BITS Pilani presentation

Dr. Nagesh BS

BITS Pilani
Pilani Campus

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG501
Software Quality Assurance and Testing
Lecture No. 4

Deep driving SQA: Software Testing Techniques

Software testing (or “testing”) was the first software quality assurance tool applied to control the software product’s quality before its shipment or installation at the customer’s premises.

SQA professionals were encouraged to extend testing to the partial in-process products of coding, which led to software module (unit) testing and integration testing.

Definition

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

Software testing is a formal process carried out by a specialized testing team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer.

All the associated tests are performed according to approved test procedures on approved test cases.

Formal – Software test plans are part of the project’s development and quality plans, scheduled in advance and often a central item in the development agreement signed between the customer and the developer.

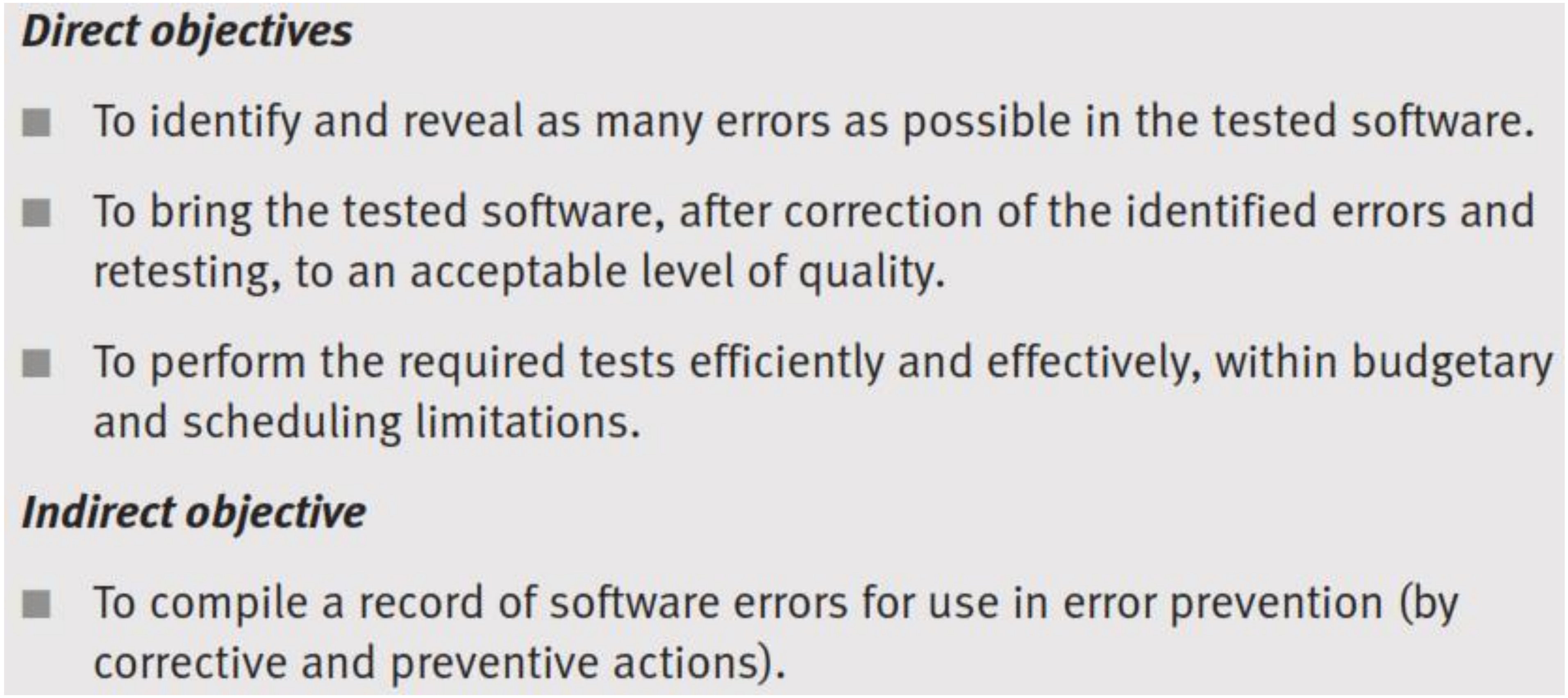
Specialized testing team – An independent team or external consultants who specialize in testing are assigned to perform these tasks mainly in order to eliminate bias and to guarantee effective testing by trained professionals.

Running the programs – Any form of quality assurance activity that does not involve running the software, for example code inspection, cannot be considered as a test.

Approved test procedures **–** The testing process performed according to a test plan and testing procedures that have been approved as conforming to the SQA procedures adopted by the developing organization.

Approved test cases **–** The test cases to be examined are defined in full by the test plan. No omissions or additions are expected to occur during testing.

Software testing objectives



Direct objectives

- To identify and reveal as many errors as possible in the tested software.
- To bring the tested software, after correction of the identified errors and retesting, to an acceptable level of quality.
- To perform the required tests efficiently and effectively, within budgetary and scheduling limitations.

Indirect objective

- To compile a record of software errors for use in error prevention (by corrective and preventive actions).

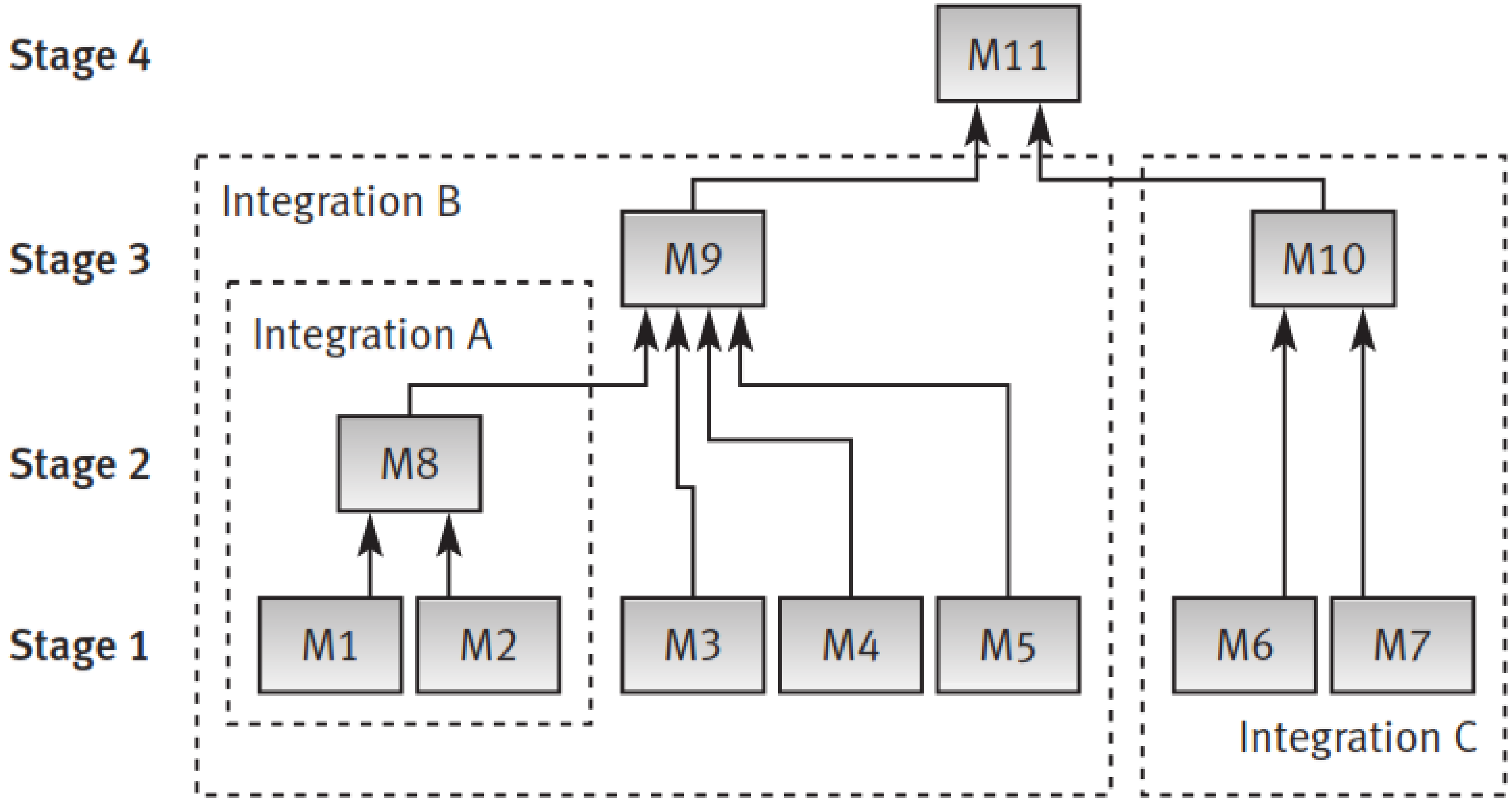
Software testing strategies

To test the software in its entirety, once the completed package is available; known as “big bang testing”.

To test the software piecemeal, in modules, as they are completed (unit tests); then to test groups of tested modules integrated with newly completed modules (integration tests). This process continues until all the Package modules have been tested. Once this phase is completed, the entire package is tested as a whole (system test). This testing strategy is usually termed “incremental testing”.

Incremental testing is also performed according to two basic strategies: bottom-up and top-down.

Both incremental testing strategies assume that the software package is constructed of a hierarchy of software modules.



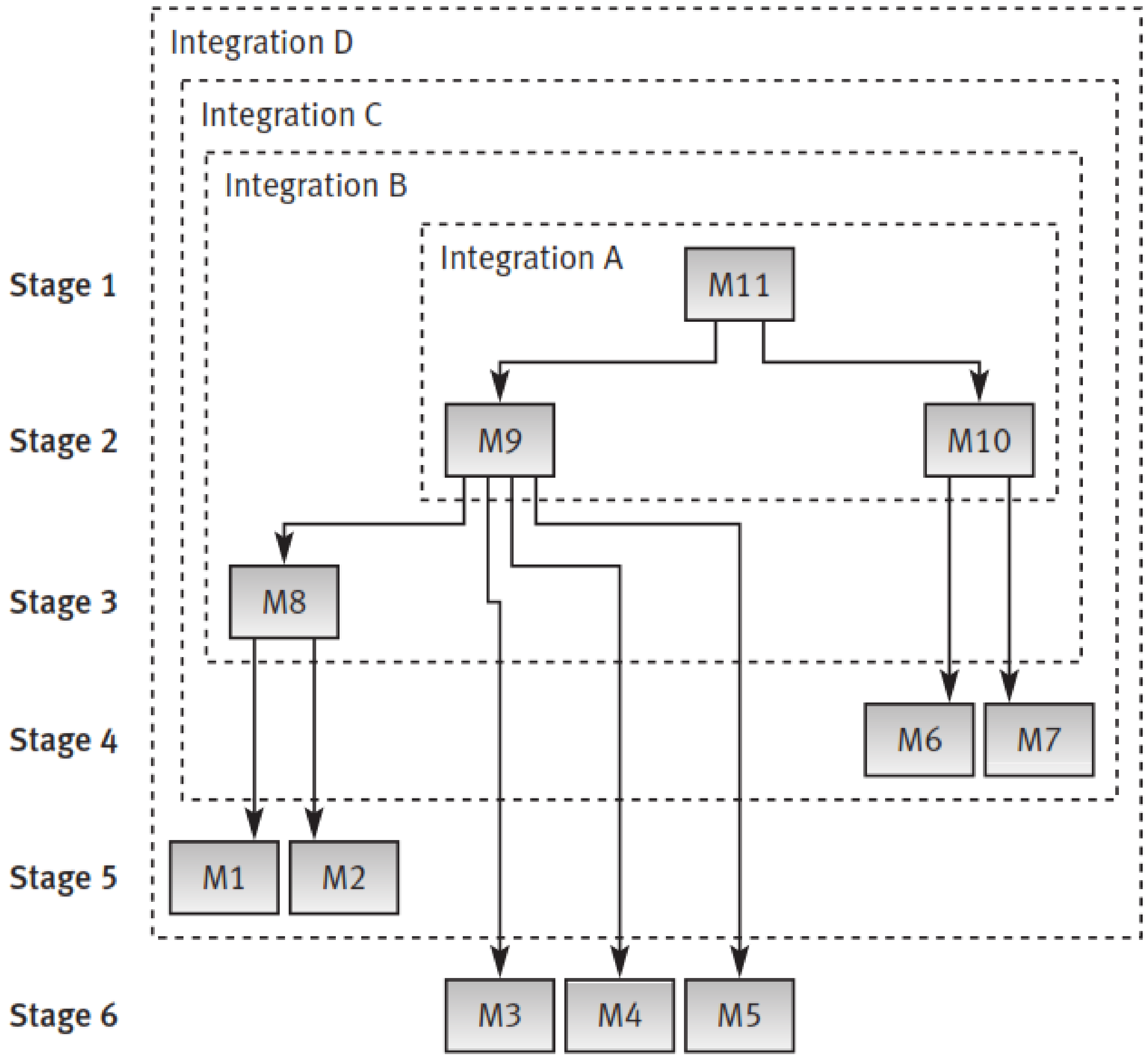
(a) Bottom-up testing

Stage 1: Unit tests of modules 1 to 7.

Stage 2: Integration test A of modules 1 and 2, developed and tested in stage 1, and integrated with module 8, developed in the current stage.

Stage 3: Two separate integration tests, B, on modules 3, 4, 5 and 8, integrated with module 9, and C, for modules 6 and 7, integrated with module 10.

Stage 4: System test is performed after B and C have been integrated with module 11, developed in the current stage.



(b) Top-down testing

Stage 1: Unit tests of module 11.

Stage 2: Integration test A of module 11 integrated with modules 9 and 10, developed in the current stage.

Stage 3: Integration test B of A integrated with module 8, developed in the current stage.

Stage 4: Integration test C of B integrated with modules 6 and 7, developed in the current stage.

Stage 5: Integration test D of C integrated with modules 1 and 2, developed in the current stage.

Stage 6: System test of D integrated with modules 3, 4 and 5, developed in the current stage.

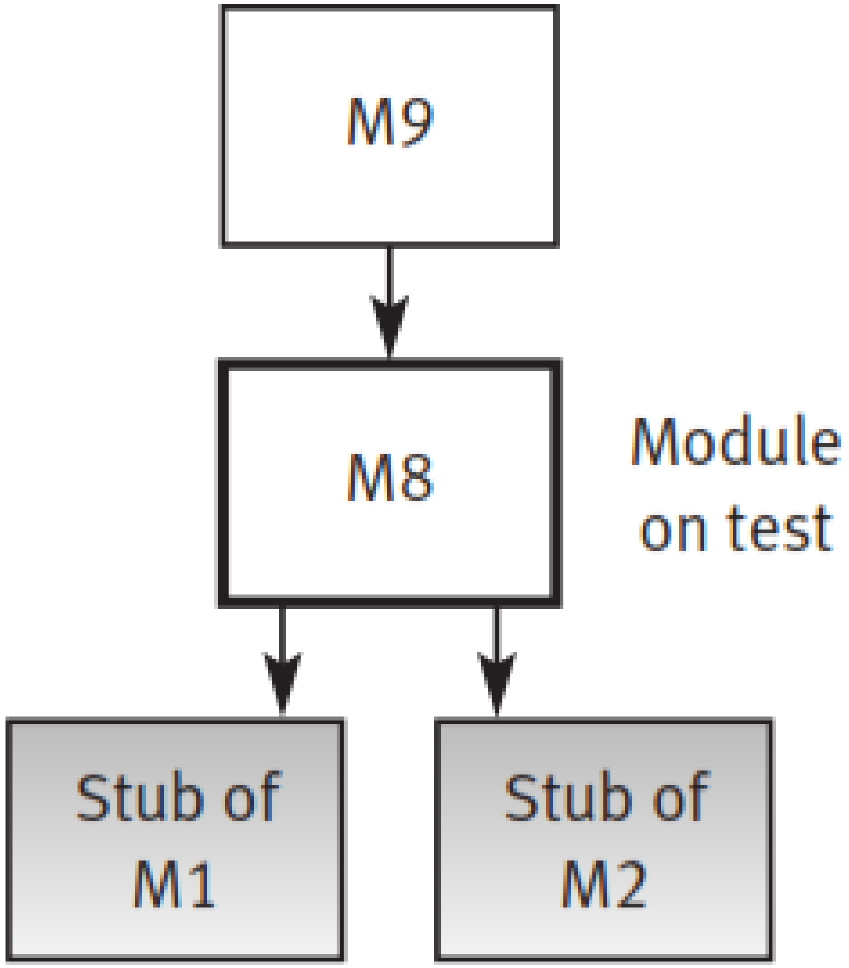
Stubs and drivers for incremental testing

Stubs and drivers are software replacement simulators required for modules not available when performing a unit or an integration test.

A stub (often termed a “dummy module”) replaces an unavailable lower level module, subordinate to the module tested.

Stubs are required for topdown testing of incomplete systems. In this case, the stub provides the results of calculations the subordinate module, yet to be developed (coded), is designed to perform.

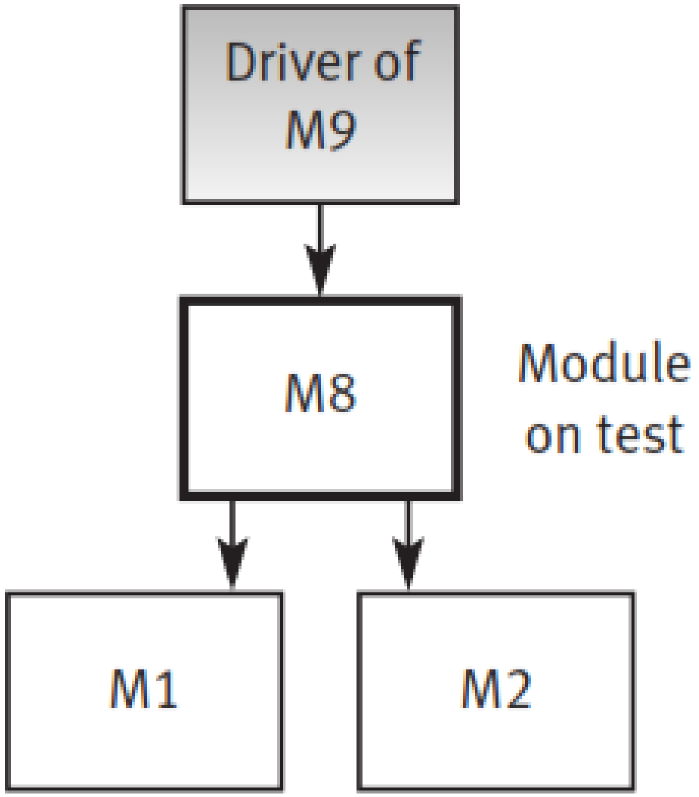
(a) Implementing top-down tests (Stage 3 testing of the example shown in Figure 9.1)



A driver is a substitute module but of the upper level module that activates the module tested. The driver is passing the test data on to the tested module and accepting the results calculated by it.

Drivers are Required in bottom-up testing until the upper level modules are developed (coded).

(b) Implementing bottom-up tests (Stage 2 testing of the example shown in Figure 9.1)



THE TESTING PROCESS

Testing is different from debugging.

- Removing errors from your programs is known as *debugging but testing aims to locate as yet undiscovered errors.*
- Starts from the requirements analysis phase and goes until the last maintenance phase.

Static testing: Requirement analysis and designing stage.

SRS is tested to check whether it is as per user requirements or not. We use techniques of code reviews, code inspections, walkthroughs, and software technical reviews (STRs) to do static testing

Dynamic testing : Starts when the code is ready or even a unit (or module) is ready.

It is dynamic testing as now the code is tested.

Techniques for dynamic testing like black-box, graybox, and white-box testing.

Five distinct levels of testing

a. Debug: It is defined as the successful correction of a failure.

b. Demonstrate: The process of showing that major features work with typical input.

c. Verify: The process of finding as many faults in the application under test (AUT) as possible.

d. Validate: The process of finding as many faults in requirements, design, and AUT.

e. Prevent: To avoid errors in development of requirements, design, and implementation by self-checking techniques, including “test before design.”

Popular equation of software testing

Software Testing = Software Verification + Software Validation

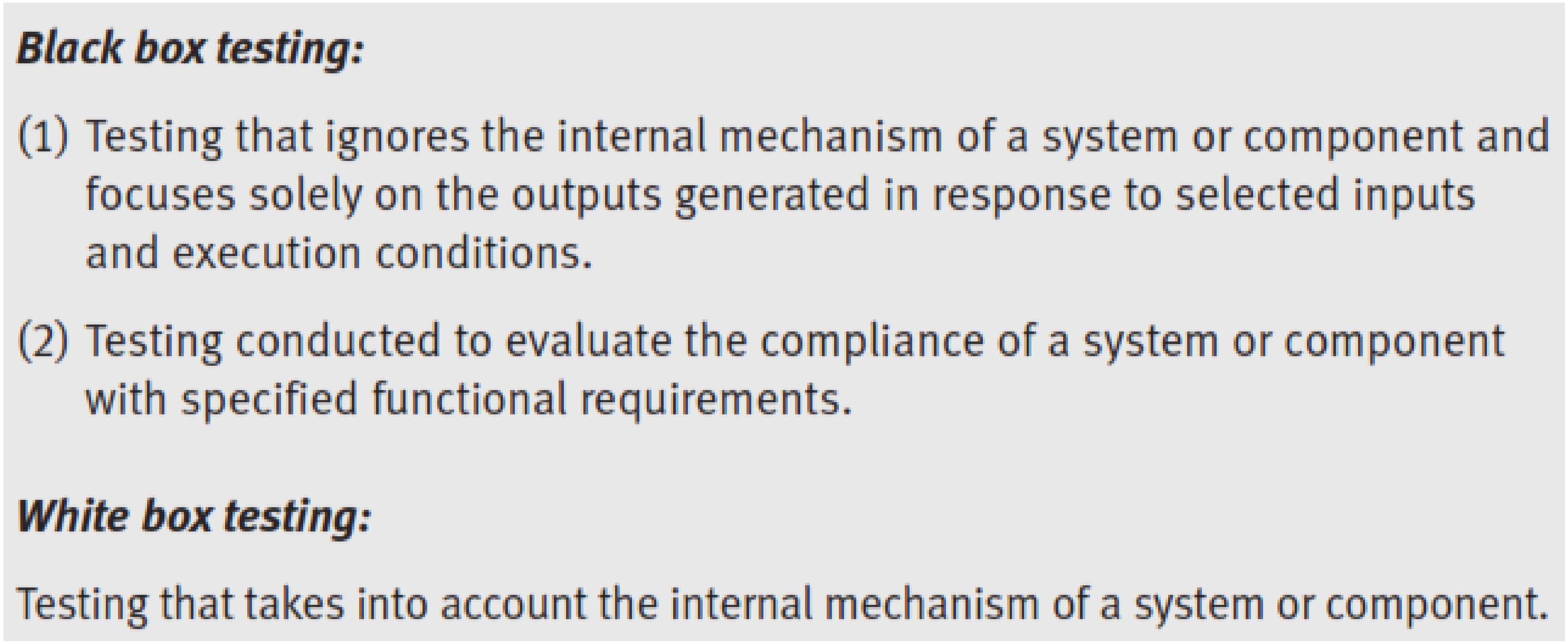
- Software Verification "It is the process of evaluating, reviewing, inspecting and doing desk checks of work products such as requirement specifications, design specifications and code."
- Verification means **Are we building the product right?**

software validation

- "It is defined as the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements. It involves executing the actual software. It is a computer based testing process."
- Validation means **Are we building the right product?**
- Both verification and validation (V&V) are complementary to each other.

Software test classifications

Classification according to testing concept



Black box testing:

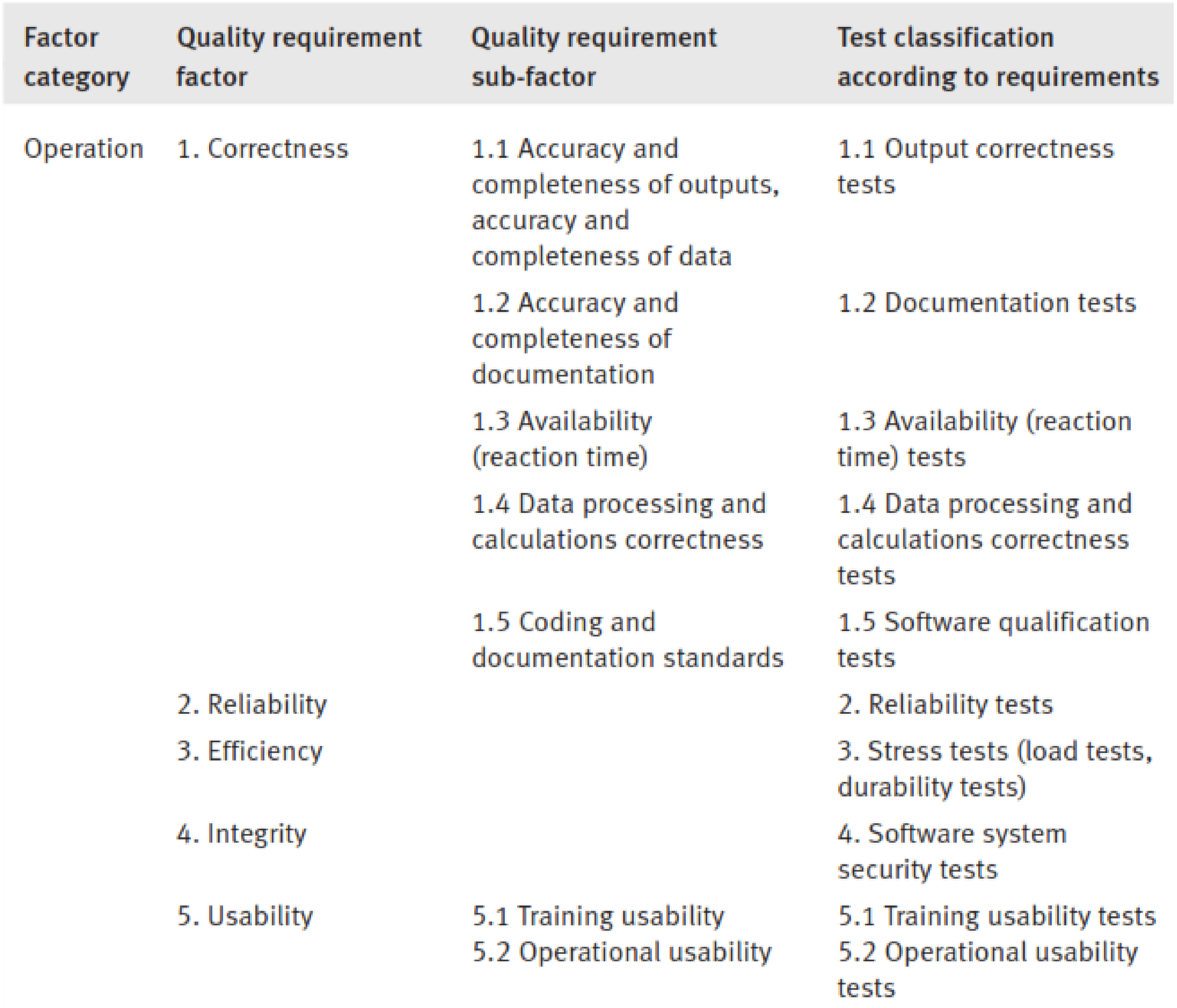
(1) Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

(2) Testing conducted to evaluate the compliance of a system or component with specified functional requirements.

White box testing:

Testing that takes into account the internal mechanism of a system or component.

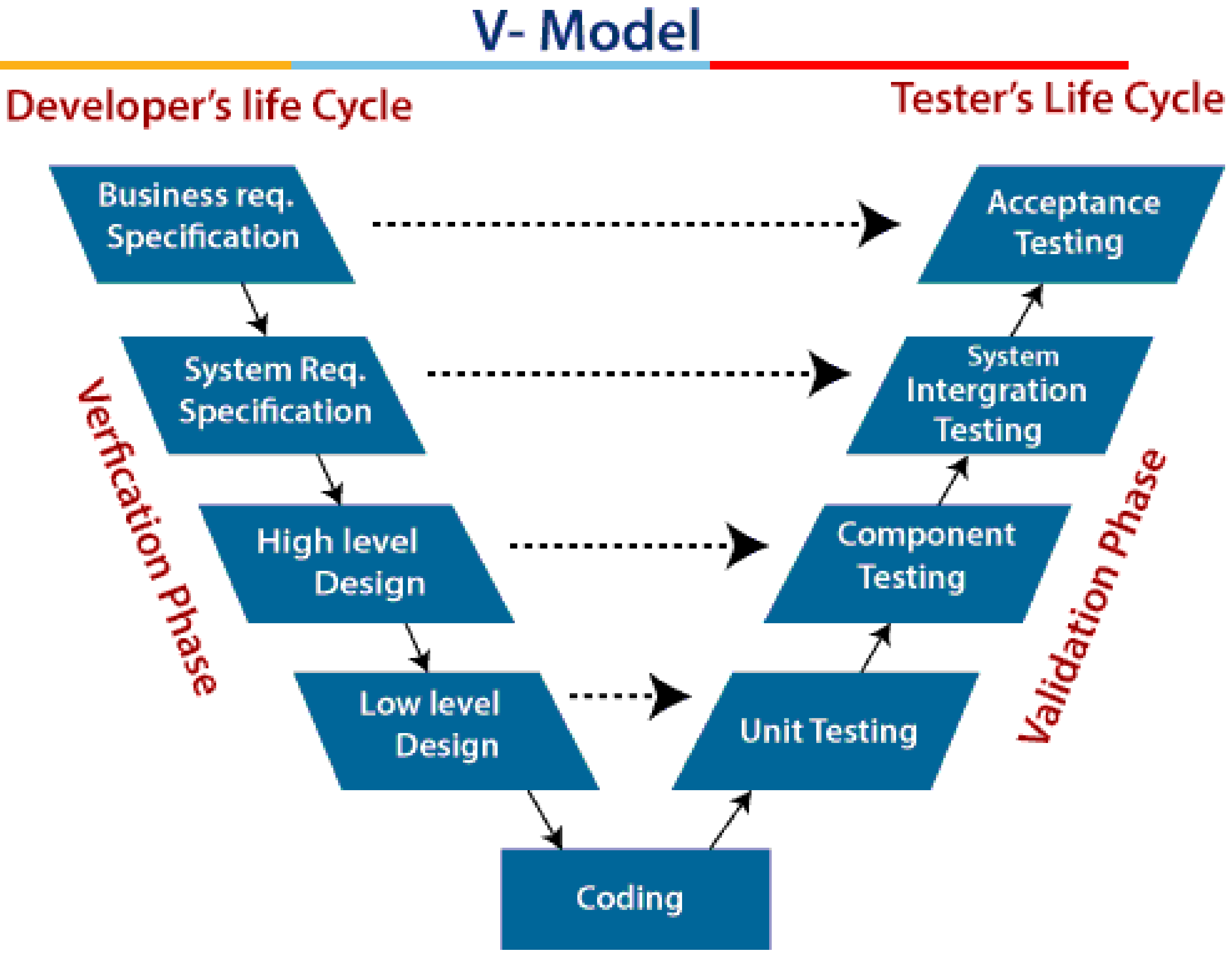
Classification according to requirements



Factor category	Quality requirement factor	Quality requirement sub-factor	Test classification according to requirements
Operation	1. Correctness	1.1 Accuracy and completeness of outputs, accuracy and completeness of data	1.1 Output correctness tests
		1.2 Accuracy and completeness of documentation	1.2 Documentation tests
		1.3 Availability (reaction time)	1.3 Availability (reaction time) tests
		1.4 Data processing and calculations correctness	1.4 Data processing and calculations correctness tests
		1.5 Coding and documentation standards	1.5 Software qualification tests
	2. Reliability		2. Reliability tests
	3. Efficiency		3. Stress tests (load tests, durability tests)
	4. Integrity		4. Software system security tests
	5. Usability	5.1 Training usability 5.2 Operational usability	5.1 Training usability tests 5.2 Operational usability tests

Revision	6. Maintainability		6. Maintainability tests
	7. Flexibility		7. Flexibility tests
	8. Testability		8. Testability tests
Transition	9. Portability		9. Portability tests
	10. Reusability		10. Reusability tests
	11. Interoperability	11.1 Interoperability with other software	11.1 Software interoperability tests
		11.2 Interoperability with other equipment	11.2 Equipment interoperability tests

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.

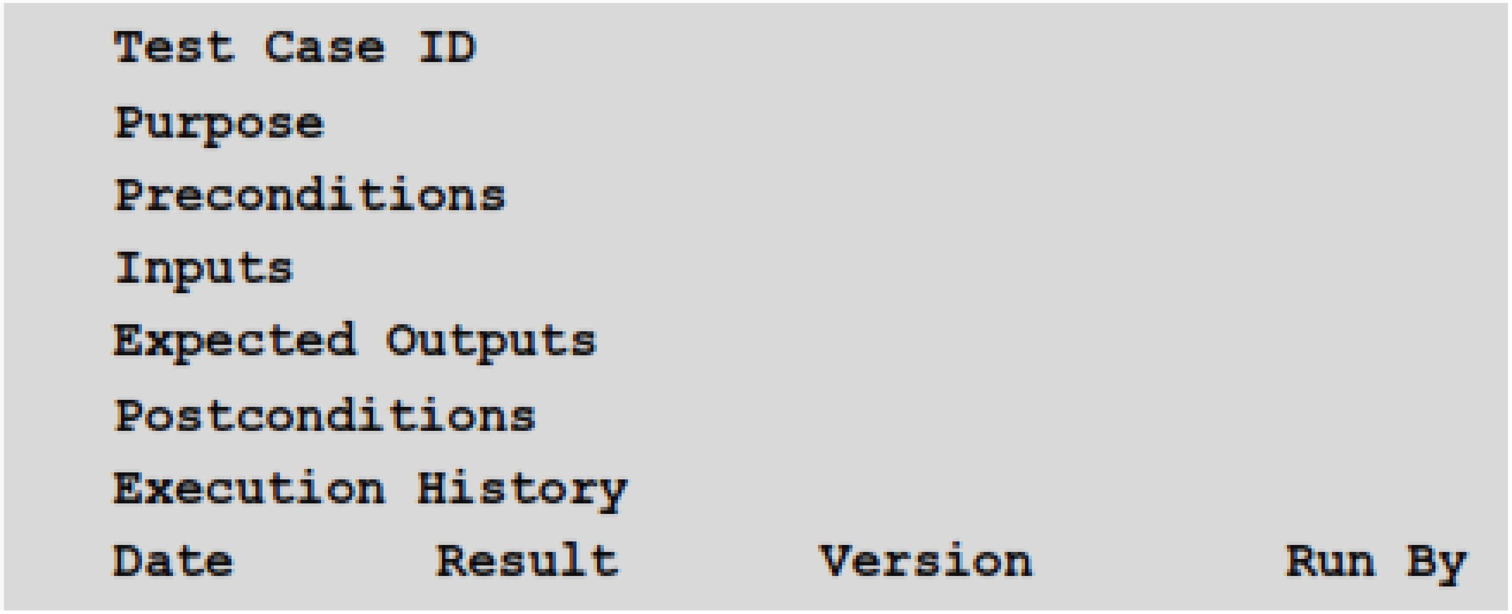
Verification phases on one side and the Validation phases on the other side.

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

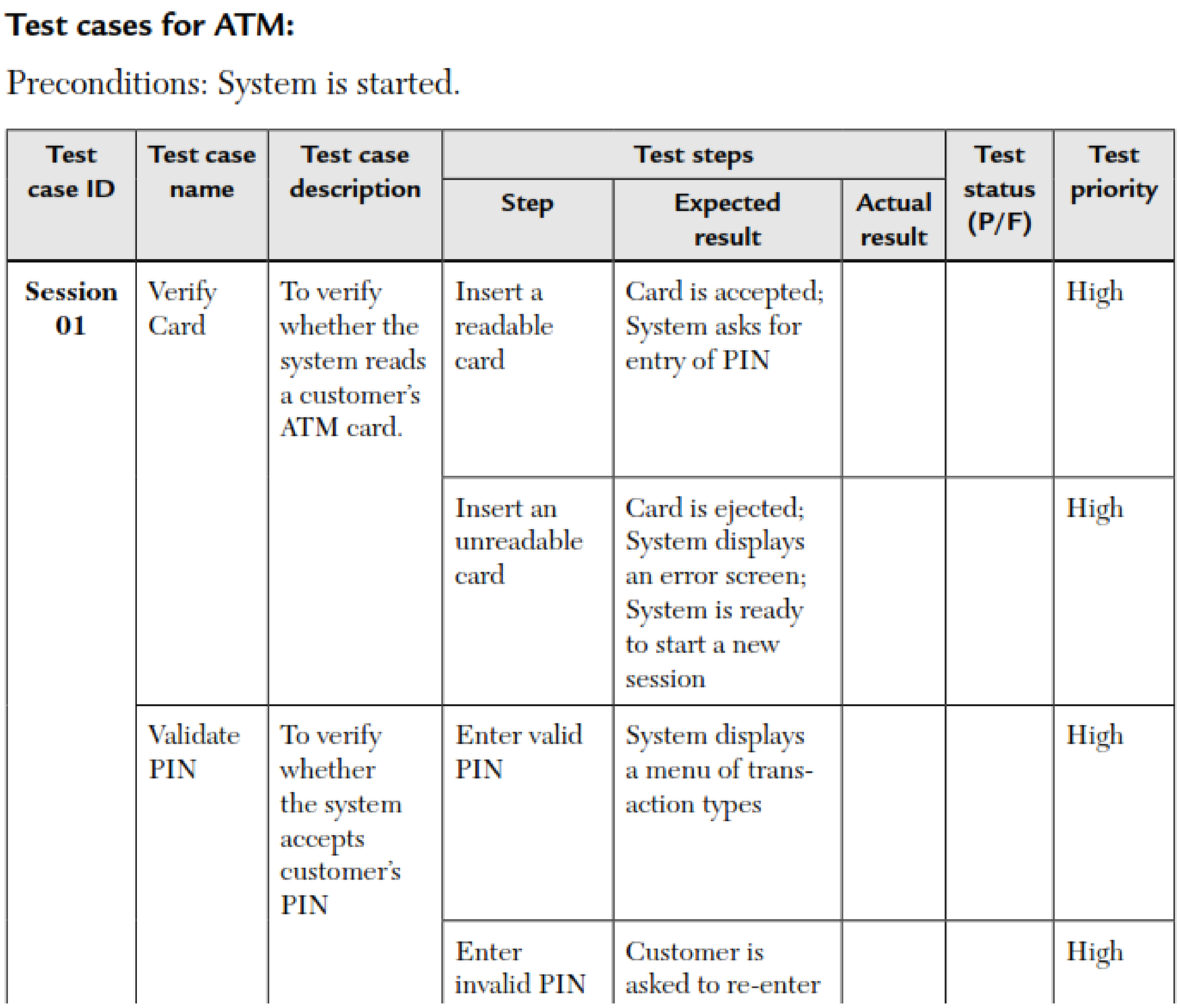


```
Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs
Postconditions
Execution History
Date         Result          Version            Run By
```

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.



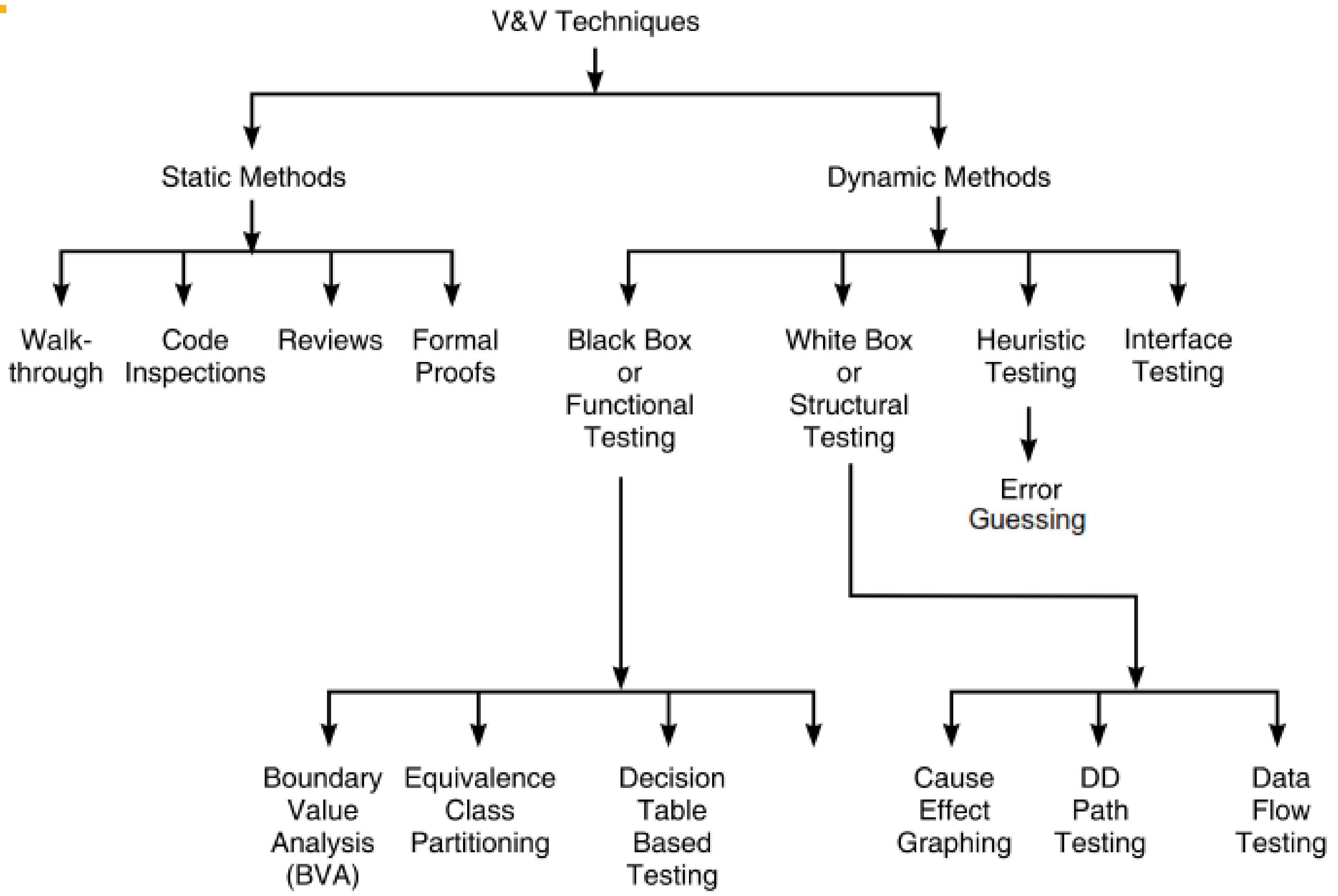
Test cases for ATM:

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.

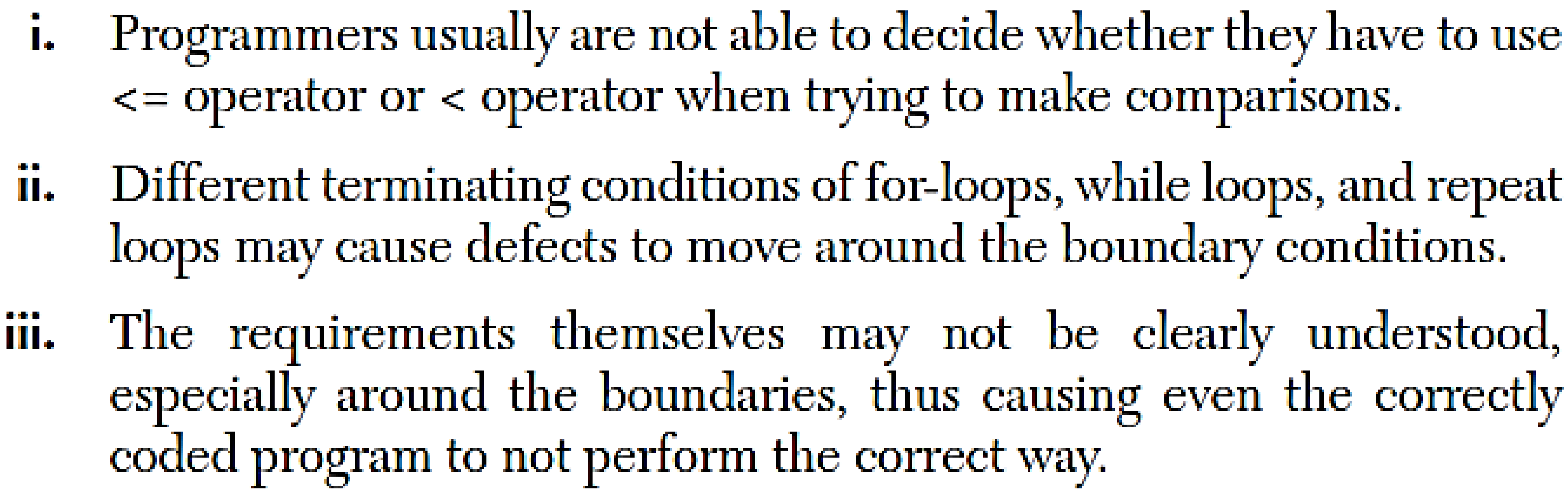
The system or source code is not checked at all.

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 that believes and extends the concept that the density of defect is more towards the boundaries. 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}

- BVA is based upon a critical assumption that is known as single fault assumption theory.
- 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. For n variable to be checked:
    - ***Maximum of 4n+1 test cases***

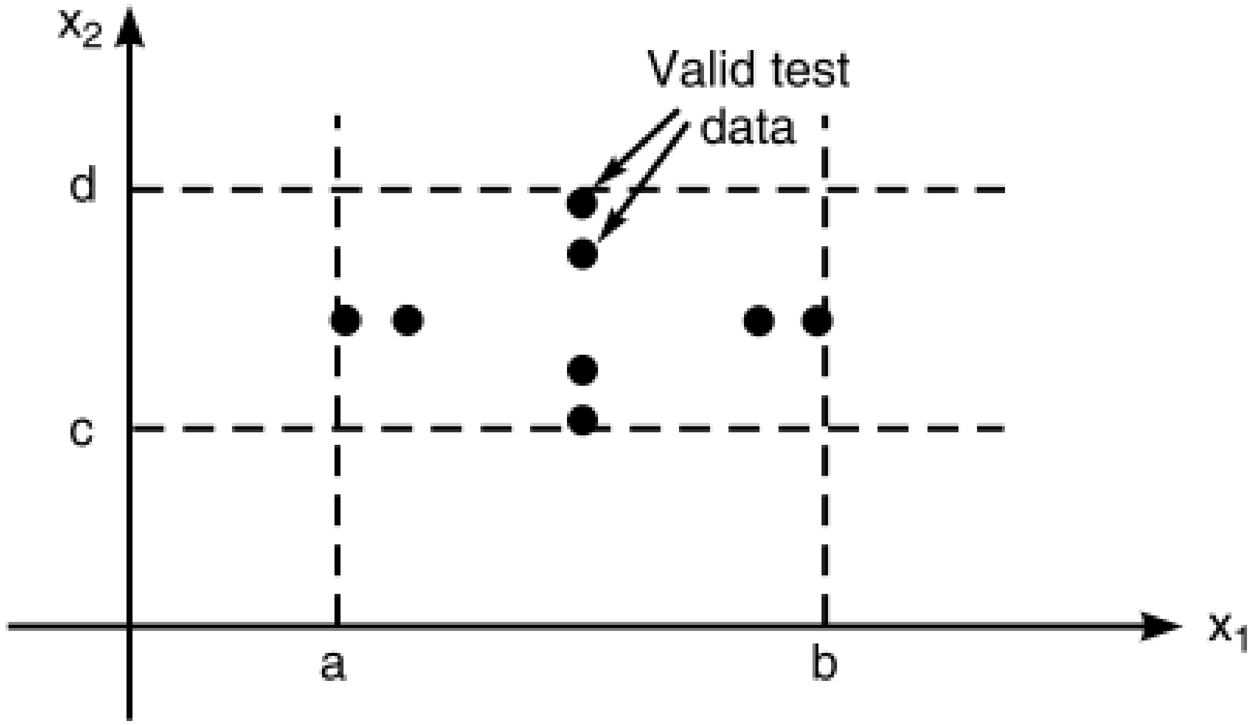


FIGURE 3.1 BVA Test Cases.

Problem: Consider a Program for determining the Previous Date.

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

$1 \leq$ Month≤ 12

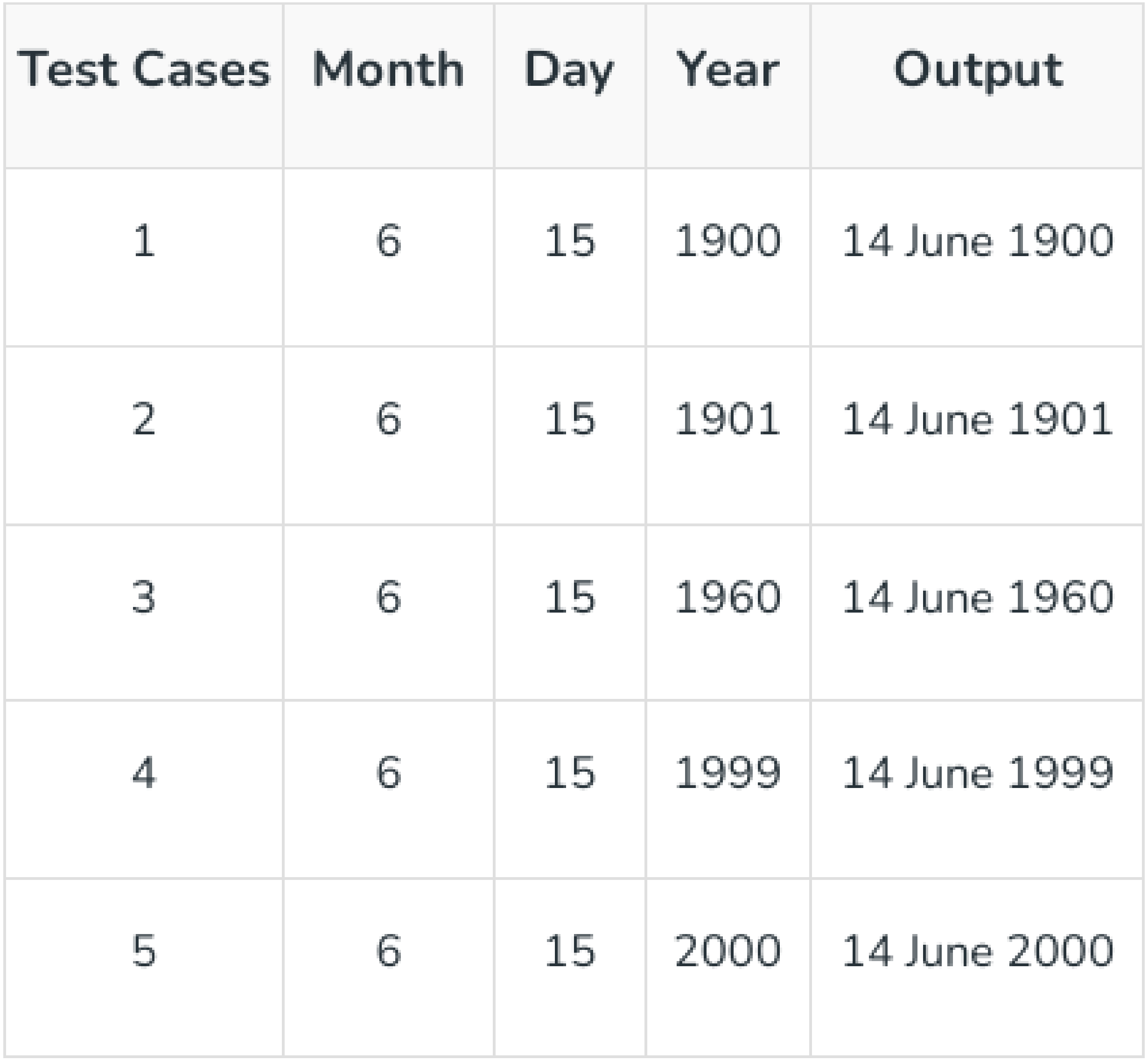
$1 \leq$ Day ≤ 31

$1900 \leq$ Year ≤ 2000

Design Boundary Value Test Cases.

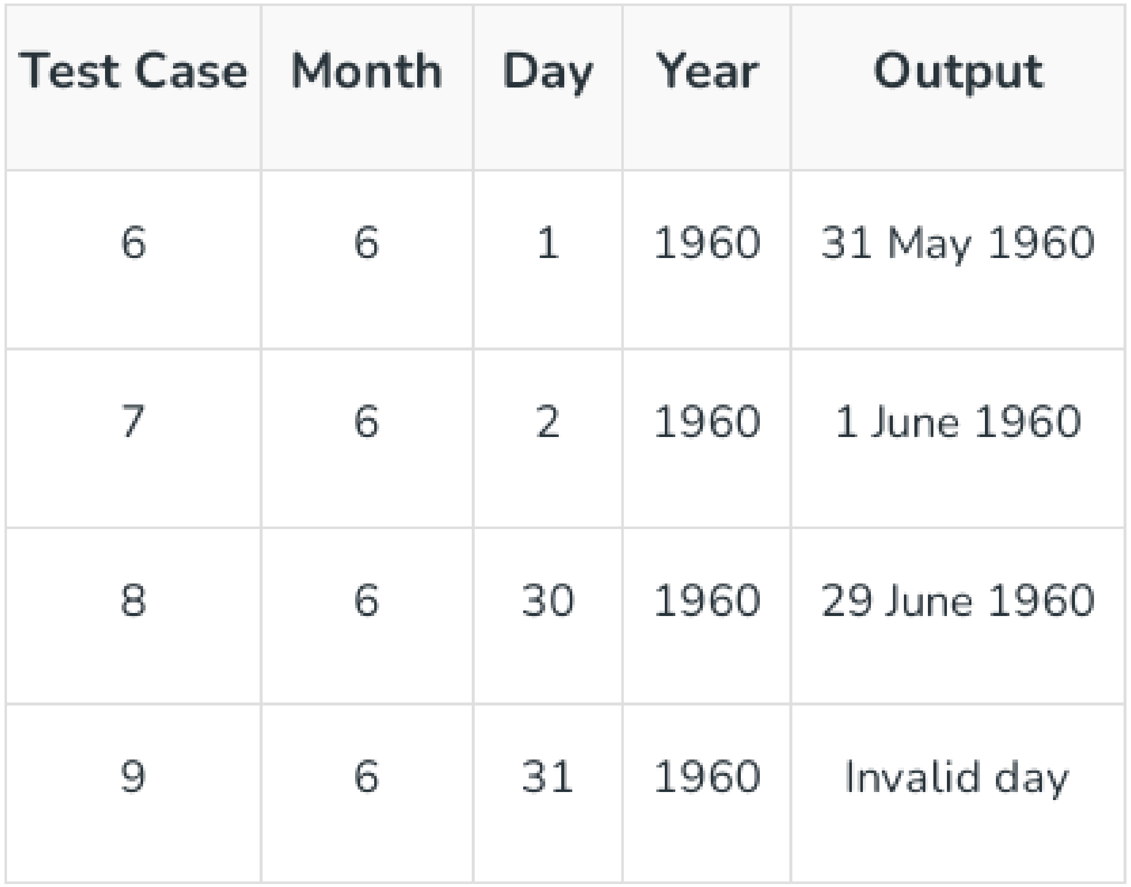
Solution:

1) Year as a Single Fault Assumption



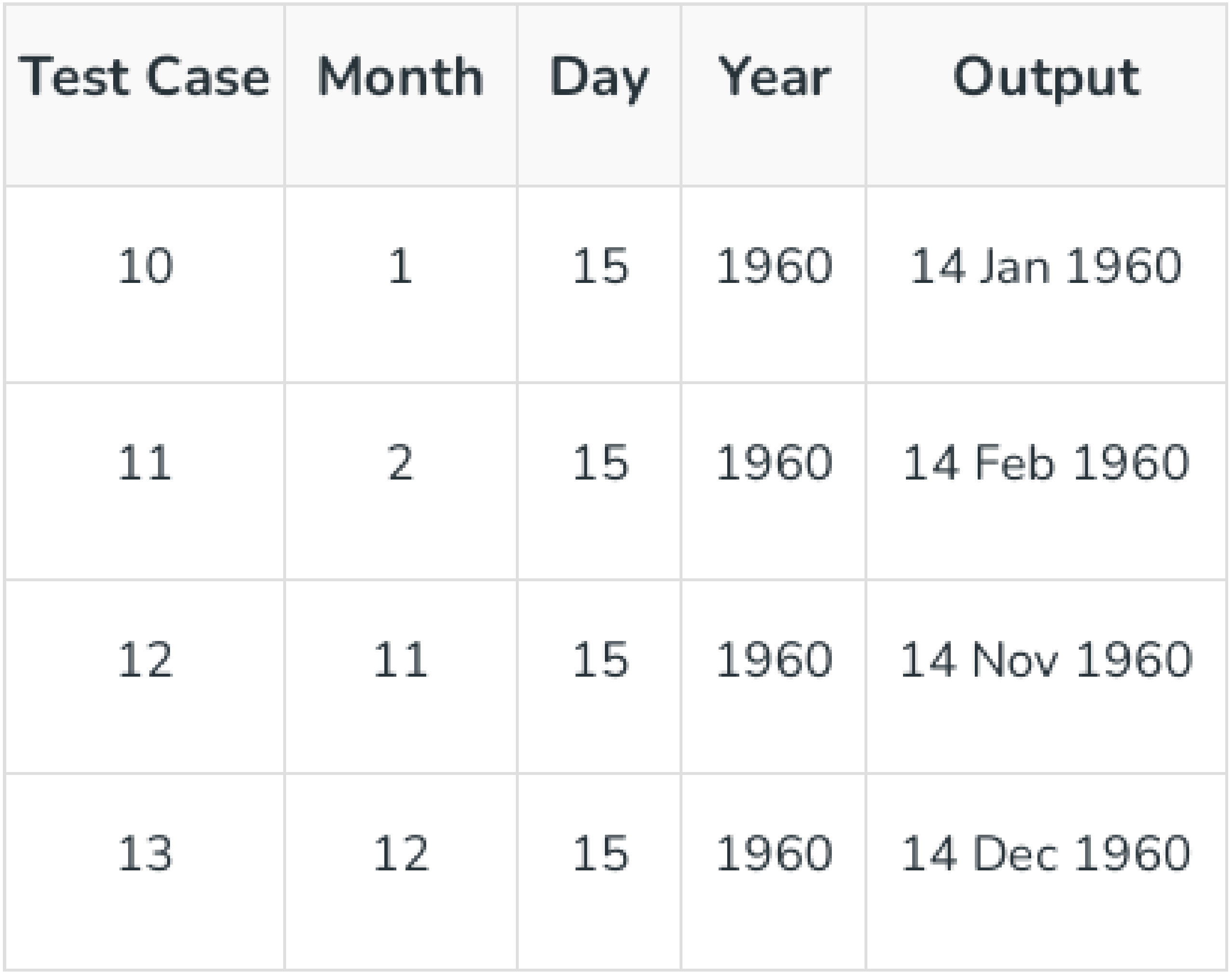
Test Cases	Month	Day	Year	Output
1	6	15	1900	14 June 1900
2	6	15	1901	14 June 1901
3	6	15	1960	14 June 1960
4	6	15	1999	14 June 1999
5	6	15	2000	14 June 2000

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



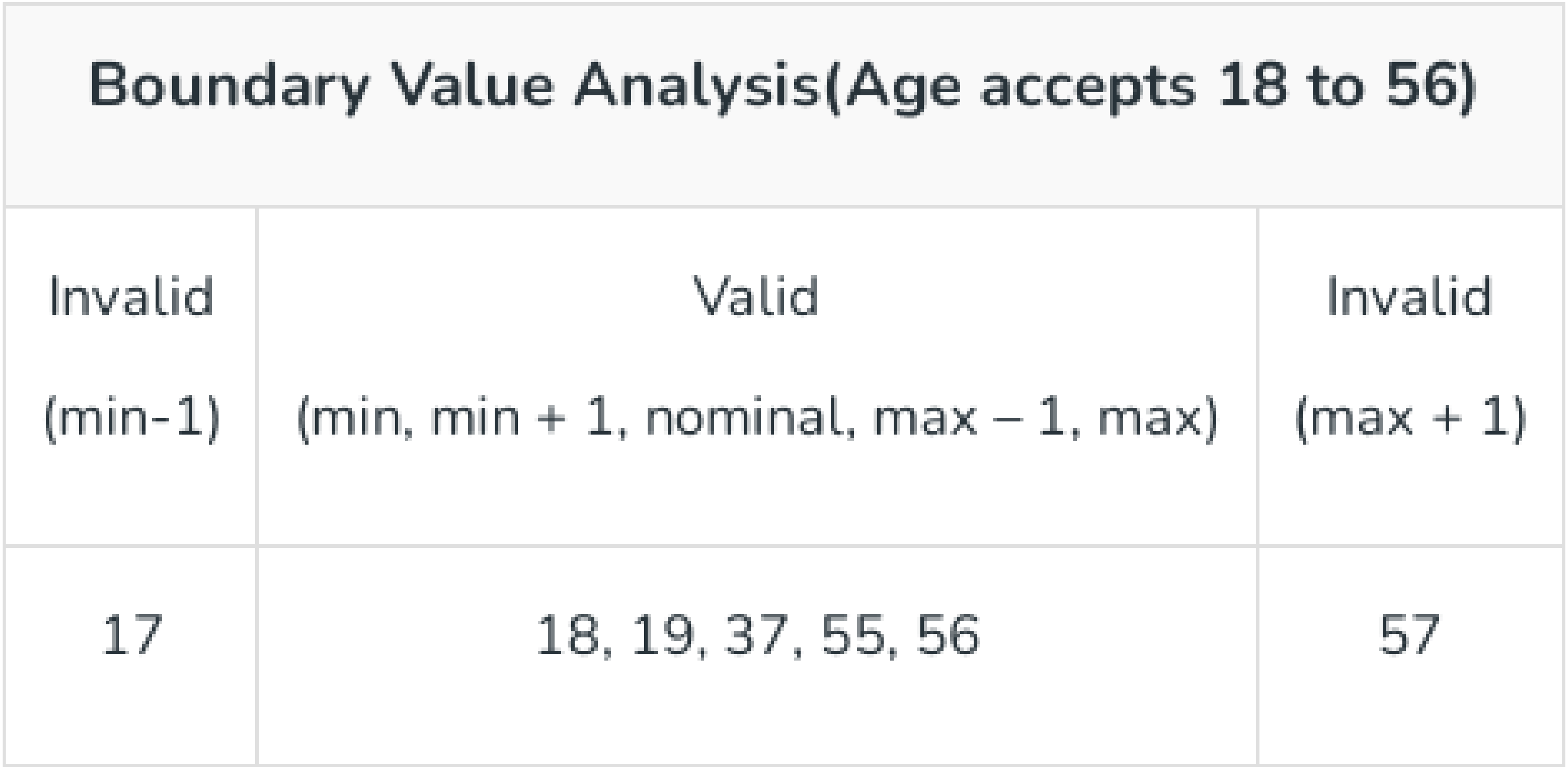
Test Case	Month	Day	Year	Output
10	1	15	1960	14 Jan 1960
11	2	15	1960	14 Feb 1960
12	11	15	1960	14 Nov 1960
13	12	15	1960	14 Dec 1960

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.



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.

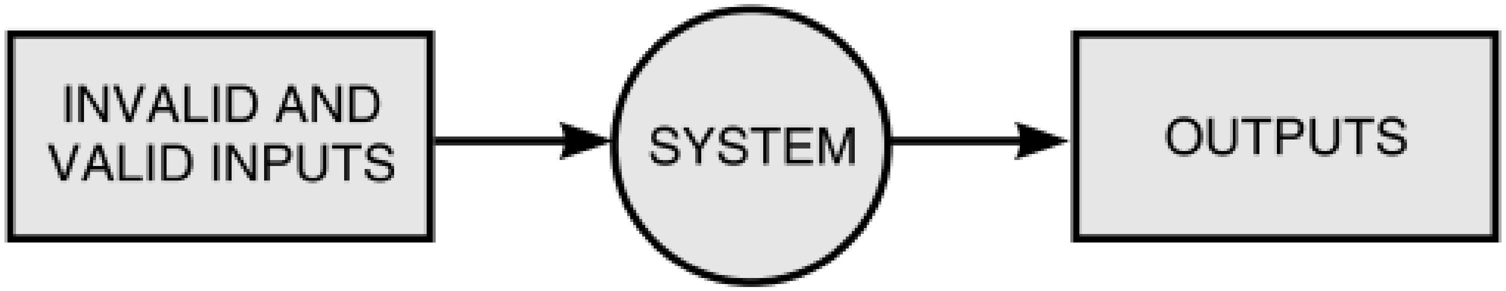


FIGURE 3.4 Equivalence Class Partitioning.

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 he 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 one 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.

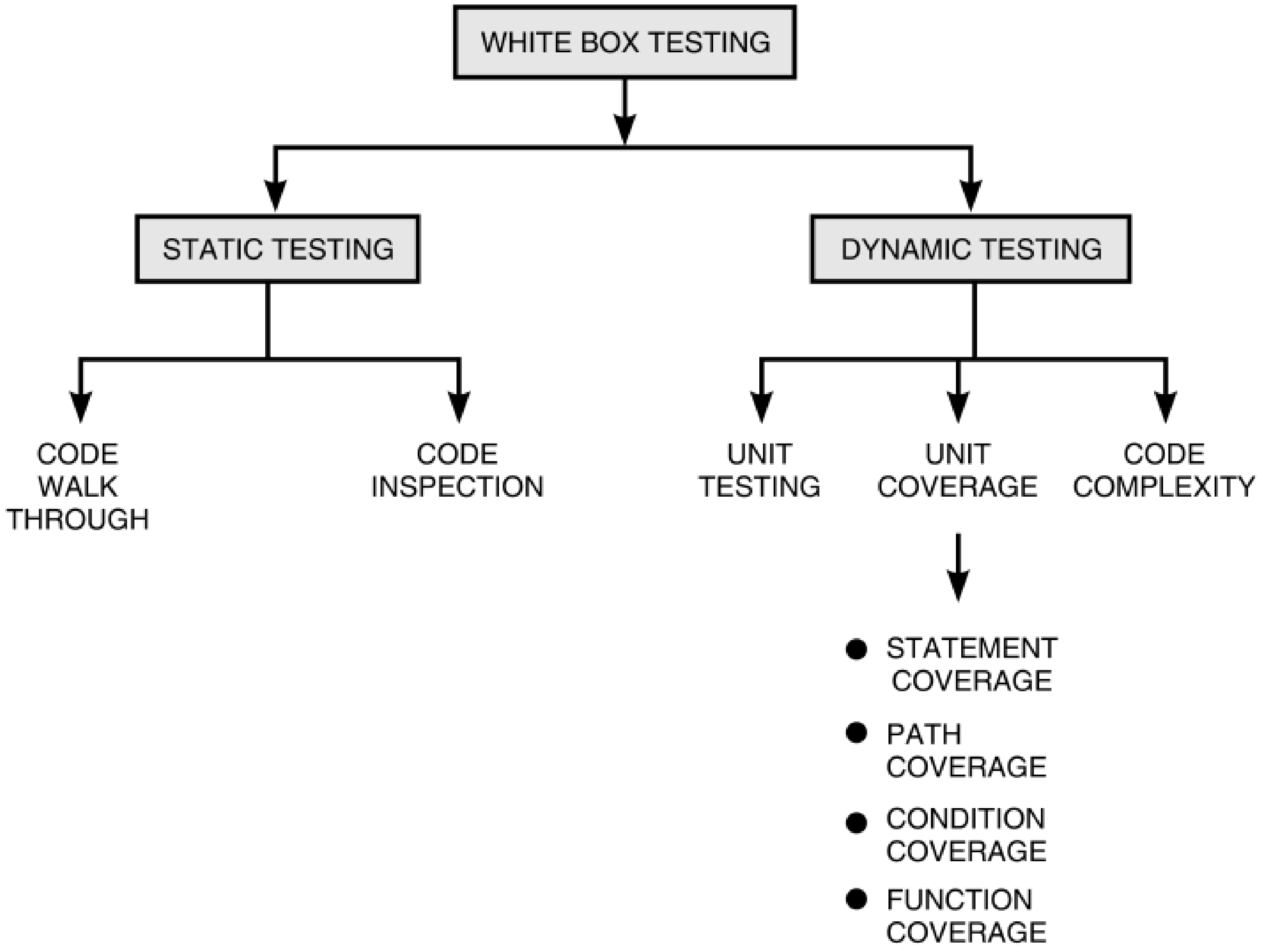


FIGURE 4.1 Classification of White-Box Testing.

CODE COVERAGE TESTING

- 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*

Thank You