BITS Pilani presentation

BITS Pilani
Pilani Campus

Dr. Nagesh BS

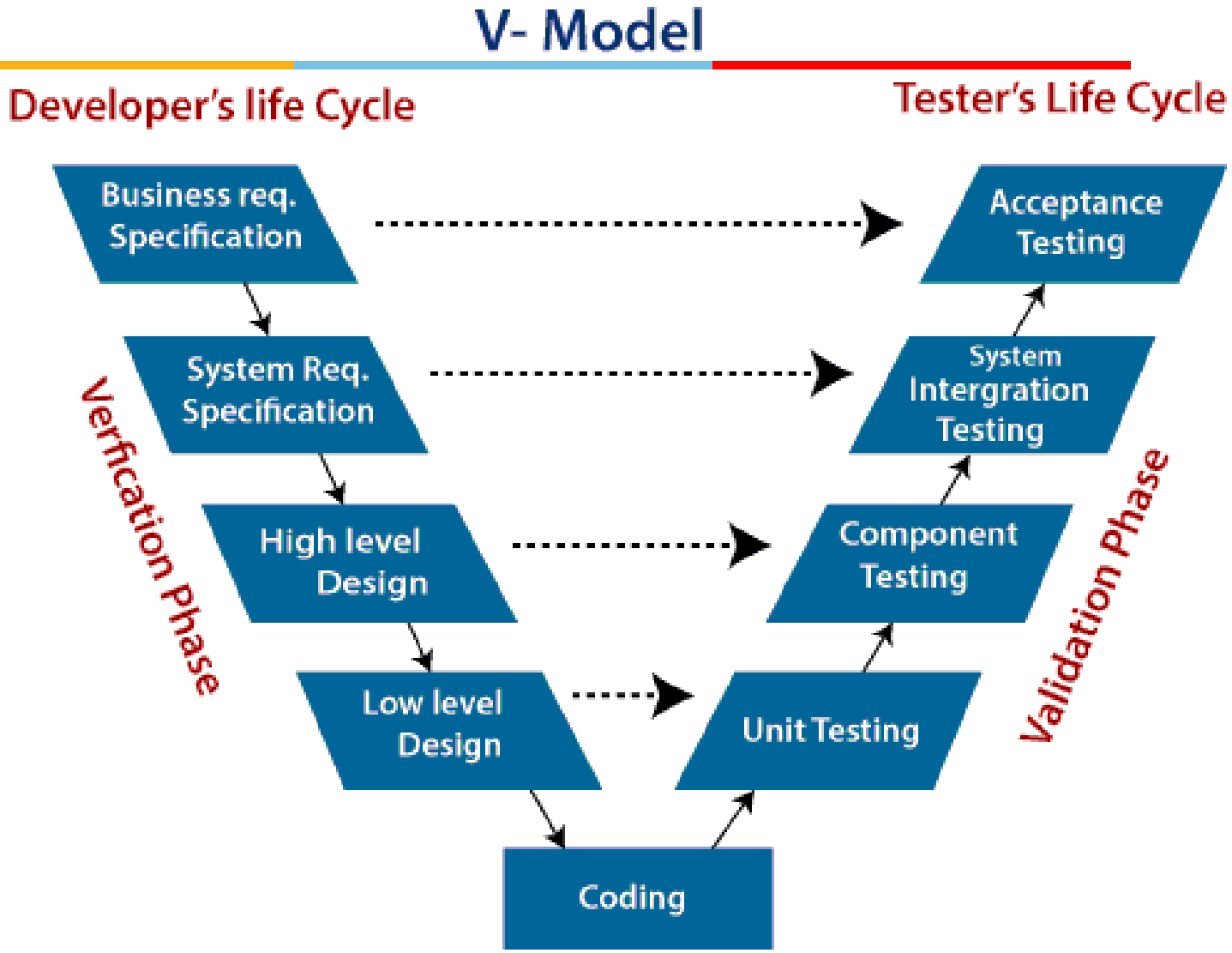
innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG501
Software Quality Assurance and Testing
Lecture No. 5

Deep driving SQA: Software Testing Techniques

V-Model in Software Testing



V-Model in Software testing is an SDLC model where the **test execution** takes place in a hierarchical manner. The execution process makes a V-shape.

It is also called a Verification and Validation model that undertakes the testing process for every development phase.

According to the waterfall model, testing is a post-development activity.

The spiral model took one step further by breaking the product into increments each of which can be tested separately.

V-model brings in a new perspective that different types of testing apply at different levels.

The V-model splits testing into two parts

- DESIGN
- EXECUTION.

Verification phases on one side and the Validation phases on the other side.

Verification and Validation process is joined by coding phase in V-shape.

Test: Testing is concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals—**to find failures or to demonstrate correct execution.**

Test case: A test case has an **identity** and is associated with program behavior. A test case also has a set of **inputs and a list of expected outputs**. The essence of software testing is to **determine a set of test cases for the item to be tested.**

Test case template

innovate achieve lead

```
Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs
Postconditions
Execution History
Date        Result        Version        Run By
```

Test suite: A collection of **test scripts or test cases** that is used for validating bug fixes (or finding new bugs) within a logical or physical area of a product.

For example, an acceptance test suite contains all of the test cases that were used to verify that the software has met certain predefined acceptance criteria.

Test script: The step-by-step instructions that describe how a test case is to be executed. It may contain one or more test cases.

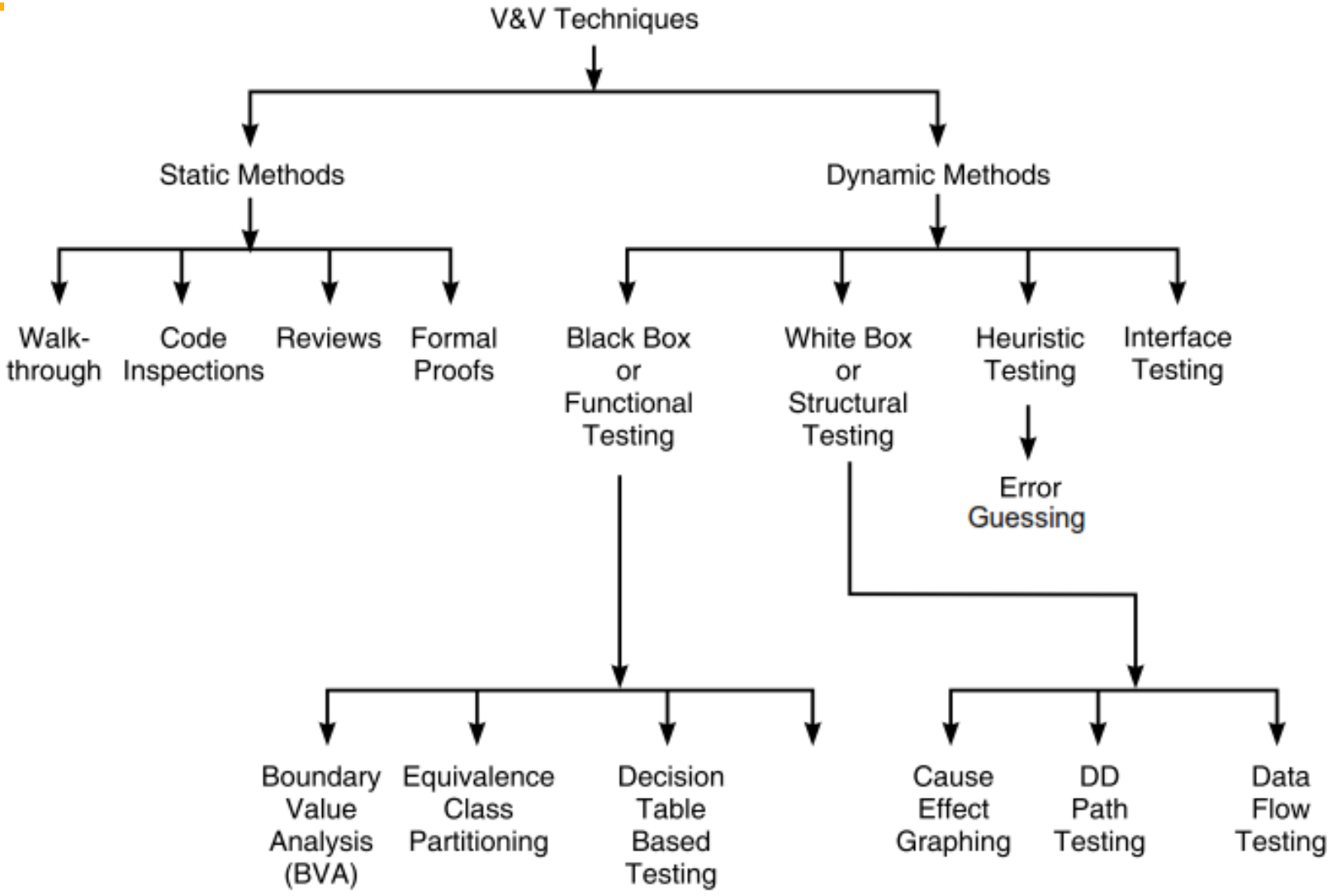
Test cases for ATM:

Preconditions: System is started.

Test case ID	Test case name	Test case description	Test steps			Test status (P/F)	Test priority
			Step	Expected result	Actual result		
Session 01	Verify Card	To verify whether the system reads a customer's ATM card.	Insert a readable card	Card is accepted; System asks for entry of PIN			High
			Insert an unreadable card	Card is ejected; System displays an error screen; System is ready to start a new session			High
	Validate PIN	To verify whether the system accepts customer's PIN	Enter valid PIN	System displays a menu of trans-action types			High
			Enter invalid PIN	Customer is asked to re-enter			High

Test case ID	Test case name	Test case description	Test steps			Test status (P/F)	Test priority
			Step	Expected result	Actual result		
			Enter incorrect PIN the first time, then correct PIN the second time	System displays a menu of transaction types.			High
			Enter incorrect PIN the first time and second time, then correct PIN the third time	System displays a menu of transaction types.			High
			Enter incorrect PIN three times	An appropriate message is displayed; Card is retained by machine; Session is terminated			High

CATEGORIZING V&V TECHNIQUES



BLACK-BOX (OR FUNCTIONAL TESTING)

- The term Black-Box refers to the software which is treated as a black-box.
- The system or source code is not checked at all.
- It is done from the customer's viewpoint.
- The test engineer engaged in black-box testing only knows the set of inputs and expected outputs and is unaware of how those inputs are transformed into outputs by the software.

BOUNDARY VALUE ANALYSIS (BVA)

It is a **black-box testing technique** based on the principle that defects are more likely to occur at the boundaries of input ranges rather than in the middle. This is because boundary conditions are more prone to errors due to incorrect implementation of logical conditions. This is done for the following reasons:

i. Programmers usually are not able to decide whether they have to use <= operator or < operator when trying to make comparisons.

ii. Different terminating conditions of for-loops, while loops, and repeat loops may cause defects to move around the boundary conditions.

iii. The requirements themselves may not be clearly understood, especially around the boundaries, thus causing even the correctly coded program to not perform the correct way.

The basic idea of BVA is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum.

$$\{min, min+, nom, max–, max\}$$

- When more than one variable for the same application is checked then one can use a single fault assumption.
- Holding all but one variable to the extreme value and allowing the remaining variable to take the extreme value.

BVA is based upon a critical assumption that is known as single fault assumption theory.

The **single fault assumption** states that if an error occurs, it is due to a fault in only **one variable at a time**, while all other variables are kept constant at their extreme values.

For example, if a system has **n input variables**, when testing one variable, all other variables are held at their minimum or maximum values while the selected variable is tested at its boundary value

- For n variable to be checked:
  - ***Maximum of 4n+1 test cases***

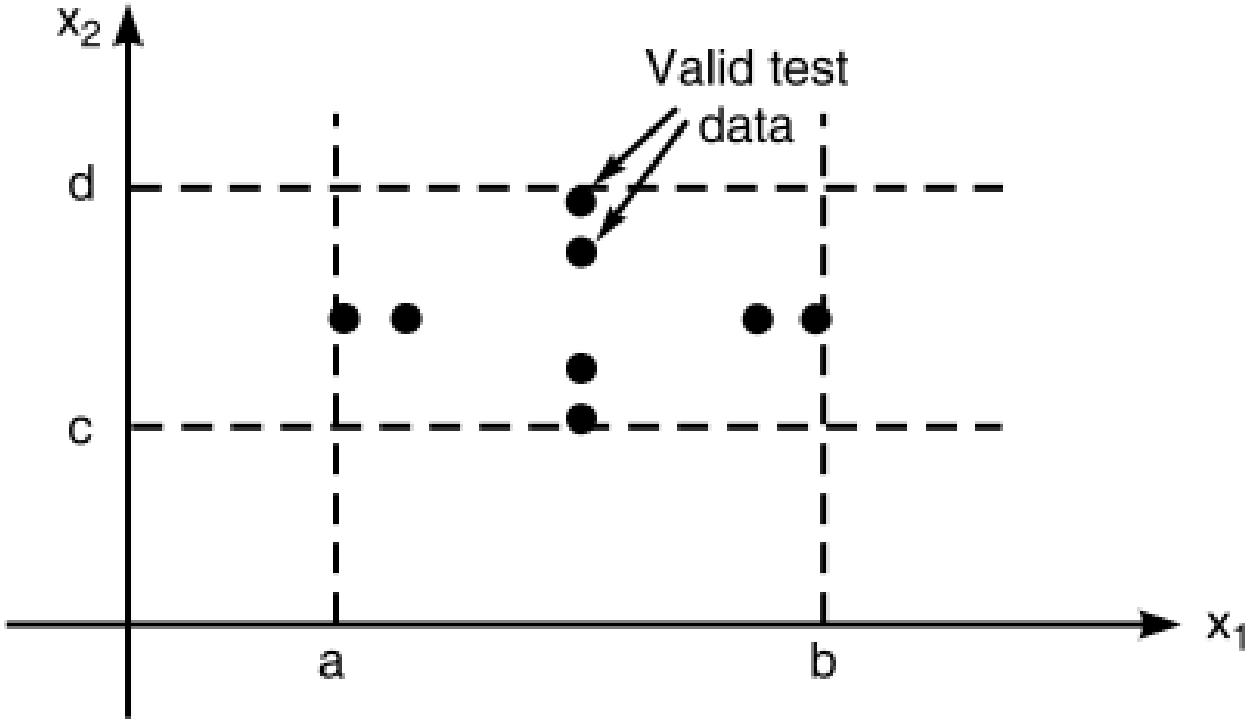


FIGURE 3.1 BVA Test Cases.

Problem: Consider a Program for determining the Previous Date.

innovate achieve lead

Input: *Day, Month, Year with valid ranges as-*

1 ≤ Month≤12

1 ≤ Day ≤31

1900 ≤ Year ≤ 2000

Design Boundary Value Test Cases.

Solution:

1) Year as a Single Fault Assumption

Test Cases	Month	Day	Year	Output
1	6	15	1900	14 June 1900
2	6	15	1901	14 June 1901
3	6	15	1960	14 June 1960
4	6	15	1999	14 June 1999
5	6	15	2000	14 June 2000

Day as Single Fault Assumption

Test Case	Month	Day	Year	Output
6	6	1	1960	31 May 1960
7	6	2	1960	1 June 1960
8	6	30	1960	29 June 1960
9	6	31	1960	Invalid day

Month as Single Fault Assumption

Test Case	Month	Day	Year	Output
10	1	15	1960	14 Jan 1960
11	2	15	1960	14 Feb 1960
12	11	15	1960	14 Nov 1960
13	12	15	1960	14 Dec 1960

For the n variable to be checked Maximum of 4n + 1 test case will be required.

Therefore, for n = 3, the maximum test cases are

$$4 \times 3 + 1 = 13$$

Consider a system that accepts ages from 18 to 56.

innovate achieve lead

Boundary Value Analysis(Age accepts 18 to 56)		
Invalid (min-1)	Valid (min, min + 1, nominal, max – 1, max)	Invalid (max + 1)
17	18, 19, 37, 55, 56	57

Valid Test cases: Valid test cases for the above can be any value entered greater than 17 and less than 57.

Enter the value- 18.

Enter the value- 19.

Enter the value- 37.

Enter the value- 55.

Enter the value- 56.

Invalid Testcases: When any value less than 18 and greater than 56 is entered.

Enter the value- 17.

Enter the value- 57.

EQUIVALENCE CLASS TESTING

The use of equivalence classes as the basis for functional testing has two motivations:

a. We want exhaustive testing

b. We want to avoid redundancy

This is not handled by the BVA technique as we can see massive redundancy in the tables of test cases.

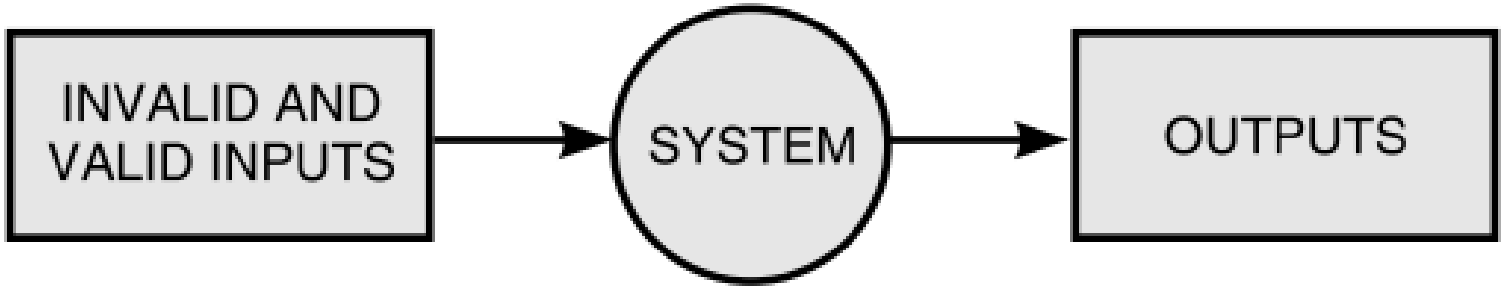


FIGURE 3.4 Equivalence Class Partitioning.

Process

- The input and the output domain **is divided into a** finite number of equivalence classes.
- select one representative of each class and test our program against it.
- It is assumed by the tester that if one representative from a class is able to detect error then why should we consider other cases.
- if this single representative test case did not detect any error then we assume that no other test case of this class can detect error.
- we consider both valid and invalid input domains.

- The key and the craftsmanship lies in the choice of the equivalence relation that determines the classes.
- The equivalence class testing can be categorized into four different types.
- **Weak Normal Equivalence Class Testing:** In this first type of equivalence class testing, one variable from each equivalence class is tested by the team. Moreover, the values are identified in a systematic manner. Weak normal equivalence class testing is also known as **single fault assumption**.

Strong Normal Equivalence Class Testing: Termed as **multiple fault assumption**, in strong normal equivalence class testing the team selects test cases from each element of the Cartesian product of the equivalence. This ensures the notion of completeness in testing, as it covers all equivalence classes and offers the team of each possible combinations of inputs.

Weak Robust Equivalence Class Testing: Like weak normal equivalence, weak robust testing too tests one variable from each equivalence class. However, unlike the former method, it is also focused on testing test cases for invalid values.

Strong Robust Equivalence Class Testing: Another type of equivalence class testing, strong robust testing produces test cases for all valid and invalid elements of the product of the equivalence class. However, it is incapable of reducing the redundancy in testing.

White Box Testing

- White-box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality.
- White-box testing is used to test the program code, code structure, and the internal design flow.

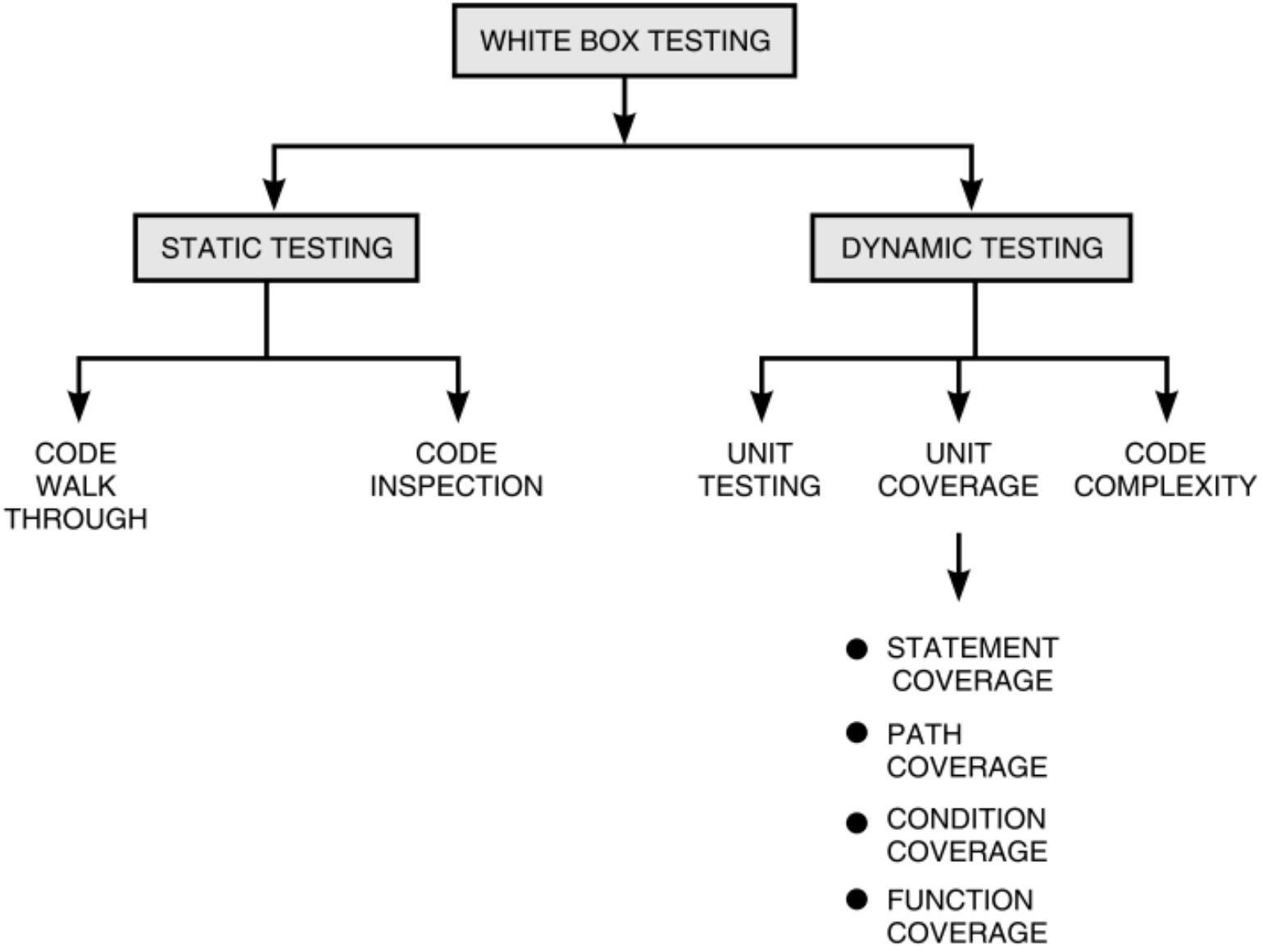


FIGURE 4.1 Classification of White-Box Testing.

CODE COVERAGE TESTING

innovate achieve lead

- 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 the instrumentation of code.
- *STATEMENT COVERAGE*
- *PATH COVERAGE*
- *CONDITION COVERAGE*
- *FUNCTION COVERAGE*

Test levels and types

Three levels of testing:

1. **Unit testing**
2. **Integration testing**
3. **System testing**

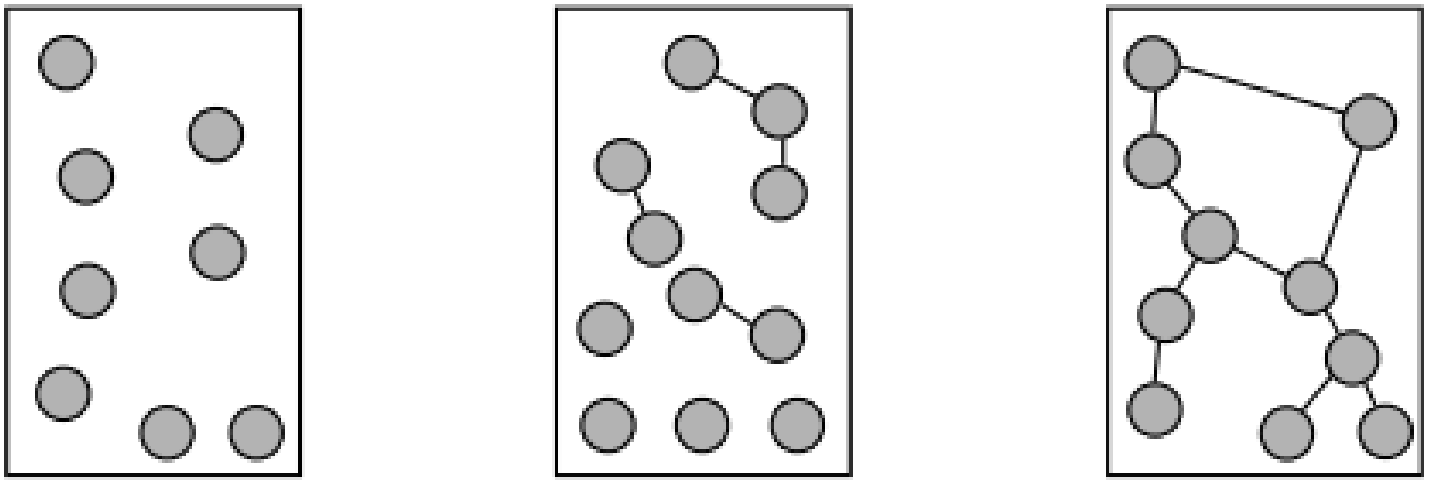
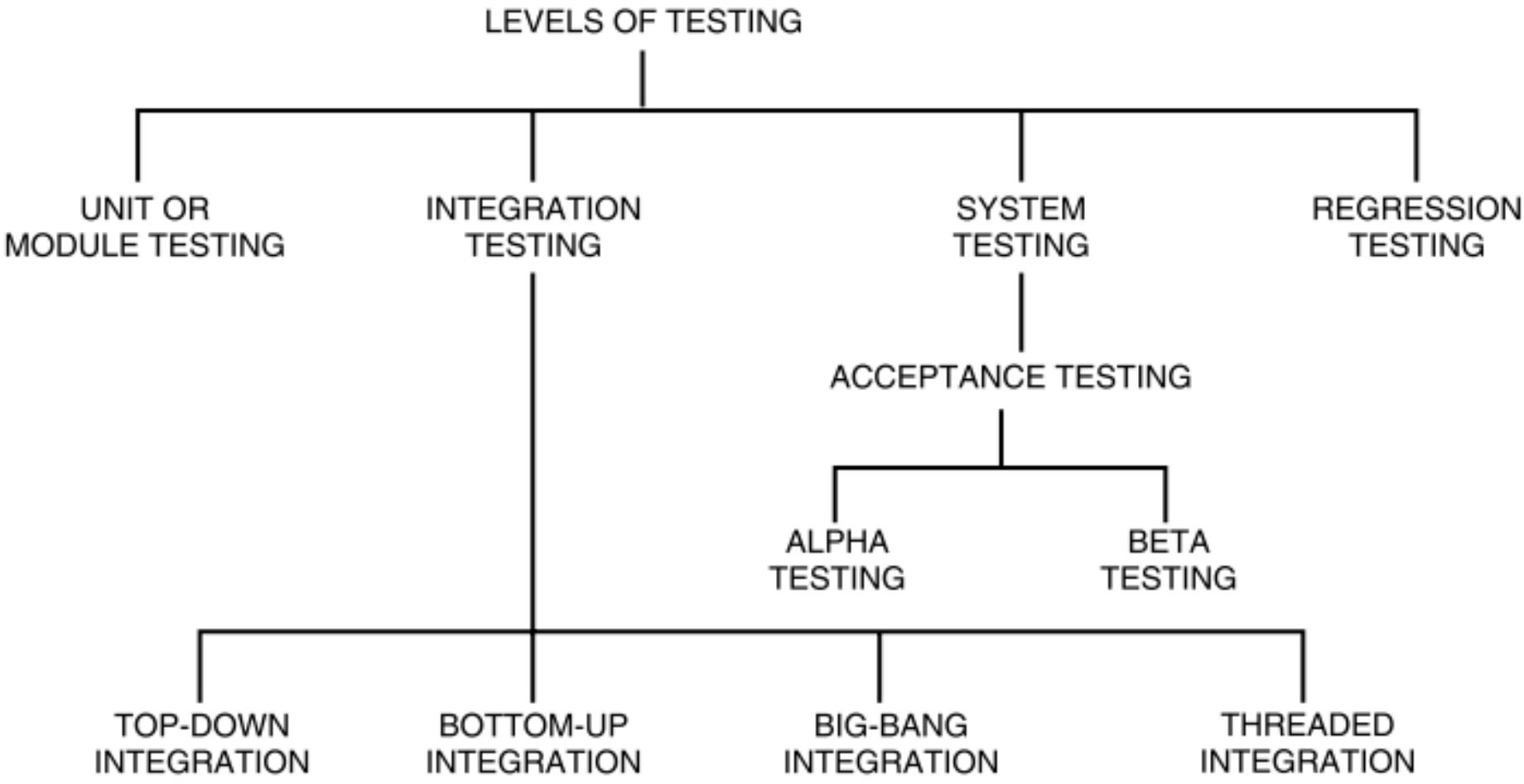


FIGURE 7.1 Levels of Testing.


LEVELS OF TESTING
UNIT OR MODULE TESTING
INTEGRATION TESTING
SYSTEM TESTING
REGRESSION TESTING
ACCEPTANCE TESTING
ALPHA TESTING
BETA TESTING
TOP-DOWN INTEGRATION
BOTTOM-UP INTEGRATION
BIG-BANG INTEGRATION
THREADED INTEGRATION

Unit (or module) testing

Unit (or module) testing "is the process of taking a module (an atomic unit) and running it in isolation from the rest of the software product by using prepared test cases and comparing the actual results with the results predicted by the specification and design module."

It is a white-box testing technique.

Importance of unit testing:

1. Because modules are being tested individually, testing becomes easier.
2. It is more exhaustive.
3. Interface errors are eliminated.

INTEGRATION TESTING

- *A system is composed of multiple components or modules that comprise hardware and software.*
- *Integration is defined as the set of interactions among components.*
- *Testing the interaction between the modules and interaction with other systems externally is called integration testing.*
- *The architecture and design can give the details of interactions within systems.*

Classification of integration testing

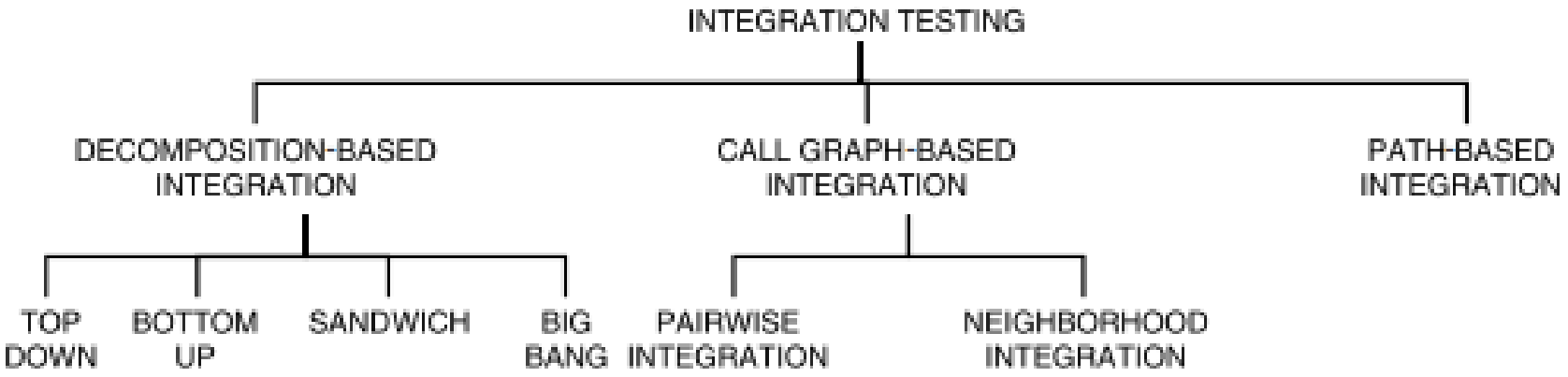


FIGURE 7.4

Decomposition-based Integration:

Decomposition-based integration involves functionally decomposing the system under test into a hierarchical structure, represented as a tree or in textual form. The primary objective is to evaluate the interfaces between individually tested units

Types Of Decomposition-based Techniques :Top-down Integration Approach

It begins with the main program, i.e., the **root of the tree**.

Any lower-level unit that is called by the main program appears as a “stub.”

A stub is a piece of throw-away code that emulates a called unit.

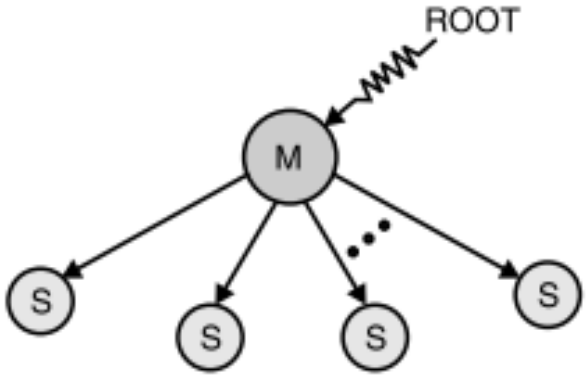


FIGURE 7.5 Stubs.

Bottom-up Integration Approach

It is a mirror image to the top-down order with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree.

start with the leaves of the decomposition tree and test them with specially coded drivers. Less throw-away code exists in drivers than there is in stubs.

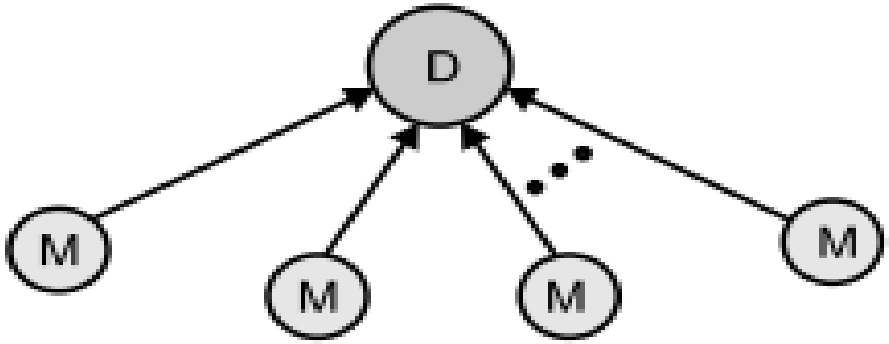


FIGURE 7.6 Drivers.

Sandwich Integration Approach

The **Sandwich Integration Approach**, also known as **Bidirectional Integration**, combines **Top-Down** and **Bottom-Up** integration testing methods.

- **Hybrid Approach**: It merges top-down and bottom-up techniques to accelerate integration.
- **Reduced Stub and Driver Effort**: Since both directions are integrated simultaneously, fewer stubs and drivers are needed.
- **Initial Testing with Stubs & Drivers**: Early integration relies on stubs (to replace lower modules) and drivers (to simulate higher modules).
- **Focus on Key Components**: Emphasis is placed on testing newly developed or critical components efficiently.

Big-bang Integration

- Instead of integrating component by component and testing, **this approach waits until all the components arrive and one round of integration testing is done**. This is known as *big-bang integration.*
- *It reduces testing effort and* **removes duplication in testing** for the multi-step component integrations.
- Big-bang integration is **ideal for a product where the interfaces are stable with fewer number of defects**.

7.2.2.6. GUIDELINES TO CHOOSE INTEGRATION METHOD AND CONCLUSIONS

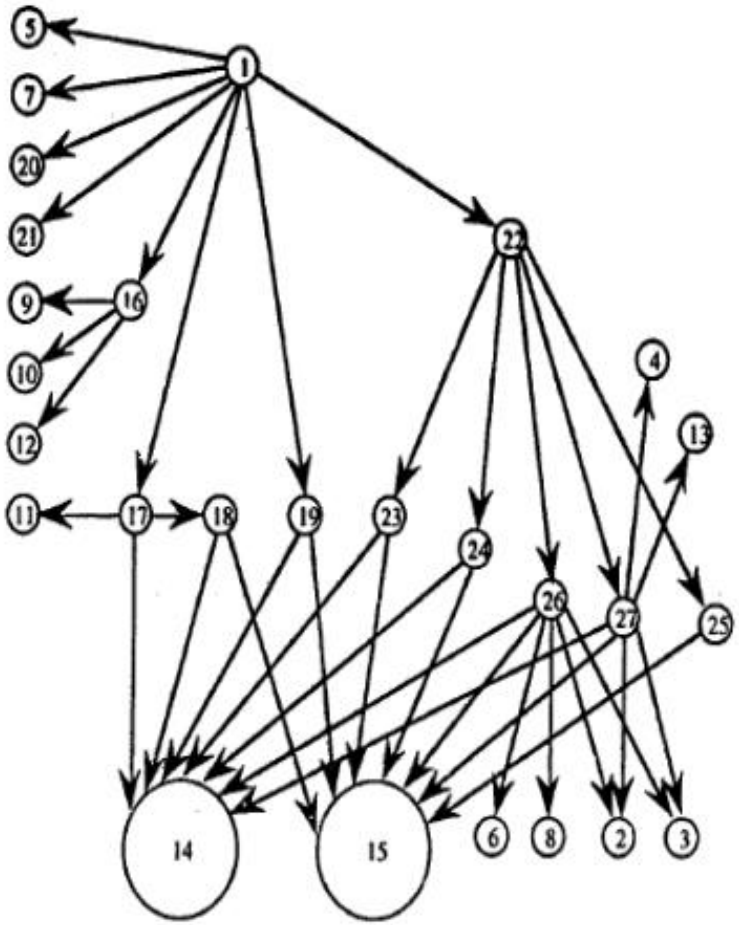
S. No.	Factors	Suggested method
1.	Clear requirements and design.	Top-down approach.
2.	Dynamically changing requirements, design, and architecture.	Bottom-up approach.
3.	Changing architecture and stable design.	Sandwich (or bi-directional) approach.
4.	Limited changes to existing architecture with less impact.	Big-bang method.
5.	Combination of above.	Select any one after proper analysis.

Call Graph-based Integration

- Structural testing(shows **how functions/modules interact**)
- A **Call Graph** is a **directed graph** that represents **function/method calls** within a program.
- Call graph is a directed graph thus, we can use it as a program graph.
    - *Pairwise Integration*
    - *Neighborhood Integration*

Table 13.1 SATM Units and Abbreviated Names

Unit Number	*Level Number*	*Unit Name*
1	1	SATM System
A	1.1	Device Sense & Control
D	1.1.1	Door Sense & Control
2	1.1.1.1	Get Door Status
3	1.1.1.2	Control Door
4	1.1.1.3	Dispense Cash
E	1.1.2	Slot Sense & Control
5	1.1.2.1	WatchCardSlot
6	1.1.2.2	Get Deposit Slot Status
7	1.1.2.3	Control Card Roller
8	1.1.2.3	Control Envelope Roller
9	1.1.2.5	Read Card Strip
10	1.2	Central Bank Comm.
11	1.2.1	Get PIN for PAN
12	1.2.2	Get Account Status
13	1.2.3	Post Daily Transactions
B	1.3	Terminal Sense & Control
14	1.3.1	Screen Driver
15	1.3.2	Key Sensor
C	1.4	Manage Session
16	1.4.1	Validate Card
17	1.4.2	Validate PIN
18	1.4.2.1	GetPIN
F	1.4.3	Close Session
19	1.4.3.1	New Transaction Request
20	1.4.3.2	Print Receipt
21	1.4.3.3	Post Transaction Local
22	1.4.4	Manage Transaction
23	1.4.4.1	Get Transaction Type
24	1.4.4.2	Get Account Type
25	1.4.4.3	Report Balance
26	1.4.4.4	Process Deposit
27	1.4.4.5	Process Withdrawal



Pairwise Integration

- Pairwise Integration reduces the effort needed to create **stubs and drivers** by directly integrating **two related units at a time** from the **call graph**.
- Instead of developing temporary stubs and drivers, we test real code components that interact.
- Each integration focuses on a single pair of units, ensuring their functionality before moving to the next pair.
- The total number of test sessions remains similar to top-down or bottom-up approaches, but the effort in creating extra test code is significantly reduced.

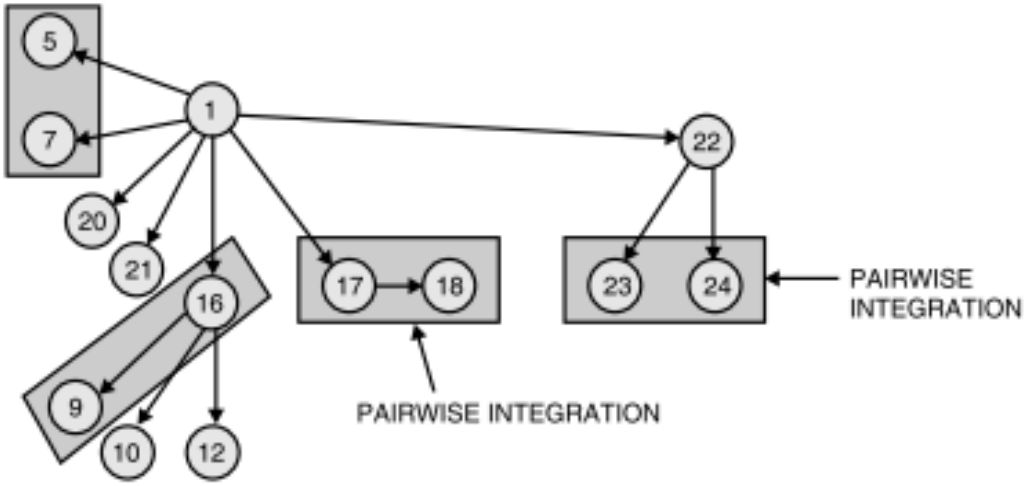


FIGURE 7.7 Pairwise Integration.

Neighborhood Integration

The neighborhood of a node in a graph is the set of nodes that are one edge away from the given node.

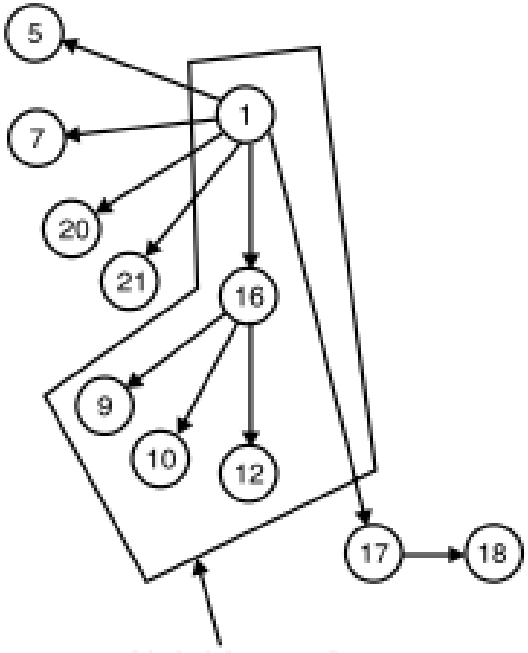


FIGURE 7.8 Neighborhood Integration.

Path-based Integration

innovate achieve lead

- Path-Based Integration Testing is a **structural testing approach** that focuses on testing **all possible execution paths** between interacting modules or components. Instead of only checking individual interfaces, it ensures that different **execution flows and conditions are**
- The combination of structural and functional testing.
- structural and functional testing are highly desirable at the unit level and it would be nice to have a similar capability for integration and system testing.
- "*Instead of testing interfaces among separately developed and tested units, we focus on interactions among these units.*"
- Here, *cofunctioning might be a* good term.
- Interfaces are structural whereas interaction is behavioral.

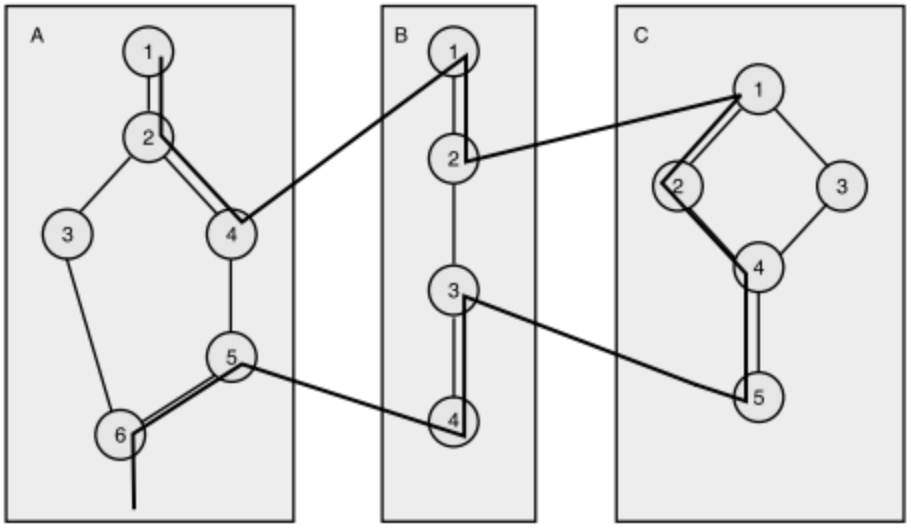


FIGURE 7.9 MM-Path Across Three Units (A, B, and C).

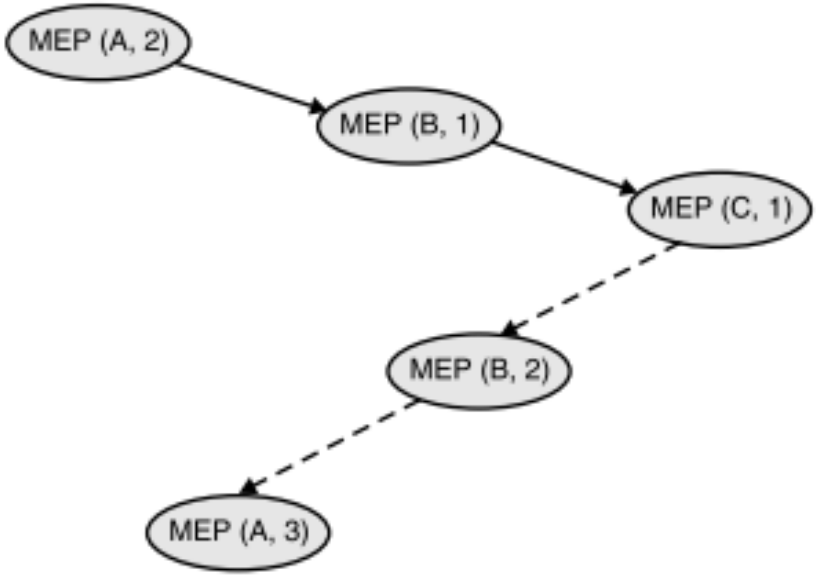


FIGURE 7.10 MM-Path Graph Derived from Figure 7.9.

SYSTEM TESTING

The testing that is conducted on the complete integrated products and solutions to evaluate system compliance with specified requirements on functional and non functional aspects is called *system testing. It is done after* unit, and integration testing phases.

System testing is done to:

1. Provide independent perspective in testing as the team becomes more quality centric.
2. Bring in customer perspective in testing.
3. Provide a "fresh pair of eyes" to discover defects not found earlier by testing.
4. Test product behavior in a holistic, complete, and realistic environment.
5. Test both functional and non functional aspects of the product.
6. Build confidence in the product.
7. Analyze and reduce the risk of releasing the product.
8. Ensure all requirements are met and ready the product for acceptance testing.

Thank You