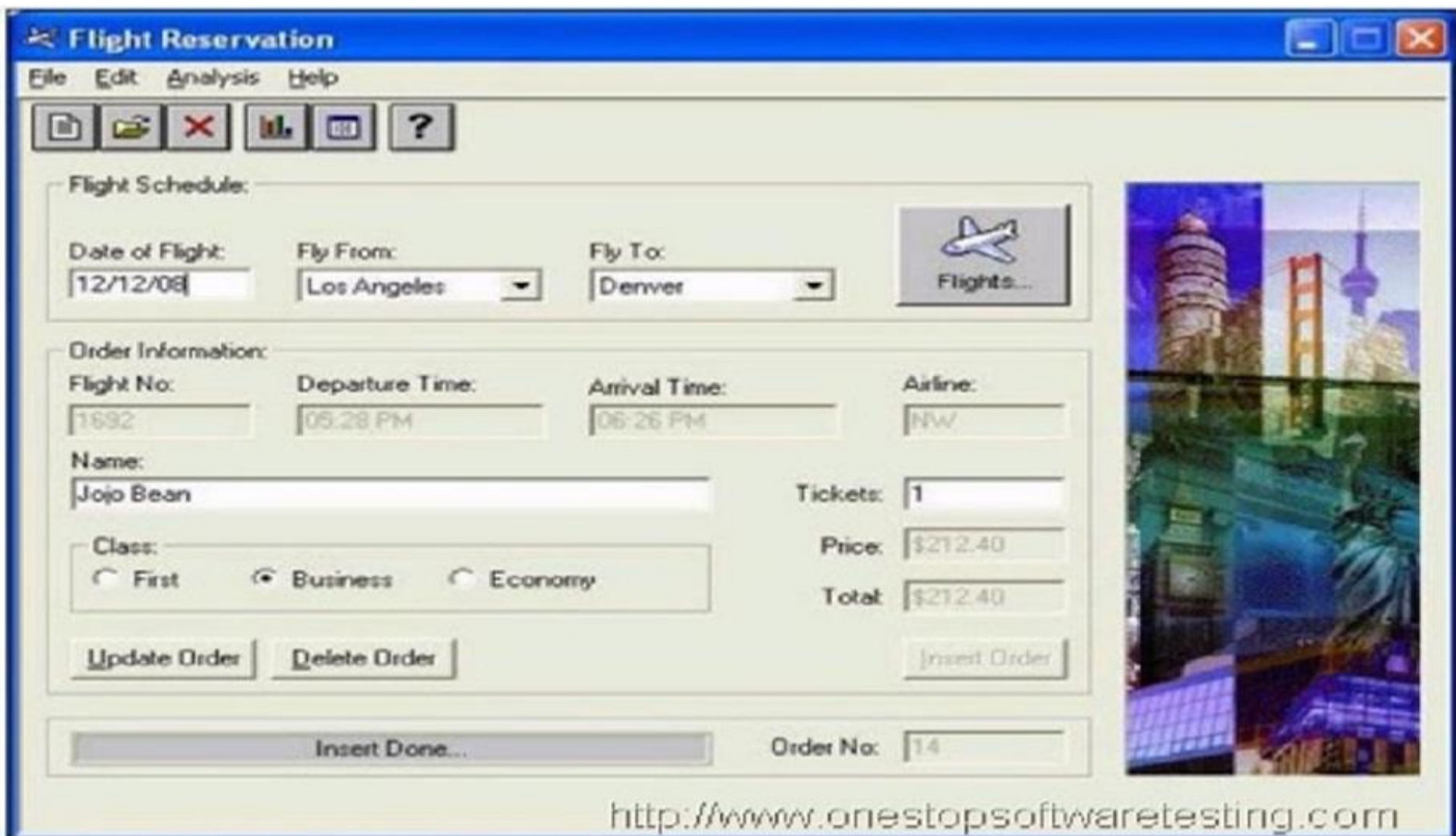
Rules →

Condition Stub	Condition Entry
Action Stub	Action Entries

Structure of Decision Tables

Decision Table Testing is a good way to deal with combination of inputs, which produce different results.
To understand this with an example let's consider the behavior of Flight Button for different combinations of Fly from & Fly To

3-Decision Table Based Testing



The screenshot shows a Windows-style application window titled "Flight Reservation". It has a menu bar with "File", "Edit", "Analysis", and "Help". Below the menu bar is a toolbar with icons for file operations, a help icon, and a question mark. The main area is divided into two sections: "Flight Schedule:" and "Order Information:". The "Flight Schedule:" section includes fields for "Date of Flight:" (12/12/08), "Fly From:" (Los Angeles), and "Fly To:" (Denver), along with a "Flights..." button featuring an airplane icon. The "Order Information:" section includes fields for "Flight No.:" (1692), "Departure Time:" (05:28 PM), "Arrival Time:" (06:26 PM), and "Airline:" (NW). It also has a "Name:" field (Jojo Bean), a "Class:" section with radio buttons for "First", "Business" (selected), and "Economy", and a "Tickets:" field (1). The "Price:" and "Total:" fields both show \$212.40. At the bottom of the "Order Information:" section are buttons for "Update Order", "Delete Order", and "Insert Order". A status bar at the very bottom shows "Insert Done..." and "Order No.:" (14). On the right side of the window, there is a vertical strip of images showing various cityscapes and landmarks, including the Golden Gate Bridge and the CN Tower.

Flight Reservation

File Edit Analysis Help

Flight Schedule:

Date of Flight: 12/12/08 Fly From: Los Angeles Fly To: Denver Flights...

Order Information:

Flight No: 1692 Departure Time: 05:28 PM Arrival Time: 06:26 PM Airline: NW

Name: Jojo Bean

Class: ☐ First ☒ Business ☐ Economy

Tickets: 1

Price: \$212.40

Total: \$212.40

Update Order Delete Order Insert Order

Insert Done... Order No: 14

<http://www.onestopsoftwaretesting.com>

3-Decision Table Based Testing

- a. When both Fly From & Fly To are not set the Flight Icon is disabled. In the decision table , we register values False for Fly From & Fly To and the outcome would be ,which is Flights Button will be disabled i.e. FALSE
- b. Next, when Fly From is set but Fly to is not set, Flight button is disabled. Correspondingly you register True for Fly from in the decision table and rest of the entries are false
- c. When , Fly from is not set but Fly to is set , Flight button is disabled And you make entries in the decision table
- d. Lastly, only when Fly to and Fly from are set, Flights button is enabled And you make corresponding entry in the decision table.

Decision Table for above diagram:

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM	F	T	F	T
FLY TO	F	F	T	T
<u>OUTCOME</u> FLIGHTS BUTTON	F	F	F	T

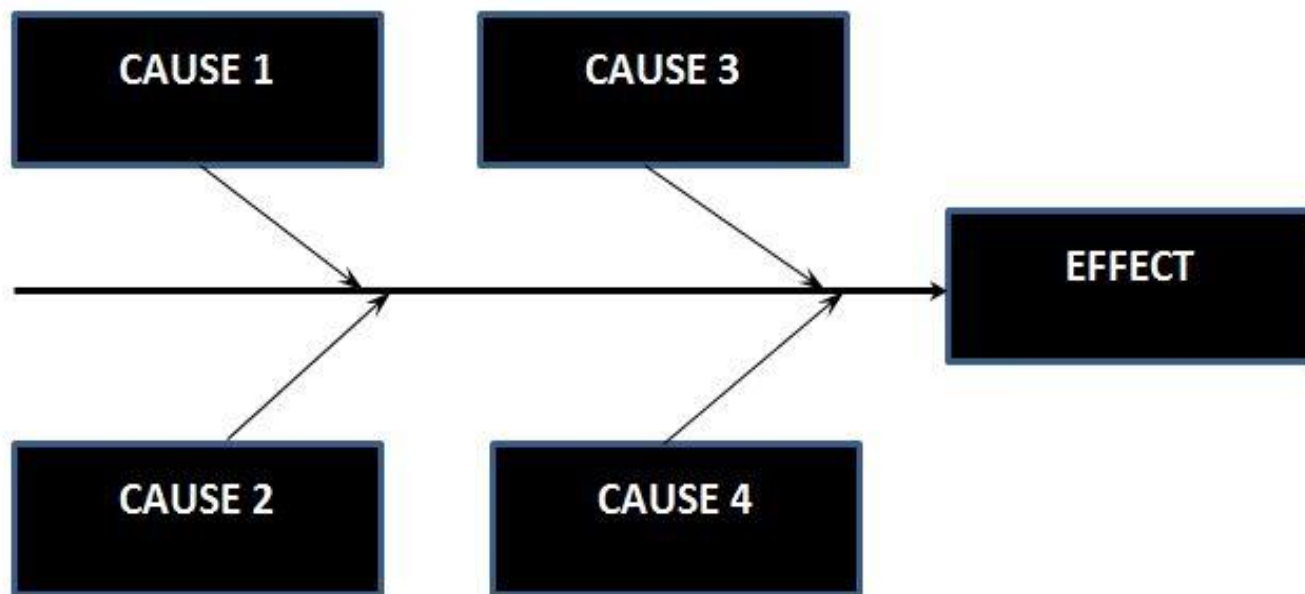
3-Decision Table Based Testing

- e. If you observe the outcomes for Rule 1, 2 & 3 remain the same. So you can select any of them and rule 4 for your testing.
- f. The significance of this technique becomes immediately clear as the number of inputs increases. Number of possible Combinations is given by 2^n , where n is number of Inputs.
- g. For $n = 10$, which is very common in web based testing, having big input forms, the number of combinations will be 1024. Obviously, you cannot test all but you will choose a rich sub-set of the possible combinations using decision based testing technique

4-Causes & Effect Graph

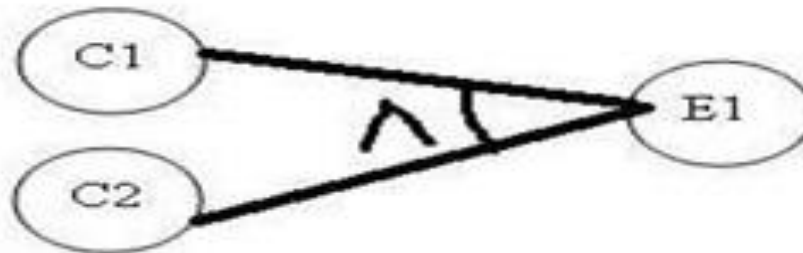
(Causes --> intermediate nodes --> Effects)

Cause Effect Graph is a black box testing technique that graphically illustrates the relationship between a given outcome and all the factors that influence the outcome.

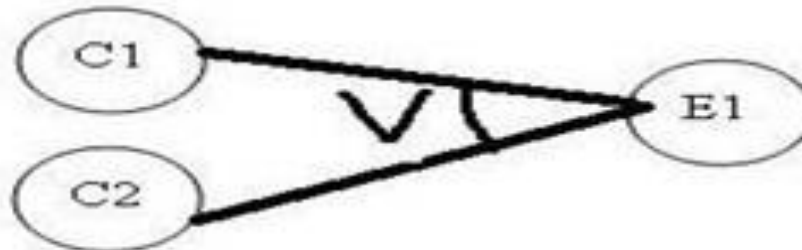


4-Causes & Effect Graph

AND – For effect E1 to be true, both the causes C1 and C2 should be true



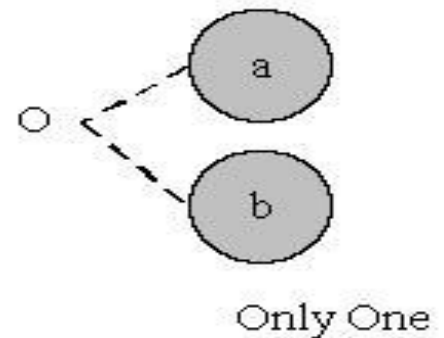
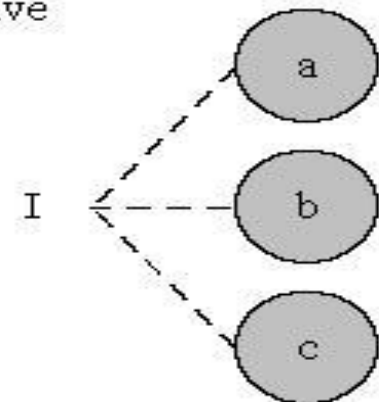
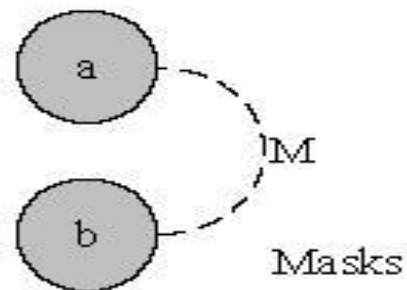
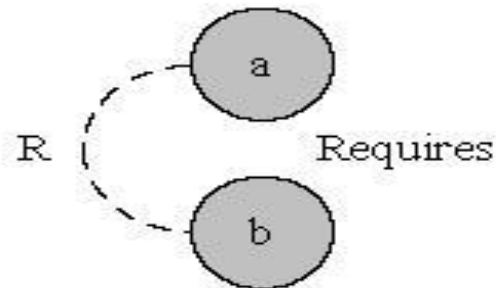
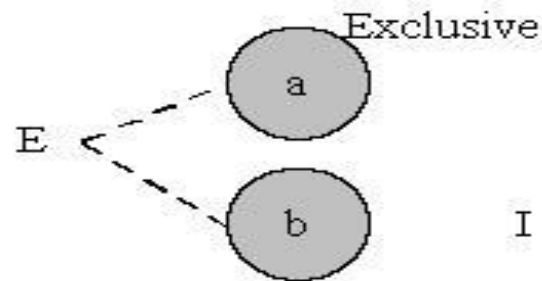
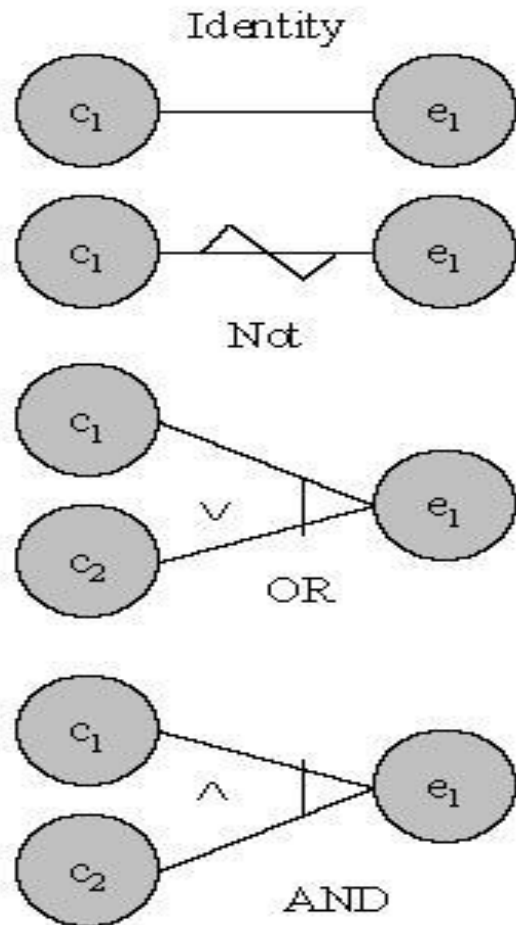
OR – For effect E1 to be true, either of causes C1 OR C2 should be true



NOT – For Effect E1 to be True, Cause C1 should be false



4-Causes & Effect Graph



4-Causes & Effect Graph

- In software testing, a **Cause–Effect Graph** is a directed graph that maps a set of causes to a set of effects.
- The causes may be thought of as the input to the program, and the effects may be thought of as the output.
- Usually the graph shows the nodes representing the causes on the left side and the nodes representing the effects on the right side.
- There may be intermediate nodes in between that combine inputs using logical operators such as AND and OR.
- Constraints may be added to the causes and effects.

4-Causes & Effect Graph

- These are Represented as edges labeled with the constraint symbol using a dashed line-
 - ▣ For causes, valid constraint symbols are **E (exclusive)**, **O (one and only one)**, **I (at least one)**, and **R (Requires)** and **M (Mask)**.
 - ▣ The **exclusive constraint E** states that **at-most one** of the causes 1 and 2 can be true, i.e. both cannot be true simultaneously.
 - ▣ The **Inclusive (at least one) I** constraint states that **at least one** of the causes 1, 2 or 3 must be true, i.e. all cannot be false simultaneously.
 - ▣ The **one and only one (O)** constraint states that **only one** of the causes 1, 2 or 3 can be true. The **Requires constraint R** states that **if cause 1 is true, then cause 2 must be true**, and it is impossible for 1 to be true and 2 to be false.
 - ▣ For effects, valid constraint symbol is **M (Mask)**. The **mask constraint** states that **if effect 1 is true then effect 2 is false**. Note that the mask constraint relates to the effects and not the causes like the other constraints.

Let's draw a cause and effect graph based on a Situation-

The "Print message" is software that read two characters and, depending of their values, messages must be printed.

- The first character must be an "A" or a "B".
- The second character must be a digit.
- If the first character is an "A" or "B" and the second character is a digit, the file must be updated.
- If the first character is incorrect (not an "A" or "B"), the message X must be printed.
- If the second character is incorrect (not a digit), the message Y must be printed.

Solution:

The causes for this situation are:

C1 – First character is A

C2 – First character is B

C3 – Second character is a digit

The effects (results) for this situation are

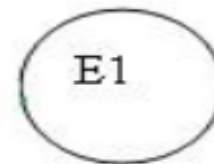
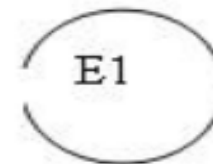
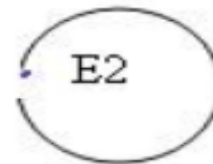
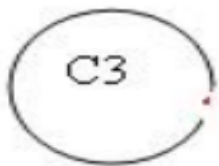
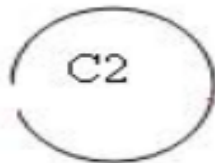
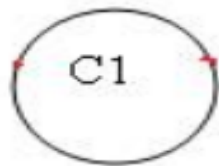
E1 – Update the file

E2 – Print message "X"

E3 – Print message "Y"

Solution-

First draw the causes and effects as shown below:



Key – Always go from effect to cause (left to right). That means, to get effect "E", what causes should be true.

Solution-

In this example, let's start with Effect E1.

Effect E1 is to update the file. The file is updated when

- First character is "A" and second character is a digit
- First character is "B" and second character is a digit
- First character can either be "A" or "B" and cannot be both.

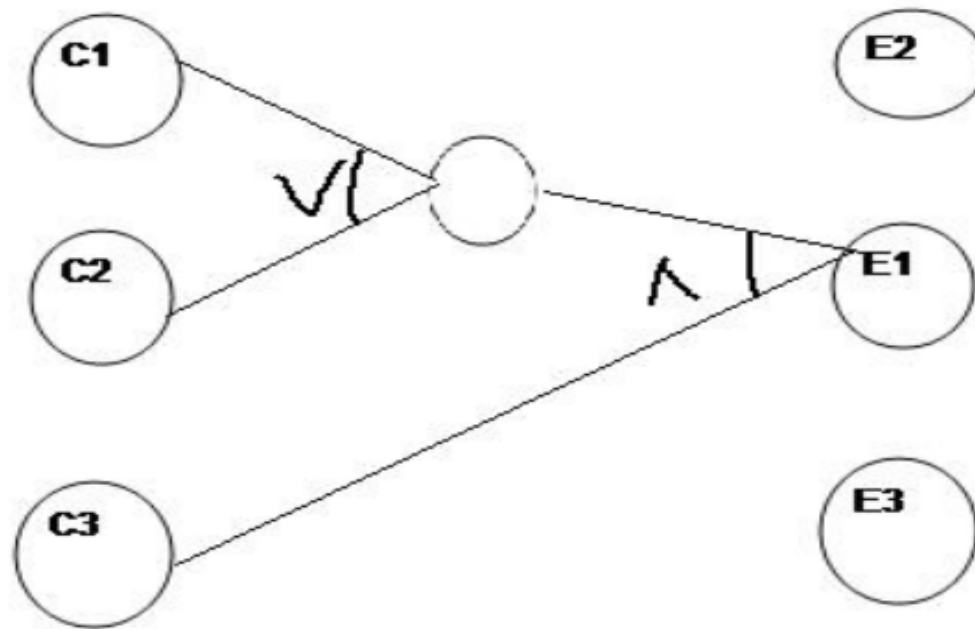
Now let's put these 3 points in symbolic form:

For E1 to be true – following are the causes:

- C1 and C3 should be true
- C2 and C3 should be true
- C1 and C2 cannot be true together. This means C1 and C2 are mutually exclusive.

Solution-

Now let's draw this:

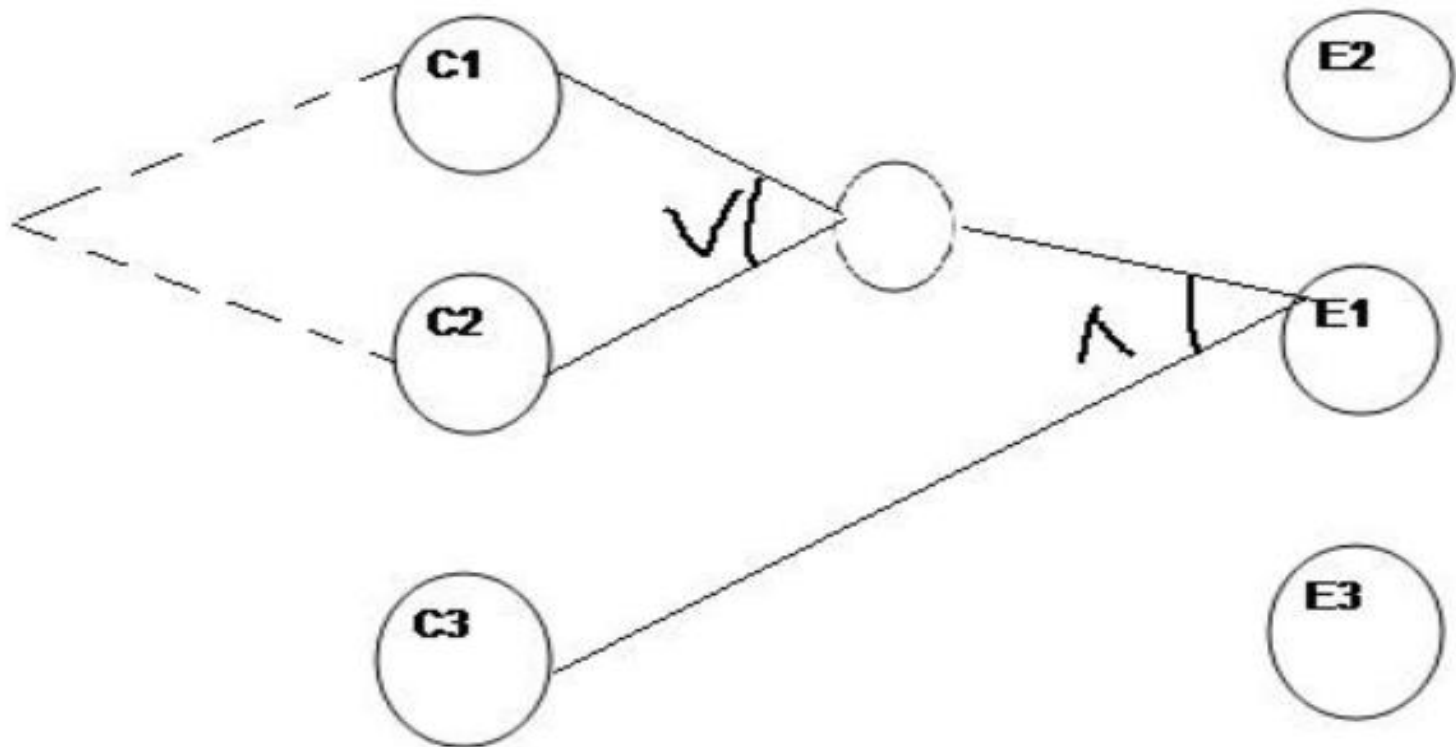


So as per the above diagram, for E1 to be true the condition is $(C1 \vee C2) \wedge C3$

The circle in the middle is just an interpretation of the middle point to make the graph less messy.

Solution-

There is a third condition where C1 and C2 are mutually exclusive. So the final graph for effect E1 to be true is shown below:

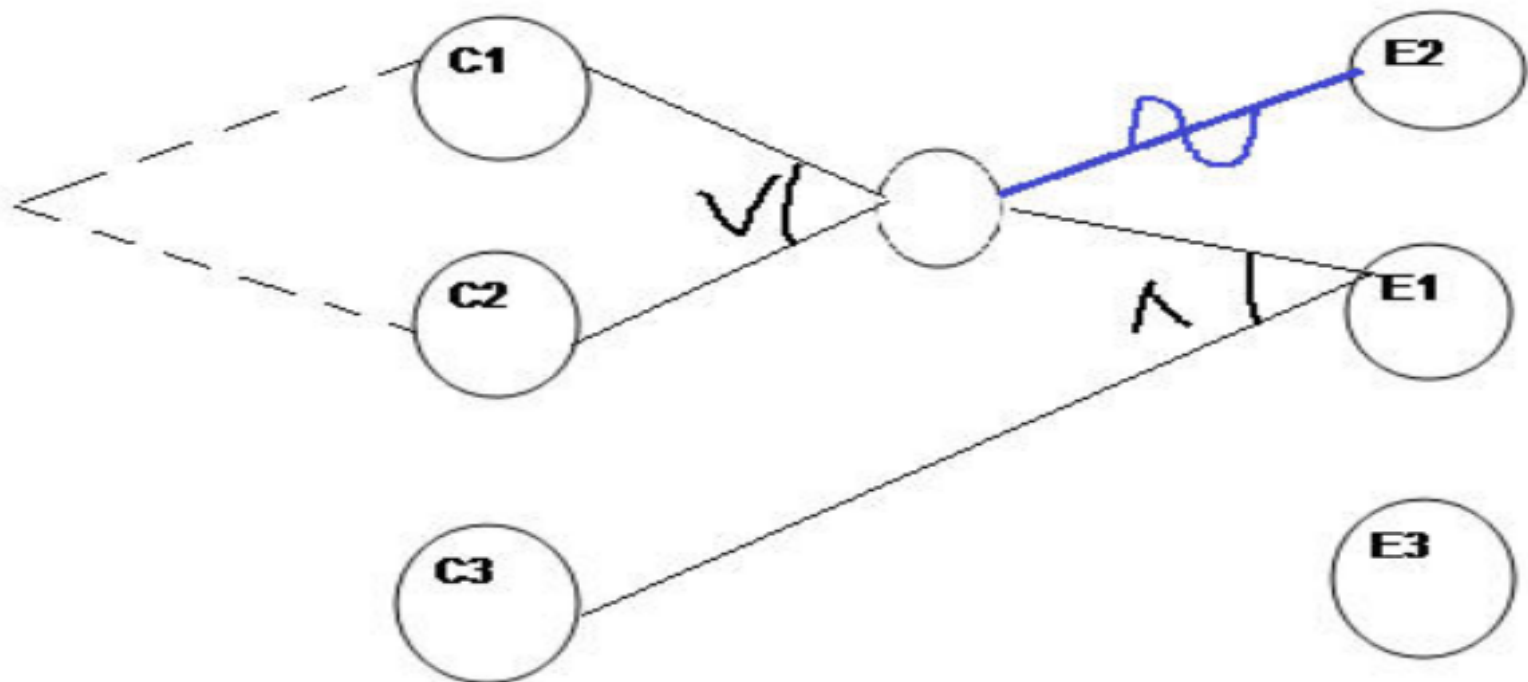


Solution-

Lets move to Effect E2:

E2 states to print message "X". Message X will be printed when First character is neither A nor B.

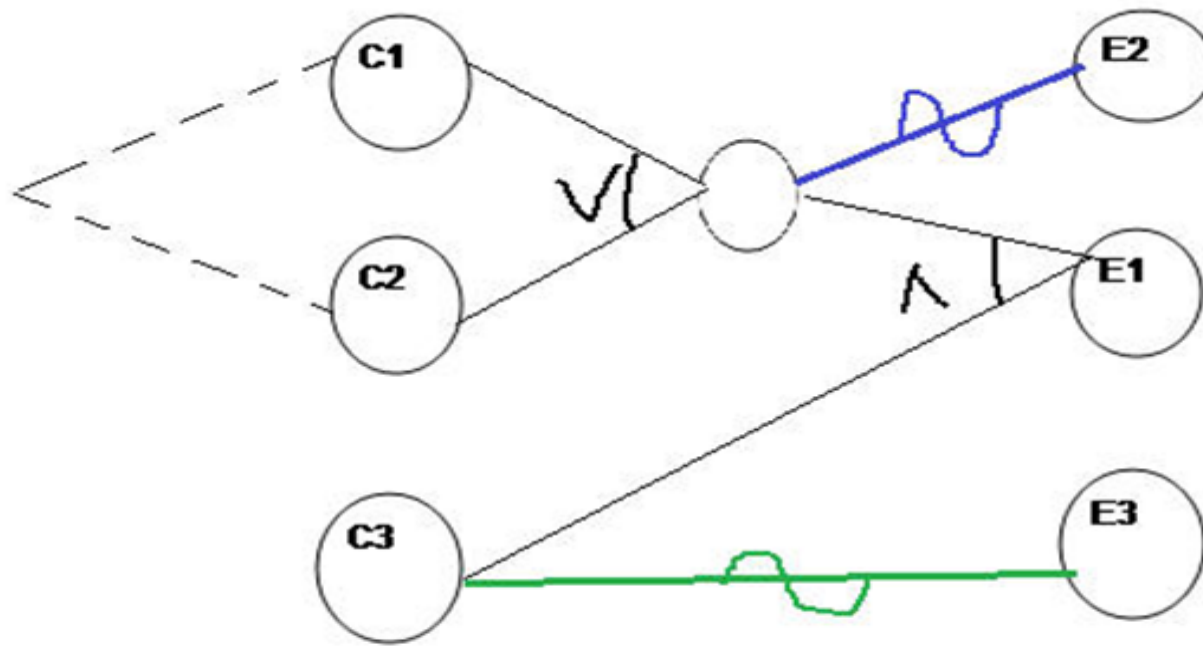
Which means Effect E2 will hold true when either C1 OR C2 is invalid.
So the graph for Effect E2 is shown as (In blue line)



Solution-

For Effect E3.

E3 states to print message “Y”. Message Y will be printed when Second character is incorrect. Which means Effect E3 will hold true when C3 is invalid. So the graph for Effect E3 is shown as (In Green line)



This completes the Cause and Effect graph for the above situation.

Now let's move to draw the Decision table based on the above graph.

Writing Decision table based on Cause and Effect graph

First write down the Causes and Effects in a single column shown below

Actions
C1
C2
C3
E1
E2
E3

Key is the same. Go from bottom to top which means traverse from effect to cause.

Start with Effect E1. For E1 to be true, the condition is: $(C1 \vee C2) \wedge C3$.

Here we are representing True as **1** and False as **0**

Decision table

First put Effect E1 as True in the next column as

Actions	
C1	
C2	
C3	
E1	1
E2	
E3	

Now for E1 to be "1" (true), we have the below two conditions
C1 AND C3 will be true
C2 AND C3 will be true

Actions		
C1	1	
C2		1
C3	1	1
E1	1	1
E2		
E3		

Decision table

For E2 to be True, either C1 or C2 has to be false shown as

Actions				
C1	1		0	
C2		1		0
C3	1	1	0	1
E1	1	1		
E2			1	1
E3				

For E3 to be true, C3 should be false.

Actions						
C1	1		0		1	
C2		1		0		1
C3	1	1	0	1		
E1	1	1				
E2			1	1		
E3					1	1

Decision table

So it's done. Let's complete the graph by adding 0 in the blank column and including the test case identifier.

Actions	TC1	TC2	TC3	TC4	TC5	TC6
C1	1	0	0	0	1	0
C2	0	1	0	0	0	1
C3	1	1	0	1	0	0
E1	1	1	0	0	0	0
E2	0	0	1	1	0	0
E3	0	0	0	0	1	1

4-Causes & Effect Graph[Example]

Myers explained this effectively with following example. “The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the file update is made. If the character in column 1 is incorrect, message x is issued. If the character in column 2 is not a digit, message y is issued”.

The causes are

c_1 : character in column 1 is A

c_2 : character in column 1 is B

c_3 : character in column 2 is a digit

and the effects are

e_1 : update made

e_2 : message x is issued

e_3 : message y is issued

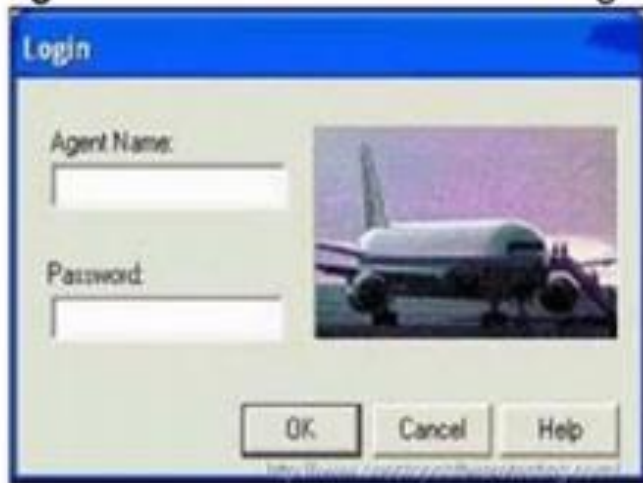
5-State Transition Based Testing

State transition testing is used where some aspect of the system can be described in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the “machine”. This is the model on which the system and the tests are based.

A state transition model has four basic parts:

- The **states** that the software may occupy
- The **transitions** from one state to another
- The **events** that cause a transition
- The **actions** that result from a transition

E.g. 1: Consider the behavior of a login screen shown below:



5-State Transition Based Testing

If we provide valid login & password in Agent Name & Password fields, we get the access screen shown below:

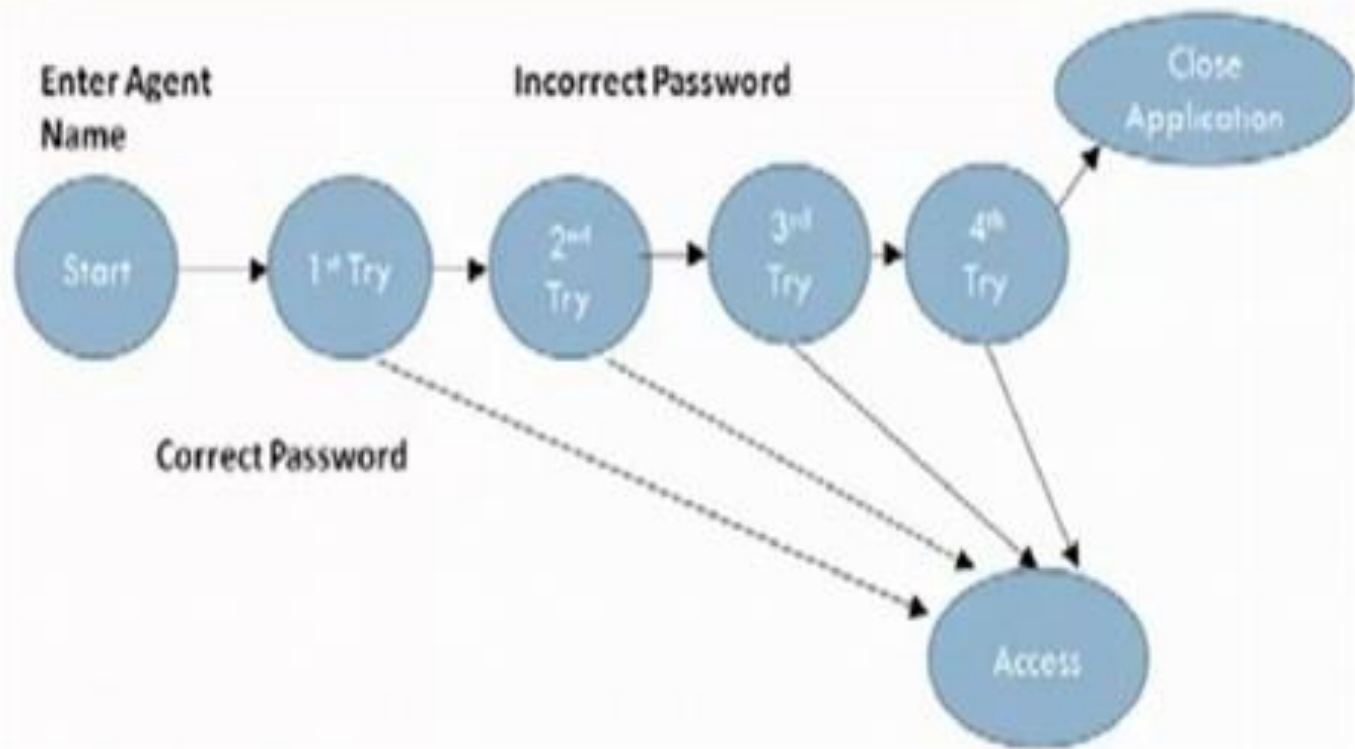


If we provide wrong values, we get the error screen show below:



& if we provide wrong values 4 consecutive times the application gets closed automatically. Hence, we can show the transition diagram as follows:

5-State Transition Based Testing



In the above transition diagram, two scenarios are more important (than others) to be tested i.e.

1. Getting login at first attempt and,
2. Closing the application after 4th attempt.

State graph is also useful for identifying the invalid transitions; as shown in state table:

Test, Test Case and Test Suite

- **Test and Test case terms are used interchangeably.**
- **In Practice, Both are same and are treated as synonyms.**
- **Test case describes an input description and an expected output description.**
- **The set of test cases is called a test suite. Hence any combination of test cases may generate a test suite.**

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig. : Test case template

TEST PLAN

- A test plan is a document consisting of different test cases designed for different testing objects and different testing attributes.
- The test plan is a matrix of test and test cases listed in order of its execution.
- A test plan states-
 - ▣ The items to be tested.
 - ▣ At what level they will be tested at.
 - ▣ The sequence they are to be tested in.
 - ▣ How the test strategy will be applied to the testing of each item and the test environment.

TEST PLAN

Table shows the matrix of test and test cases within the test.

Project Name Project ID

Project Manager Q.A. Manager

TABLE 7.1 Test Plan

Test									
Test ID		Test	1	2	3	...	N	Planned Date	
ID	Tester	Name						Completed	Successful

Test ID, test name, and test cases are designed well before the development phase and have been designed for those who conduct the tests.

TEST-CASE DESIGN



- A test case is a set of instructions designed to discover a particular type of error or defect in the software system by inducing a failure.
- There are two criteria for the selection of test cases:
 - ▣ Specifying a criterion for evaluating a set of test cases.
 - ▣ Generating a set of test cases that satisfy a given criterion.

TEST-CASE DESIGN

Each test case needs proper documentation, preferably in a fixed format. There are many formats; one format is suggested below:

Test case name	Test Case ID
Purpose of test	Testing object (unit, application, module, etc.)
Test attribute	
Tests focus (function, feature, process, interface, validation, verification, etc.)	
Test type (alpha, beta, unit, integration, system)	
Test process	A set of instructions for conducting the test-initial stating condition-inputs-specifications-output expected
Test results	Expected and actual and comparison, error description, post-process state
Action	Correction, authorization, and feedback through retest
Action to initialize the pre-test status	

Three Main kinds of System Testing

[Alpha, Beta and Acceptance Testing]

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms **alpha** and **beta testing** are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Three Main kinds of System Testing

[Alpha, Beta and Acceptance Testing]

1. **Alpha Testing.** *Alpha testing refers to the system testing carried out by the test team within the development organization.*

The alpha test is conducted at the developer's site by the customer under the project team's guidance. In this test, users test the software on the development platform and point out errors for correction. However, the alpha test, because a few users on the development platform conduct it, has limited ability to expose errors and correct them. Alpha tests are conducted in a controlled environment. It is a simulation of real-life usage. Once the alpha test is complete, the software product is ready for transition to the customer site for implementation and development.

2. **Beta Testing.** *Beta testing is the system testing performed by a selected group of friendly customers.*

If the system is complex, the software is not taken for implementation directly. It is installed and all users are asked to use the software in testing mode; this is not live usage. This is called the beta test. Beta tests are conducted at the customer site in an environment where the software is exposed to a number of users. The developer may or may not be present while the software is in use. So, beta testing is a real-life software experience without actual implementation. In this test, end users record their observations, mistakes, errors, and so on and report them periodically.

Three Main kinds of System Testing

[Alpha, Beta and Acceptance Testing]

In a beta test, the user may suggest a modification, a major change, or a deviation. The development has to examine the proposed change and put it into the change management system for a smooth change from just developed software to a revised, better software. It is standard practice to put all such changes in subsequent version releases.

3. **Acceptance Testing.** *Acceptance testing is the system testing performed by the customer to determine whether to accept or reject the delivery of the system.*

When customer software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end-user rather than the software engineers, an acceptance test can range from an informal 'test drive' to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.