ABES

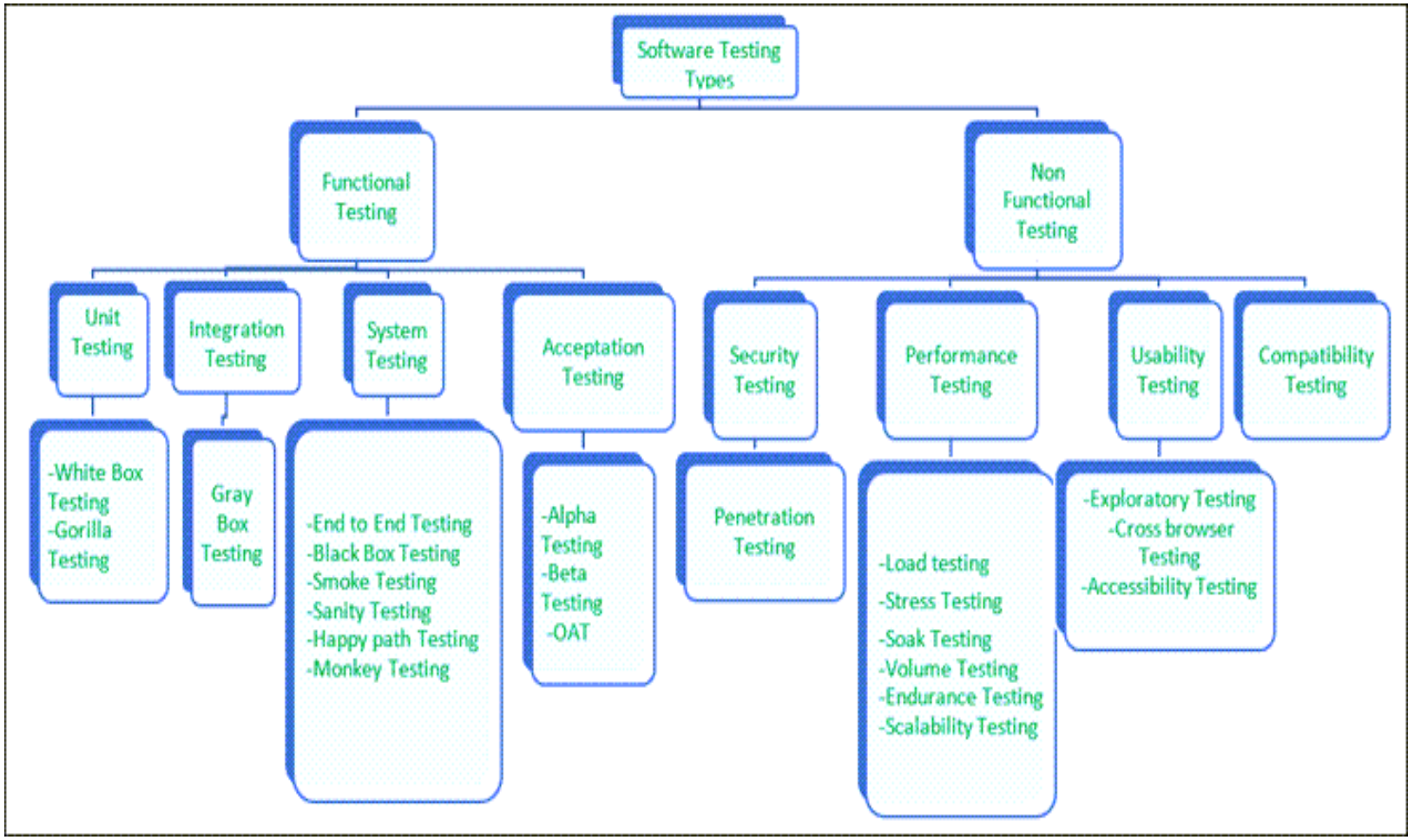
Estd. 2000

# Software Testing


**(Rank-band: 151-200)**

# Topics

- What is Testing
- Need of Testing
- Who should do testing
- Some Terminologies
- Test , Test Cases and Test Suite
- Verification and Validation
- Types of Testing : Alpha and Beta Testing
-


Software Testing Types
Functional Testing
Non Functional Testing
Unit Testing
Integration Testing
System Testing
Acceptation Testing
Security Testing
Performance Testing
Usability Testing
Compatibility Testing
-White Box Testing
-Gorilla Testing
Gray Box Testing
-End to End Testing
-Black Box Testing
-Smoke Testing
-Sanity Testing
-Happy path Testing
-Monkey Testing
-Alpha Testing
-Beta Testing
-OAT
Penetration Testing
-Load testing
-Stress Testing
-Soak Testing
-Volume Testing
-Endurance Testing
-Scalability Testing
-Exploratory Testing
-Cross browser Testing
-Accessibility Testing

# Software Testing

- What is Testing?

Many people understand many definitions of testing **:**

1. Testing is the process of demonstrating that errors are not present.
2. The purpose of testing is to show that a program performs its intended functions correctly.
3. Testing is the process of establishing confidence that a program does what it is supposed to do.

**These definitions are incorrect.**

# Software Testing

A more appropriate definition is:

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

# Software Testing

- Why should We Test ?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

# Software Testing

## Who should Do the Testing ?

- Testing requires the developers to find errors from their software.
- It is difficult for software developer to point out errors from own creations.
- Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

# Software Testing

- What should We Test ?

We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are $2^8 x 2^8$. If only one second it required to execute one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.

# Characteristics of Testable Software (continued)

- Simple
  - The program should exhibit functional, structural, and code simplicity
- Stable
  - Changes to the software during testing are infrequent and do not invalidate existing tests
- Understandable
  - The architectural design is well understood; documentation is available and organized

# Test Characteristics

- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be “best of breed”
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

# Software Testing

## Some Terminologies

- **Error, Mistake, Bug, Fault and Failure**

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "**bugs**".

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

# Software Testing

## ➢ 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.

| Test Case ID | |
|---|---|
| Section-I<br>(Before Execution) | Section-II<br>(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. 2:** Test case template

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

# Software Testing

- Verification and Validation

**Verification** is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

**Validation** is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

**Testing= Verification+Validation**

# Software 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.

# Software Testing

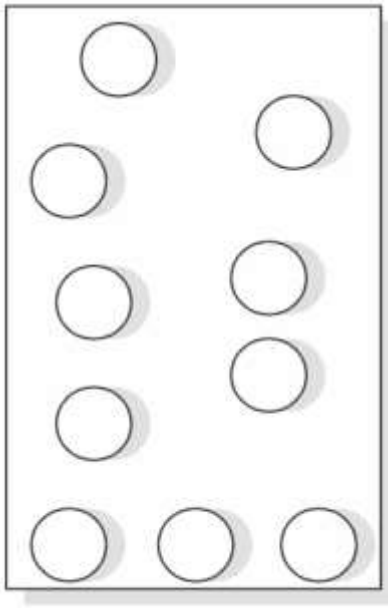
## Levels of Testing

There are 3 levels of testing:

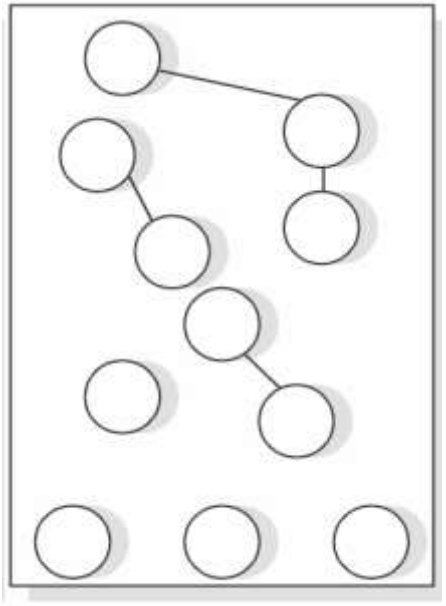
i. Unit Testing

ii. Integration Testing

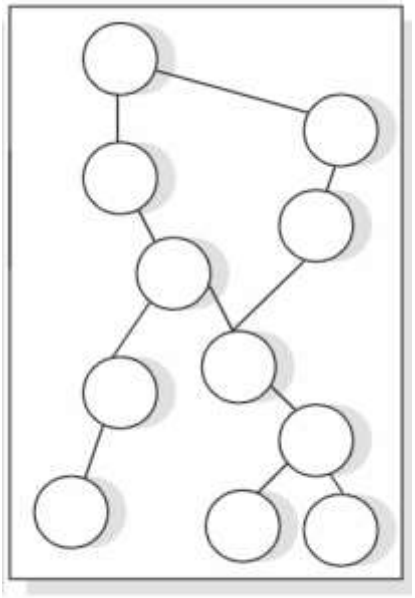
iii. System Testing



UNIT TESTING



INTEGRATION TESTING



SYSTEM TESTING

# Unit Testing

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

# Software Testing

## Unit Testing

There are number of reasons in support of unit testing than testing the entire product.

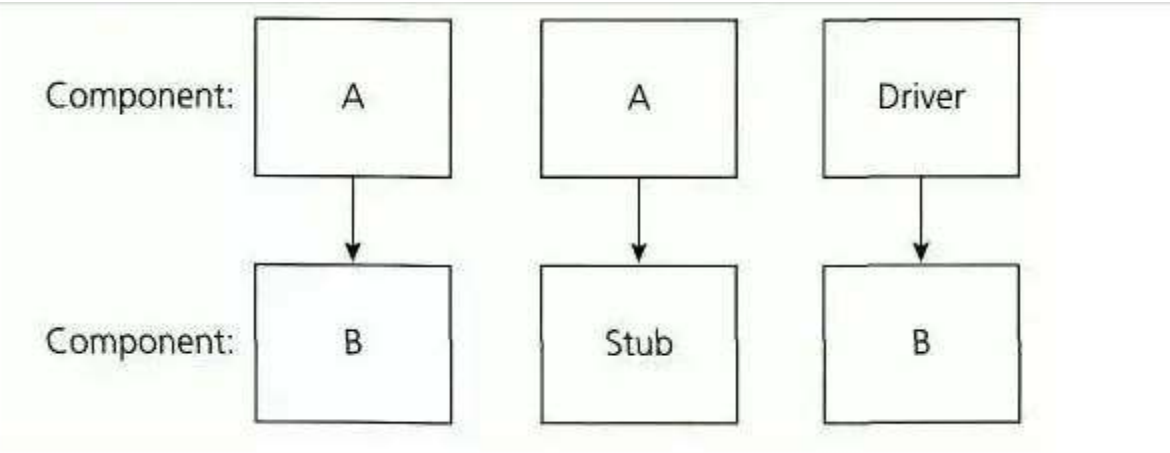
1. The size of a single module is small enough that we can locate an error fairly easily.

2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.

3. Confusing interactions of multiple errors in widely different parts of the software are eliminated.

# Driver and Stub

- The concept of Stubs and Drivers are mostly used in the case of component testing.
- Component testing may be done in isolation with the rest of the system depending upon the context of the development cycle.
- Stubs and drivers are used to replace the missing software and simulate the interface between the software components in a simple manner.



# Driver and Stub..

- Suppose you have a function (Function A) that calculates the total marks obtained by a student in a particular academic year.
- Suppose this function derives its values from another function (Function b) which calculates the marks obtained in a particular subject.
  You have finished working on Function A and wants to test it. But the problem you face here is that you can't seem to run the Function A without input from Function B; Function B is still under development. In this case, you create a dummy function to act in place of Function B to test your function.
- This dummy function gets called by another function. Such a dummy is called a **Stub.**
- To understand what a driver is, suppose you have finished Function B and is waiting for Function A to be developed. In this case you create a dummy to call the Function B. This dummy is called the **driver**.

# Two Unit Testing Techniques

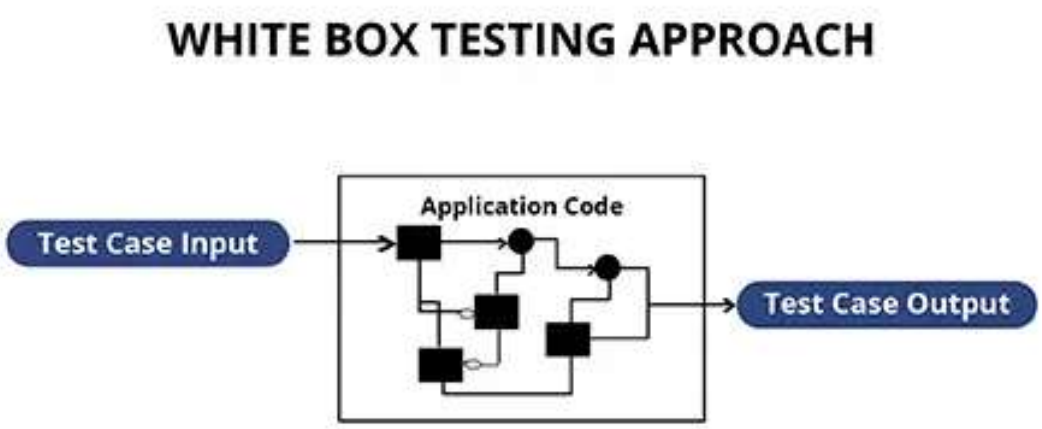
- Black-box testing
  - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software
- White-box testing
  - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
  - Involves tests that concentrate on close examination of procedural detail
  - Logical paths through the software are tested
  - Test cases exercise specific sets of conditions and loops

# White-box Testing

- Uses the control structure part of component-level design to derive the test cases
- These test cases
  - Guarantee that all independent paths within a module have been exercised at least once
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds
  - Exercise internal data structures to ensure their validity

"Bugs lurk in corners and congregate at boundaries"

# What is White Box Testing



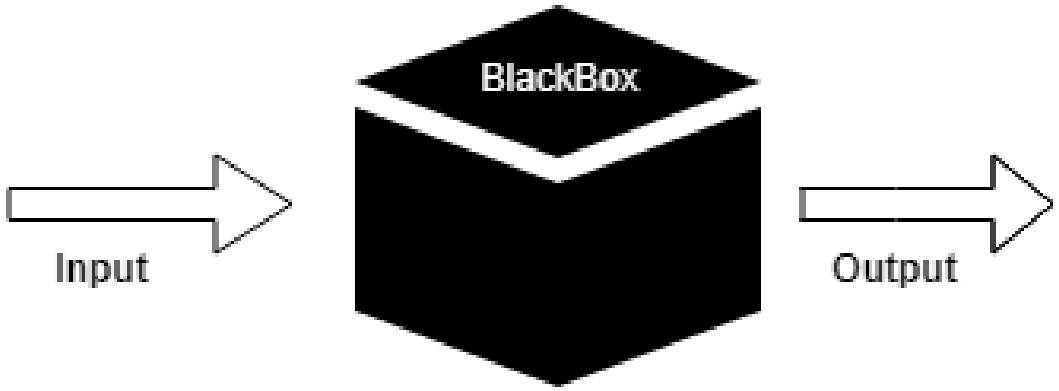
# Black-box Testing

- Complements white-box testing by uncovering different classes of errors
- Focuses on the functional requirements and the information domain of the software
- Used during the later stages of testing after white box testing has been performed
- The tester identifies a set of input conditions that will fully exercise all functional requirements for a program
- The test cases satisfy the following:
  - Reduce, by a count greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  - Tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific task at hand

# Black-box Testing Categories

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

# Black-Box Testing



# Software Testing

## Functional Testing

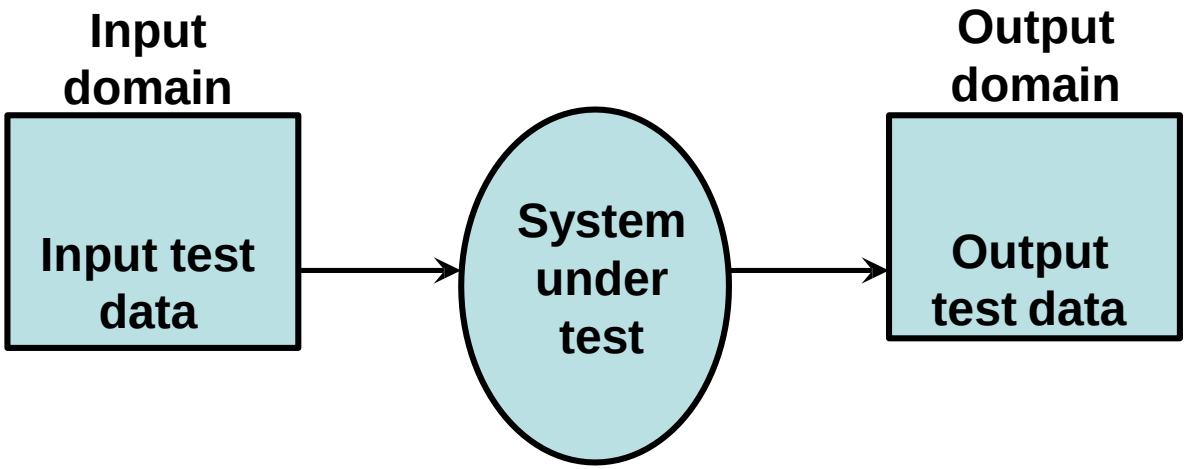
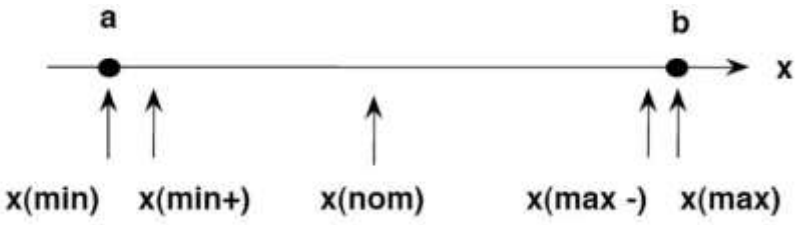



**Fig. 3:** Black box testing

# Boundary Testing

- Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.
- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".
- Thus for a program of **n** variables , the Boundary value analysis yields, **4n+1** test cases.
- The basic idea in boundary value testing is to select input variable values at their:
- Minimum
- Just above the minimum
- A nominal value
- Just below the maximum
- Maximum



# Software Testing

**Example- 8.I**

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

# Software Testing

**Solution**

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

# Software Testing

The boundary value test cases are :

| *Test Case* | *a* | *b* | *c* | *Expected output* |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not Quadratic |
| 2 | 1 | 50 | 50 | Real Roots |
| 3 | 50 | 50 | 50 | Imaginary Roots |
| 4 | 99 | 50 | 50 | Imaginary Roots |
| 5 | 100 | 50 | 50 | Imaginary Roots |
| 6 | 50 | 0 | 50 | Imaginary Roots |
| 7 | 50 | 1 | 50 | Imaginary Roots |
| 8 | 50 | 99 | 50 | Imaginary Roots |
| 9 | 50 | 100 | 50 | Equal Roots |
| 10 | 50 | 50 | 0 | Real Roots |
| 11 | 50 | 50 | 1 | Real Roots |
| 12 | 50 | 50 | 99 | Imaginary Roots |
| 13 | 50 | 50 | 100 | Imaginary Roots |

# Equivalent Class Partitioning

- Equivalent Class Partitioning is a black box technique (code is not visible to tester) which can be applied to all levels of testing like unit, integration, system, etc
- In this technique, you divide the set of test condition into a partition that can be considered the same.
- It divides the input data of software into different equivalence data classes.
- We can apply this technique, where there is a range in input field.

# Software Testing

## Equivalence Class Testing

In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value.

**Two steps are required to implementing this method:**

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.

# Software Testing

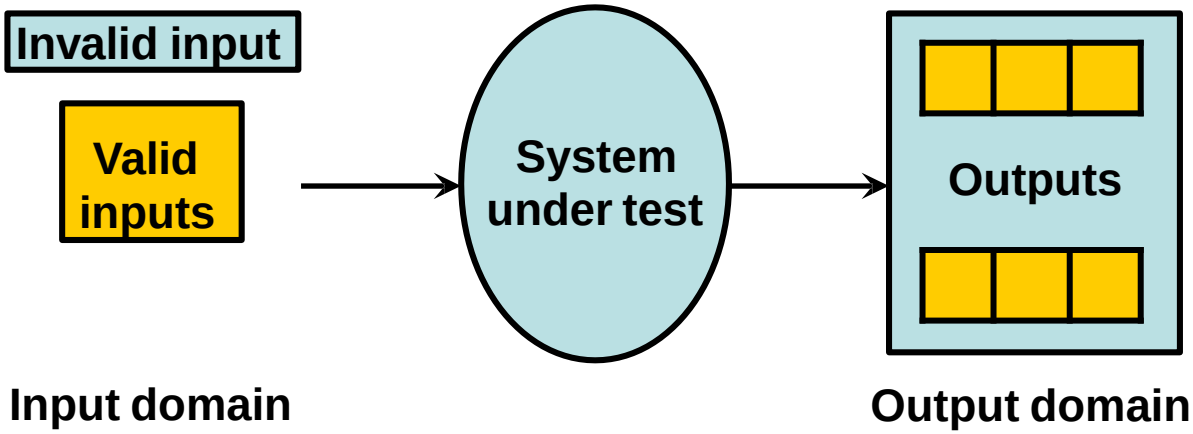
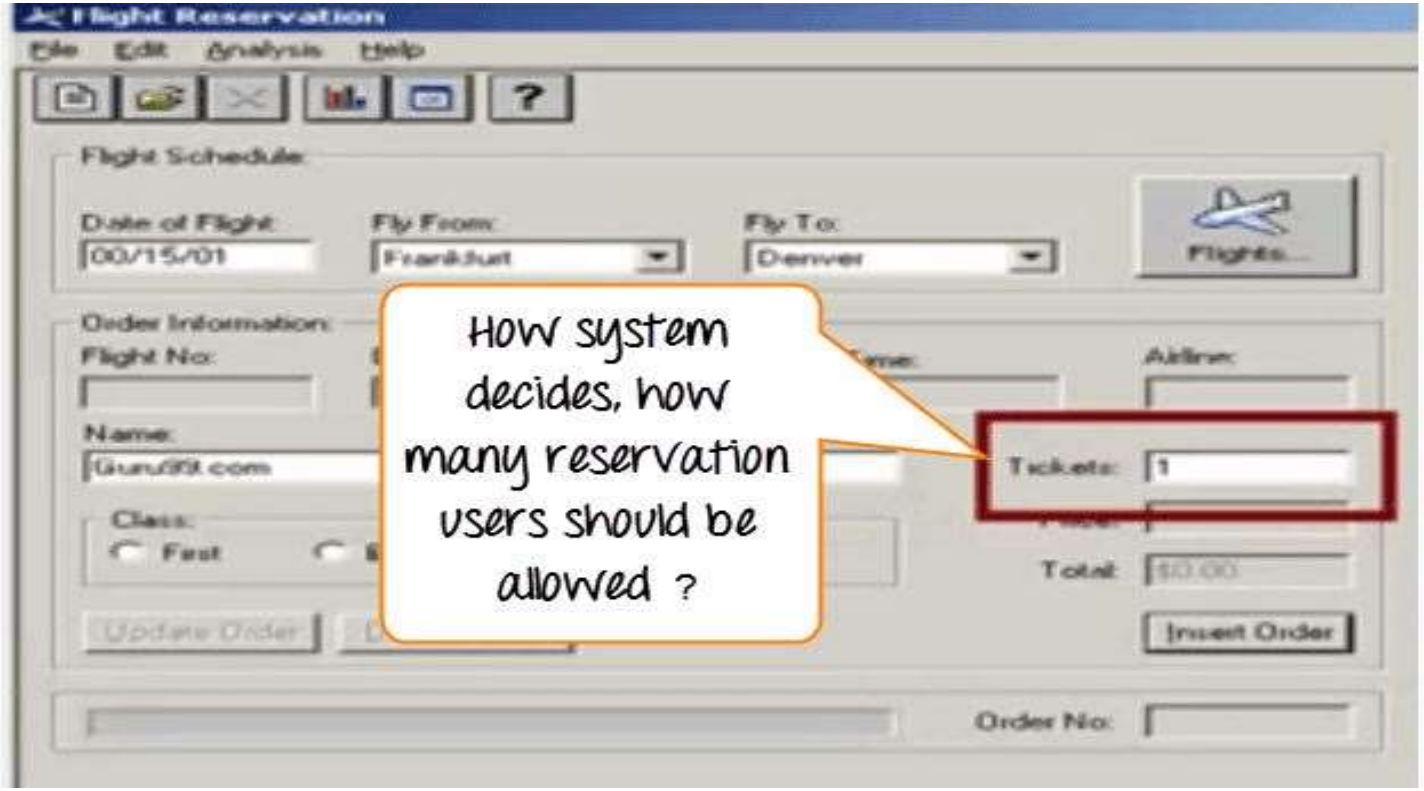



Fig. 7: Equivalence partitioning

Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domain.

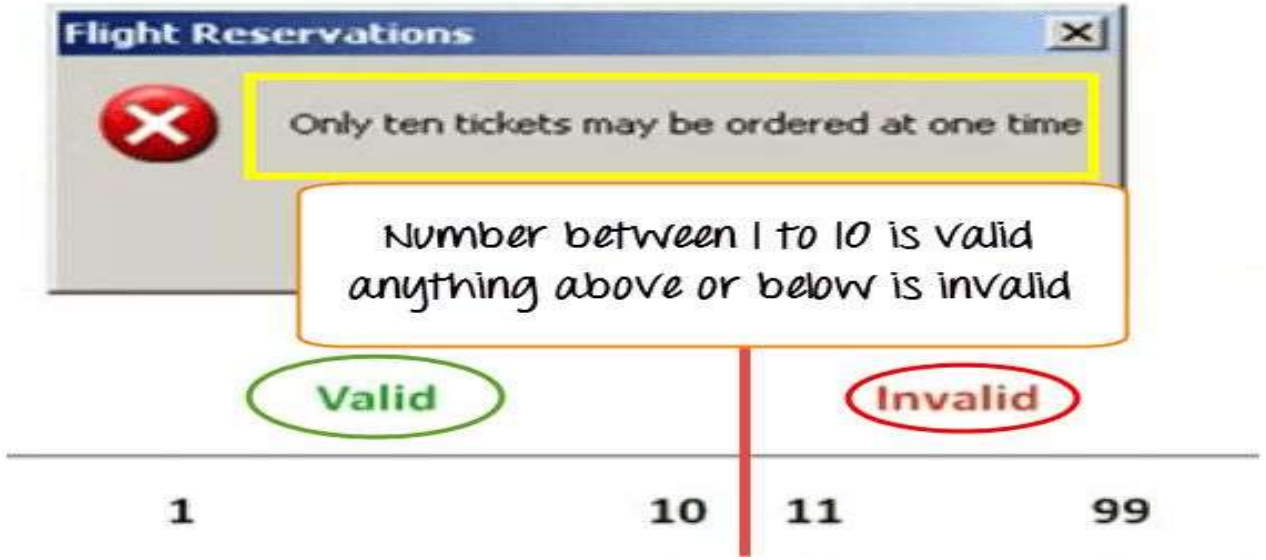
# Example : Equivalence and Boundary Value

- Let's consider the behavior of tickets in the Flight reservation application, while booking a new flight.



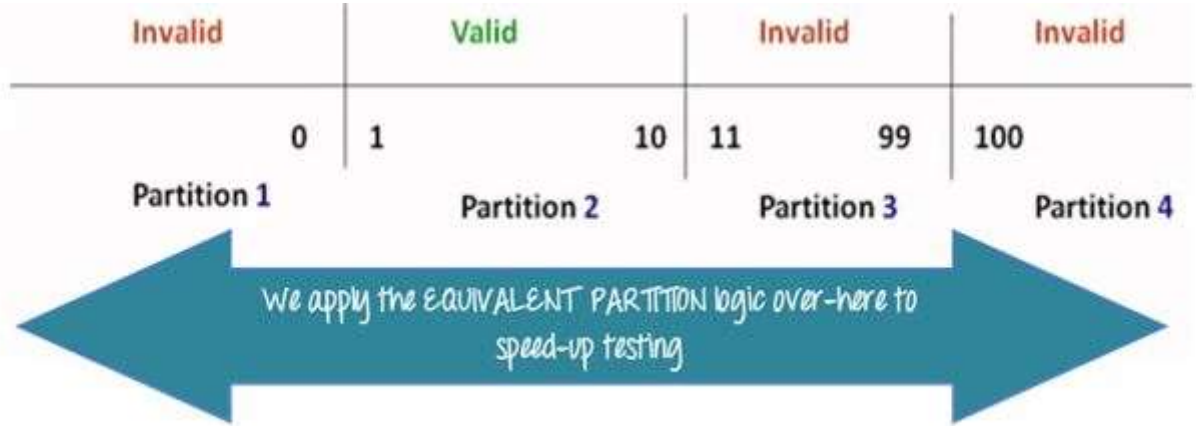
# Example

- Ticket values 1 to 10 are considered valid & ticket is booked. While value 11 to 99 are considered invalid for reservation and error message will appear, **"Only ten tickets may be ordered at one time."**



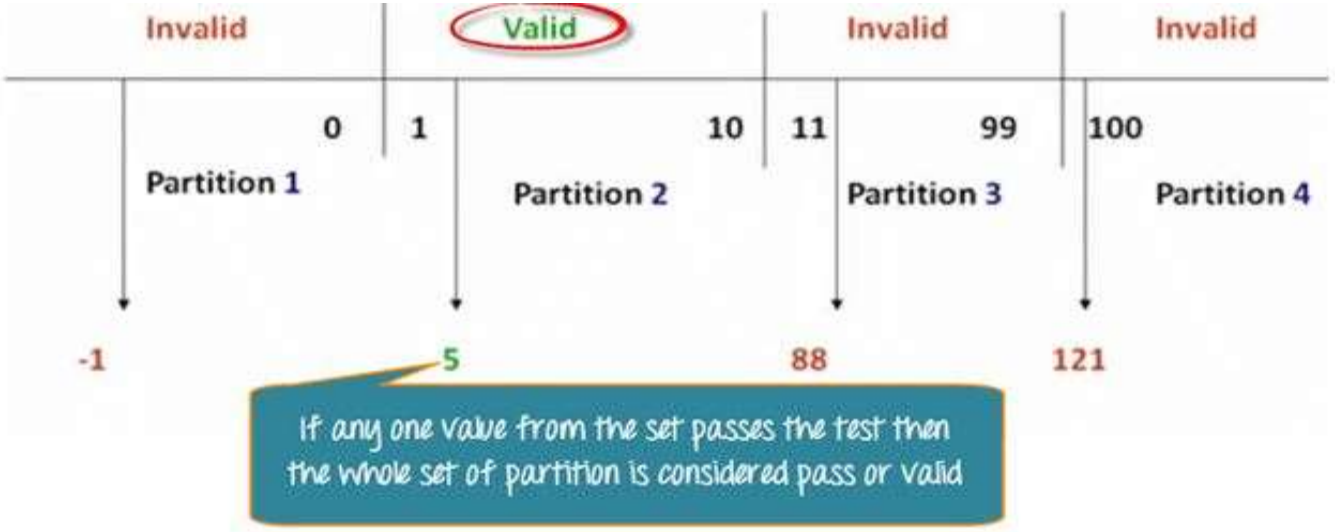
# Example: Test Condition

- Any Number greater than 10 entered in the reservation column (let say 11) is considered invalid.
- Any Number less than 1 that is 0 or below, then it is considered invalid.
- Numbers 1 to 10 are considered valid
- Any 3 Digit Number say -100 is invalid.



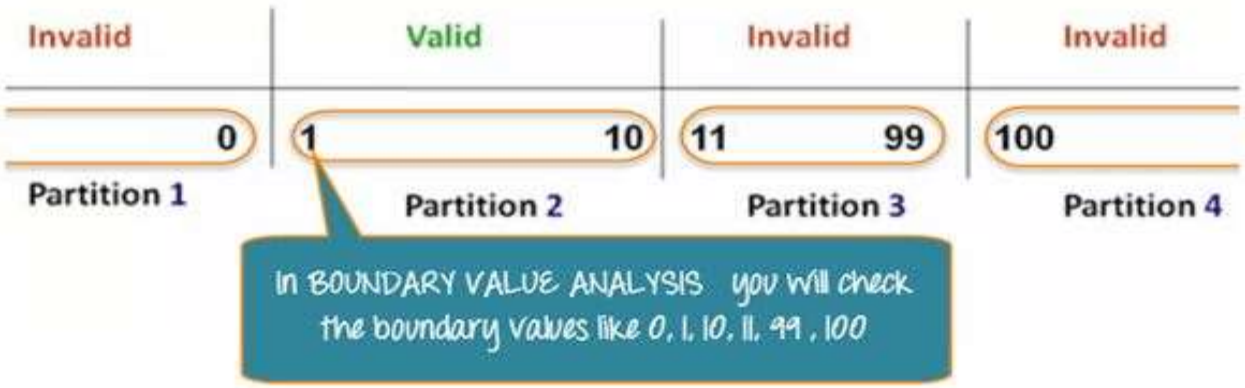
# Example

- The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is **that if one condition/value in a partition passes all others will also pass**. Likewise**, if one condition in a partition fails, all other conditions in that partition will fail**.



# Example: Boundary Value Analysis

- **Boundary Value Analysis-** in Boundary Value Analysis, you test boundaries between equivalence partitions
- In our earlier example instead of checking, one value for each partition you will check the values at the partitions like 0, 1, 10, 11 and so on. As you may observe, you test values at **both valid and invalid boundaries**. Boundary Value Analysis is also called **range checking**.



# Software Testing

**Example 8.7**

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Identify the equivalence class test cases for output and input domains.

# Software Testing

- **Solution**
- Output domain equivalence class test cases can be identified as follows: $O_1$={<a,b,c>:Not a quadratic equation if a = 0}
  - $O_1$={<a,b,c>:Real roots if ($b^2$-4ac)>0}
  - $O_1$={<a,b,c>:Imaginary roots if ($b^2$-4ac)<0}

  $O_1$={<a,b,c>:Equal roots if ($b^2$-4ac)=0}`
- The number of test cases can be derived form above relations and shown below:

| *Test case* | *a* | *b* | *c* | *Expected output* |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not a quadratic equation |
| 2 | 1 | 50 | 50 | Real roots |
| 3 | 50 | 50 | 50 | Imaginary roots |
| 4 | 50 | 100 | 50 | Equal roots |

# Software Testing

We may have another set of test cases based on input domain.

$I_1$= {a: a = 0}

$I_2$= {a: a < 0}

$I_3$= {a: 1 ≤ a ≤ 100}

$I_4$= {a: a > 100}

$I_5$= {b: 0 ≤ b ≤ 100}

$I_6$= {b: b < 0}

$I_7$= {b: b > 100}

$I_8$= {c: 0 ≤ c ≤ 100}

$I_9$= {c: c < 0}

$I_{10}$={c: c > 100}

# Software Testing

| Test Case | a | b | c | Expected output |
|---|---|---|---|---|
| 1 | 0 | 50 | 50 | Not a quadratic equation |
| 2 | -1 | 50 | 50 | Invalid input |
| 3 | 50 | 50 | 50 | Imaginary Roots |
| 4 | 101 | 50 | 50 | invalid input |
| 5 | 50 | 50 | 50 | Imaginary Roots |
| 6 | 50 | -1 | 50 | invalid input |
| 7 | 50 | 101 | 50 | invalid input |
| 8 | 50 | 50 | 50 | Imaginary Roots |
| 9 | 50 | 50 | -1 | invalid input |
| 10 | 50 | 50 | 101 | invalid input |

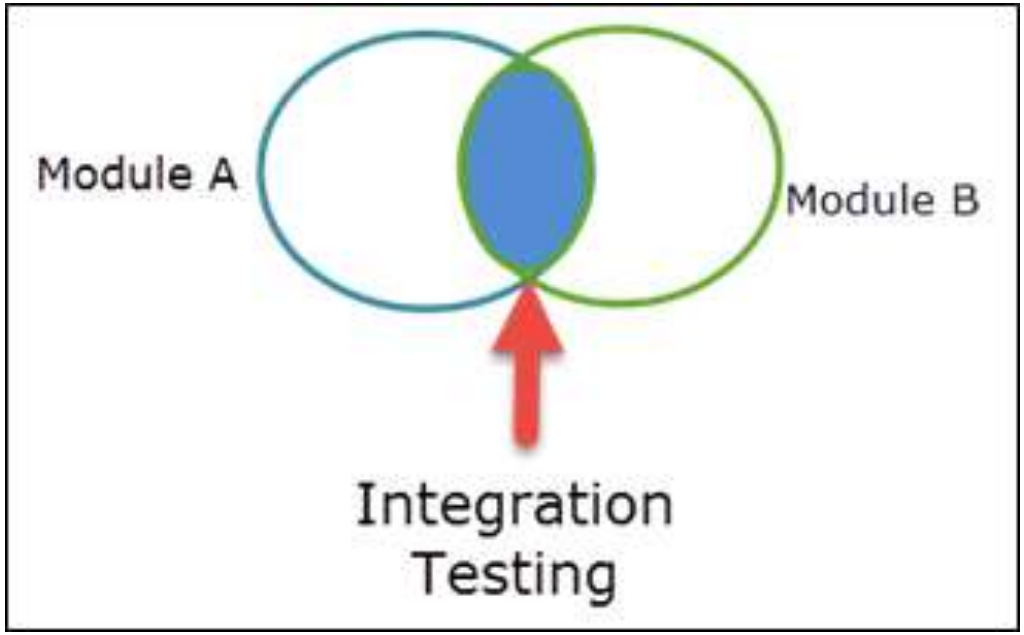
Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

# Software Testing

## Integration Testing

The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed. One specific target of integration testing is the interface: whether parameters match on both sides as to type, permissible ranges, meaning and utilization.

# Integration Testing



# Incremental Integration Testing

- Three kinds
  - Top-down integration
  - Bottom-up integration
  - Sandwich integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

# Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
  - DF: All modules on a major control path are integrated
  - BF: All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

# Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
  - This approach verifies low-level data processing early in the testing process
  - Need for stubs is eliminated
- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

# Sandwich Integration

- Consists of a combination of both top-down and bottom-up integration
- Occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next
  - High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
  - Integration within the group progresses in alternating steps between the high and low level modules of the group
  - When integration for a certain functional group is complete, integration and testing moves onto the next group
- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
- Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario

# Software Testing

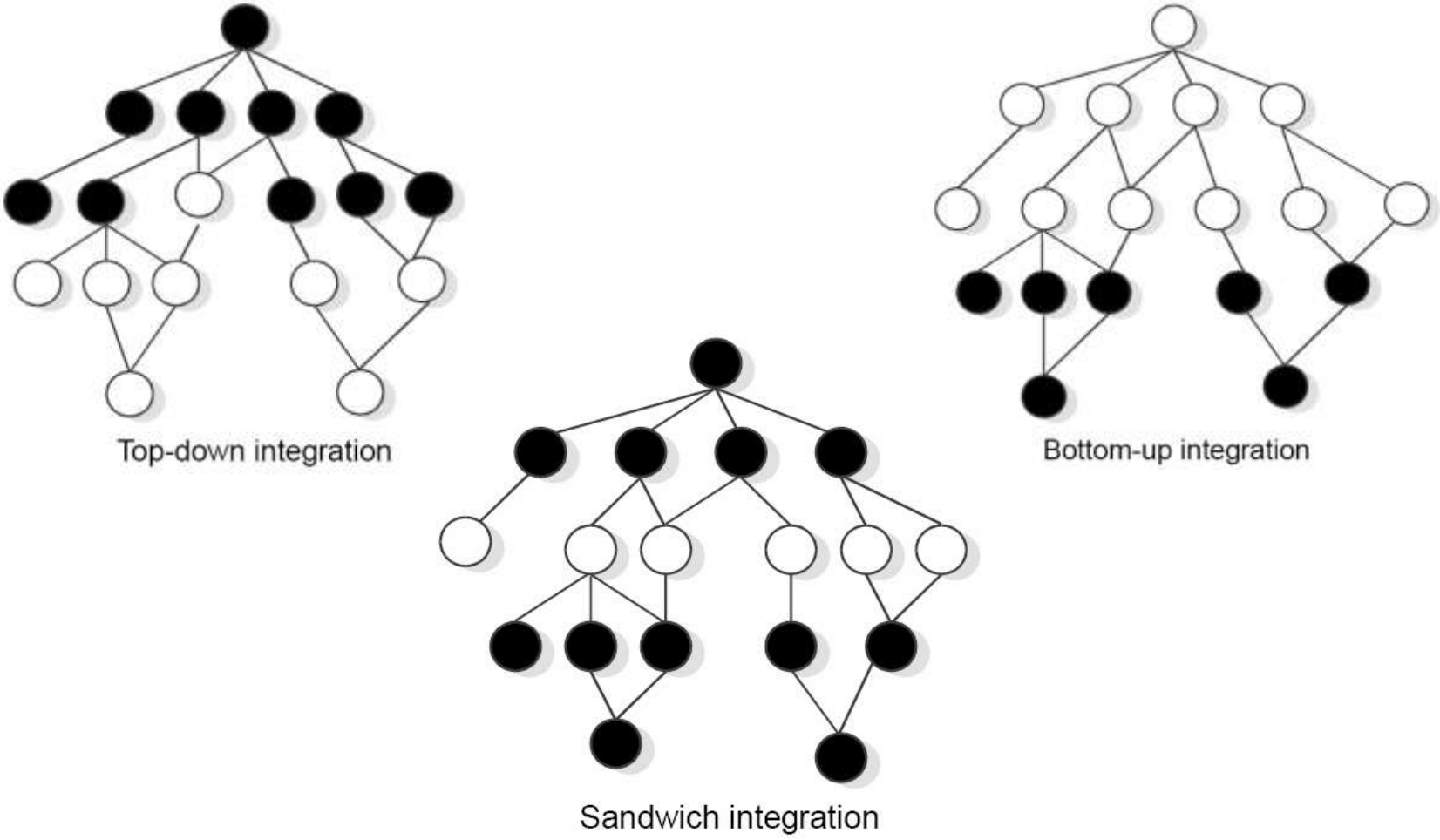



**Fig. 30 : Three different integration approaches**

# Different Types of System Testing

- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- Stress testing
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance testing
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure

# Regression Testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools
- Regression test suite contains three different classes of test cases
  - A representative sample of tests that will exercise all software functions
  - Additional tests that focus on software functions that are likely to be affected by the change
  - Tests that focus on the actual software components that have been changed

# Smoke Testing

**Smoke Testing** is a software testing process that determines whether the deployed software build is stable or not. Smoke testing is a confirmation for QA team to proceed with further software testing. It consists of a minimal set of tests run on each build to test software functionalities. Smoke testing is also known as "Build Verification Testing" or "Confidence Testing."

- Taken from the world of hardware
  - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software is compiled and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily
    - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
  - After a smoke test is completed, detailed test scripts are executed

# Benefits of Smoke Testing

- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work
  - Managers get a good indication that progress is being made