EBU6304 – Software Engineering

Implementation and Testing

- Implementation
  - Node, build
  - Mapping design to code
- Testing
  - Theory → Process, model and steps
  - Techniques
    - Black box testing
      - Partition testing
      - Scenario based testing
      - Regression testing
    - White box testing
      - Basis path testing

Implementation

Implementation

- Mapping Design to code
- Code generation should be relatively mechanical
  - Most of the creative work has been carried out in the analysis and design stages.

Purpose of Implementation

- Implement the system in terms of components (source code, scripts, binaries, executables, etc.).
- The system is implemented as a succession of small, manageable steps.
  - The components are tested, and then integrated into one or more executables.
- The system is distributed by mapping executable components onto nodes in the deployment model:
  - primarily based on active classes found during design.

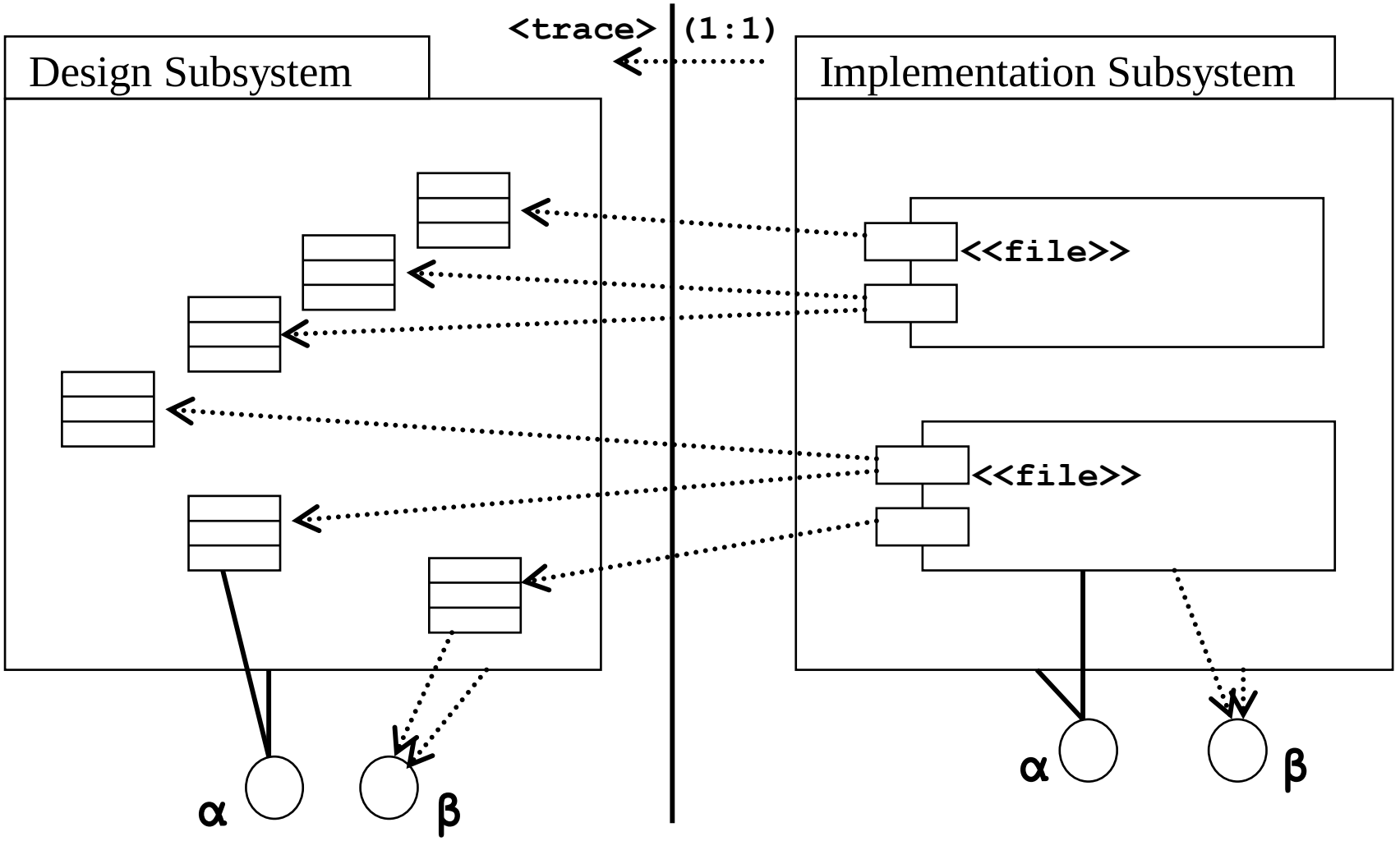
Component

- Physical packaging of model elements, such as design classes.
- Some standard stereotypes of components include the following:
  - **<<executable>>** is a program that may be run on a node
  - **<<file>>** is a file containing source code or data
  - **<<library>>** is a static or dynamic library
  - **<<table>>** is a database table
  - **<<document>>** is a document
- A component traces the design element it implements.

Implementation Subsystem (1/2)

- Implementation subsystems may consist of components, interfaces, and other subsystems.
- It provides interfaces (exportation of operations).
- It depends on the "packaging mechanism" of the implementation environment, such as:
  - A **`package`** in Java.
  - A **`project`** in Visual Basic.
  - A **`directory`** of files in a C++ project.
  - A **`subsystem`** in an integrated development environment such as Rational Apex.

Implementation Subsystem (2/2)



Build

- The software must be built incrementally in manageable steps so that each step yields small integration or test problems.
- The result of each step is called a "build", which is an executable version of the system, normally a part of the system.
- Each build is subject to integration tests before the subsequent build is created; each build has a version control so that it is possible to go back to the previous build.

Integration Build Plan

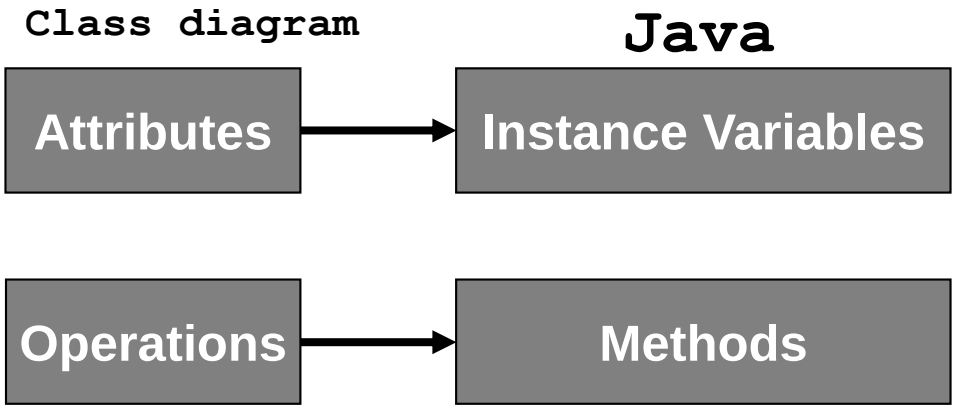
- An integration build plan describes the sequence of builds required in an iteration.
- For each build, the plan describes:
  - The functionality that is expected to be implemented.
  - Which parts of the implementation model are affected by the build, listing the subsystems and components required to implement the functionality expected by the build.

Mapping designs to code

- OOP: Object-Oriented programming
  - Code creation in an OO language
- Implementation in an object-oriented programming language (for example, Java) requires writing source code for:
  - Class definitions
  - Methods' definitions

Class definitions

- Class diagrams provide the class name, attributes, operations.
  - sufficient to create a basic class definition in an OO language.
- It should be straightforward.

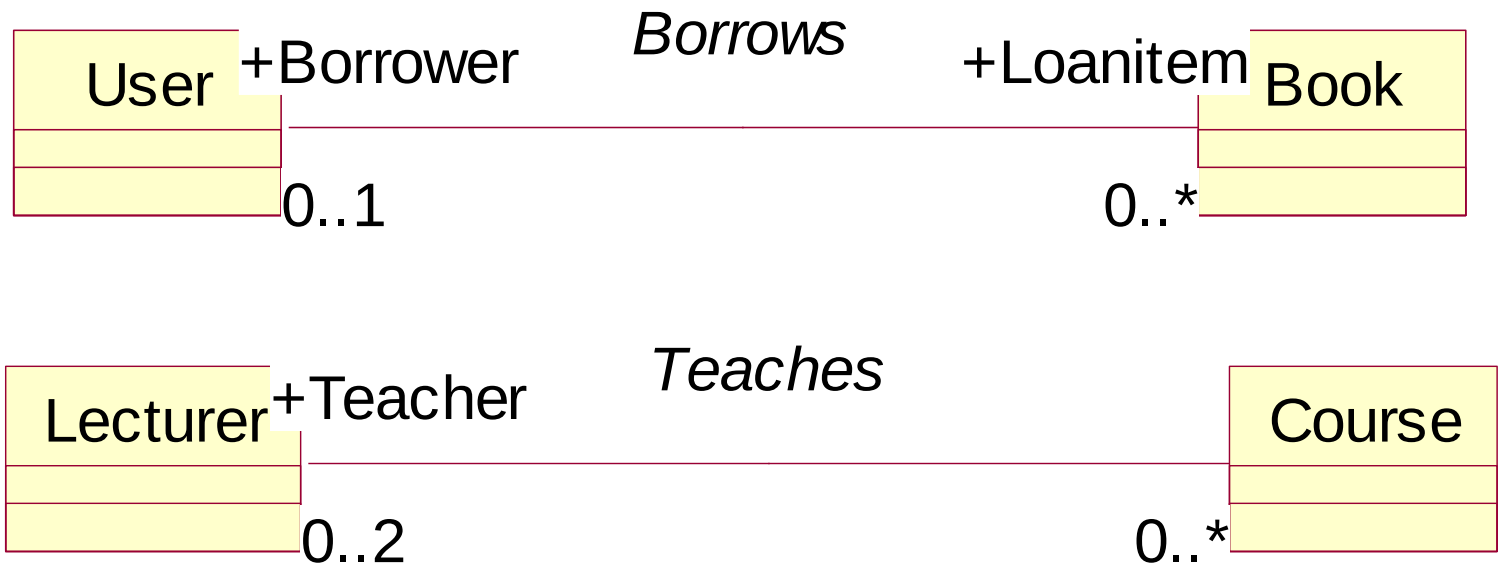


Method definitions

- Interactions between objects:
  - Show the sequence of messages that are sent in response to a method invocation.
  - The sequences of these messages translate to a series of statements in the method definition.
  - Parameters, return type, method decomposition.

Association (1/2)

- An association is a bidirectional semantic connection between classes.
- An association means there is a link between objects.



Association (2/2)

- OO programming languages do NOT provide the concept of association.
- Instead, they provide:
  - References: one object stores a handle to another object.
  - Collections: reference to several objects can be stored and ordered.
  - In Java, associations have to be implemented by an adequate combination of classes, attributes and methods.
- References are unidirectional:
  - Direction
  - Multiplicity

Unidirectional one-to-one

- The simplest association is a unidirectional one-to-one association.
- One direction: *class A to class B*
  - A calls the operations of B, but B never invokes operations of A.
- Map this association to code using a reference from *class A* to *class B*:
  - Add an attribute to *class A* with the type *B*.

```
public class A {
   private B b;
   // ...
}
```

```
public class B {
   // ...
}
```

Example: Unidirectional one-to-one


```
public class Student {
  private int id;
  private String name;
  private Address address;

  public void changeAddress() { }
}
```

```
public class Address {
  private int houseNo;
  private String street;
  private String postcode;
  // ...
}
```

Unidirectional one-to-many

- A associates with many B, B associates with one A.
- Unidirectional one to many associations can not be realised using a single reference.
- Collections should be used:
  - Ordered or not ordered
  - **Array**, **ArrayList**, **List**, **Vector**, **LinkedList**, **Tree**, **Set**, …

```
public class A {
  private B[] b;
  // ...
}
```

```
public class A {
  private ArrayList<B> b;
  // ...
}
```

```
public class A {
  private Vector<B> b;
  // ...
}
```

Example: Unidirectional one-to-many


```
public class User {
  private Book[] book;
  // ...
}
```

```
public class Book {
  // ...
}
```

Advanced Mapping

- Inheritance → *extends*
  - *Refer to EBU4201 Java Programming lectures*
- Interface → *implements*
  - *Refer to EBU4201 Java Programming lectures*
- Design patterns
  - Refer to later lectures in teaching week 3 and 4

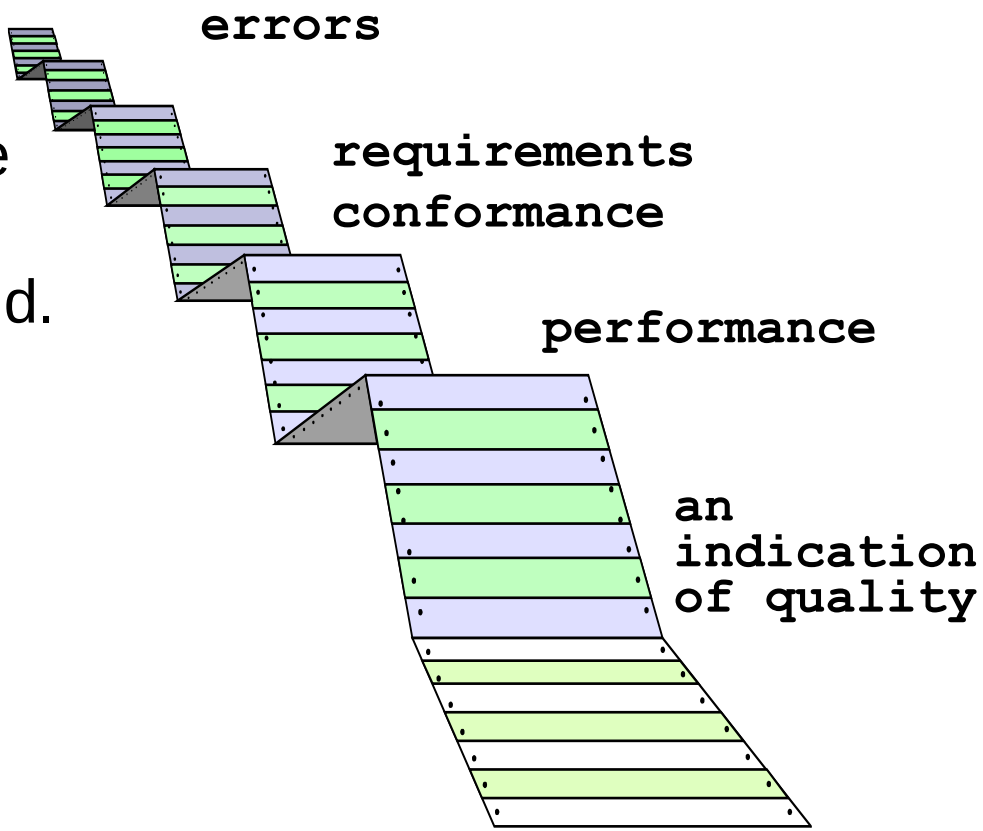
Testing

What is testing?

- To break the system:
  - find errors prior to delivery to the end user.
- Systems are often deployed without being completely tested:
  - Testing is not decidable.
  - Time and budget constraints.
- Testing perhaps is the longest process in software development cycle
  - 40% of the timeline (typical).

What does testing show?

- The aim:
  - Verify the results from the implementation stage by testing each software build.
    - Internal builds
    - Intermediate builds
    - Final customer or external party system software builds



Who tests the software?

- At the component level
  - By the developers who developed the component.
  - Driven by delivery.
- At the integration level
  - By independent testing engineer/team who
    - were not involved with the construction of the system;
    - but have a detailed understanding of the whole system.
  - Attempt to break it.
  - Driven by quality.

Testing goals

Goal 1: Validation testing

- To *demonstrate that the software meets its requirements*: to both developer and customer
  - Custom software: test for every requirement
  - General software: test for all system features
- A successful validation test shows that the system operates as intended.

Testing goals

Goal 2: Defect testing

- To *discover defects*
  - Failure: any deviation of the observed behaviour from the specified behaviour
  - Error: system is in a state such that further processing will lead to a failure
  - Defect: mechanical cause of an error
- A successful defect test makes the system perform incorrectly and so exposes a defect in the system.

Overall, the goal of testing is to build confidence: the software is good enough for operational use.

Testing policies

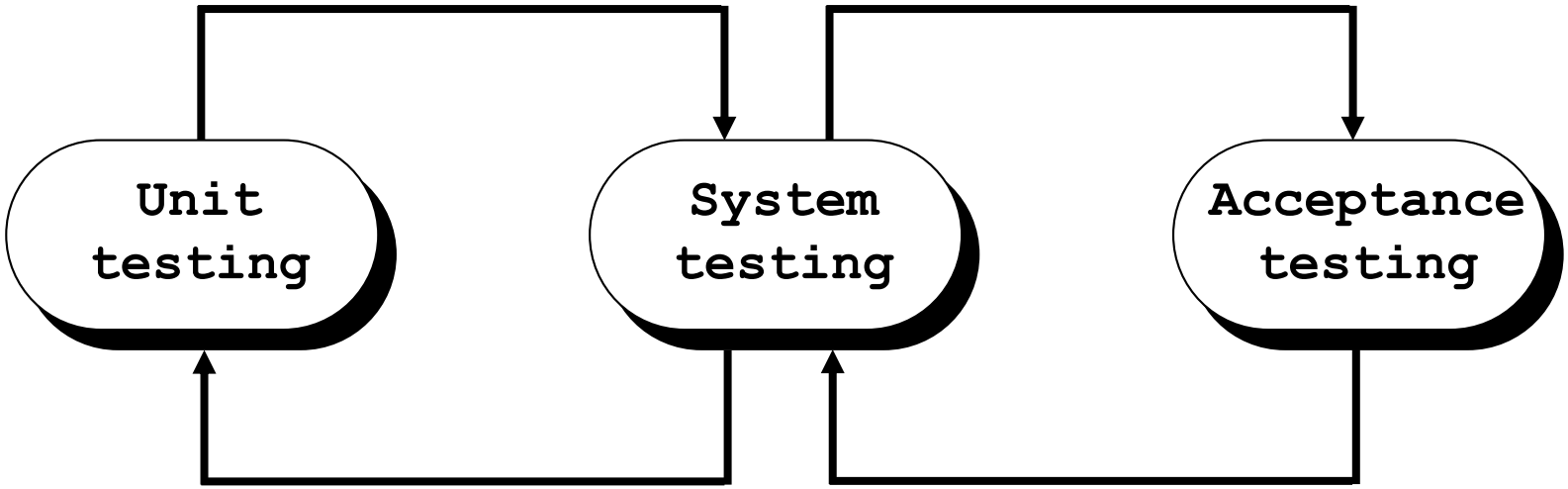
- It is impossible to test every possible program execution sequence.
- Test should be based on a subset of possible test cases.

- Testing policies define the approach to be used in selecting system tests. For example:
  - All functions accessed through menus should be tested.
  - Combinations of functions accessed through the same menu should be tested.
  - Where user input is required, all functions must be tested with correct and incorrect input.

Good Test

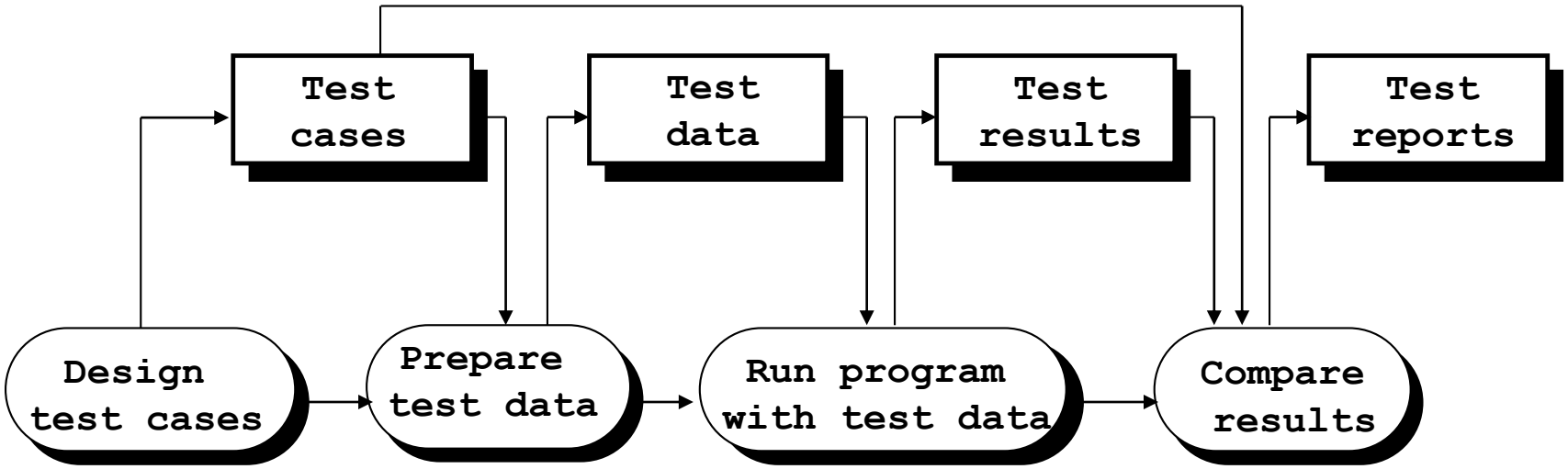
- A good test
  - Has a high probability of finding an error
    - Tester must understand the software and know where the software might fail.
  - Is not redundant
    - Every test should have a different purpose.
  - Should be "best of breed"
    - Due to time and resource limitation, only a subset of tests can be conducted. In such cases, the test that has the highest likelihood of discovering errors should be used.
  - Should be neither too simple nor too complex
    - Tests should be executed separately.

Testing process



- Unit testing (*aka* Component testing) → System components.
- System testing → The whole system.
- Acceptance testing (*aka* Alpha testing) → Customer's data.

The software testing model



- Test case: specifications of the inputs and the expected outputs, plus statements.
- Test data: inputs.
- Outputs can only be predicted by people who understand the system.

Testing Strategy

Testing Strategy

- What tests to run
- How to run them
- When to run them
- How to determine whether the testing effort is successful

Example

- 50% of the tests will be automated.
  - The remainder will be manual.
  - This is to check the user interface is correct.
  - Each subject use case should be tested for its normal flow and behaviour and also two alternative flows.
  - These should ensure that they cover incorrect user input.
- Success criteria – 90% of test cases passed.
- No high priority defects unresolved.

Test Case Design

- Design the test cases
  - The goal of test case design is to create a set of tests that are effective in validation and defect testing.
  - The test cases should have a high likelihood of finding errors.
- Tests must be conducted systematically.
- Test cases must be designed using disciplined techniques.

Test Case Design Example 1

- Example – Test Case for IdentifyAccount with correct input
  - Input
    - A valid account number exists; account number is 99001122 and PIN is 1234.
    - Account number 99001122 has been entered and PIN 1234 has been entered.
    - The system carries out the check.
  - Result
    - The system confirms that 99001122 exists and the associated PIN is correct.
  - Conditions
    - *No conditions exist on this test.*

Test Case Design Example 2

- Example – Test Case for IdentifyAccount with incorrect input
  - Input
    - A valid account number exists; the account number is 99001122 and the PIN is 1234.
    - Account number 99001122 has been entered and PIN 1324 has been entered.
    - The system carries out the check.
  - Result
    - The system confirms that 99001122 exists and the associated PIN is incorrect.
  - Conditions
    - *No conditions exist on this test.*

Test Procedures

- Identifying and structuring Test Procedures:
  - Reuse is important when identifying test procedures.
  - It is vital to reuse existing test procedures but with minimal changes.
  - Often designers will create generic test procedures that can be adapted for more specific test scenarios.

Test Procedures Example 1

- Example – Test Procedure for IdentifyAccount with correct input

1. From the main screen, select the login button. The login screen is displayed.
2. From the login screen, enter the account number, 99001122 and PIN, 1234 in the appropriate boxes and select enter. The system checks the account details and returns “Welcome (name)” and the menus appears below the message.

Test Procedures Example 2

- Example – Test Procedure for IdentifyAccount with incorrect input

1. From the main screen, select the login button. The login screen is displayed.
2. From the login screen, enter the account number, 99001122 and PIN, 1324 in the appropriate boxes and select enter. The system checks the account details and returns “Incorrect PIN, please try again”, and the login screen reappears below the message.

Example – Test Matrix

Test Case: File Open	#Test Description	Test Cases	Samples Pass/ Fail	No. of Bugs	Bug#	Comments
N/A	Create account in system [account number - 99001122 and PIN - 1234]	setup	-	-	-	Added to system for testing purposes
1.1	Test that the account number 99001122 does exist in the system	1.1	P	0	0	Correct behaviour observed
1.2	Test that the account number 99001121 does not exist in the system	1.2	P	0	0	Correct behaviour observed
1.3	Test that the PIN 1324 of account 99001122 is incorrect	1.3	F	1	001	PIN was accepted as a valid account, although the PIN is incorrect

Finding defect

- Example – Defect in IdentifyAccount

Defect number: 001

Defect Title: Invalid PIN given access to banking system

Build number: 100

Test Number: 1.3

Description:

The PIN 1324 was entered into the system and access was given to the account 99001122. This was not expected.

Assigned to Component Engineer : MH

Raised by/assigned to Test Engineer : LM

Testing: Techniques

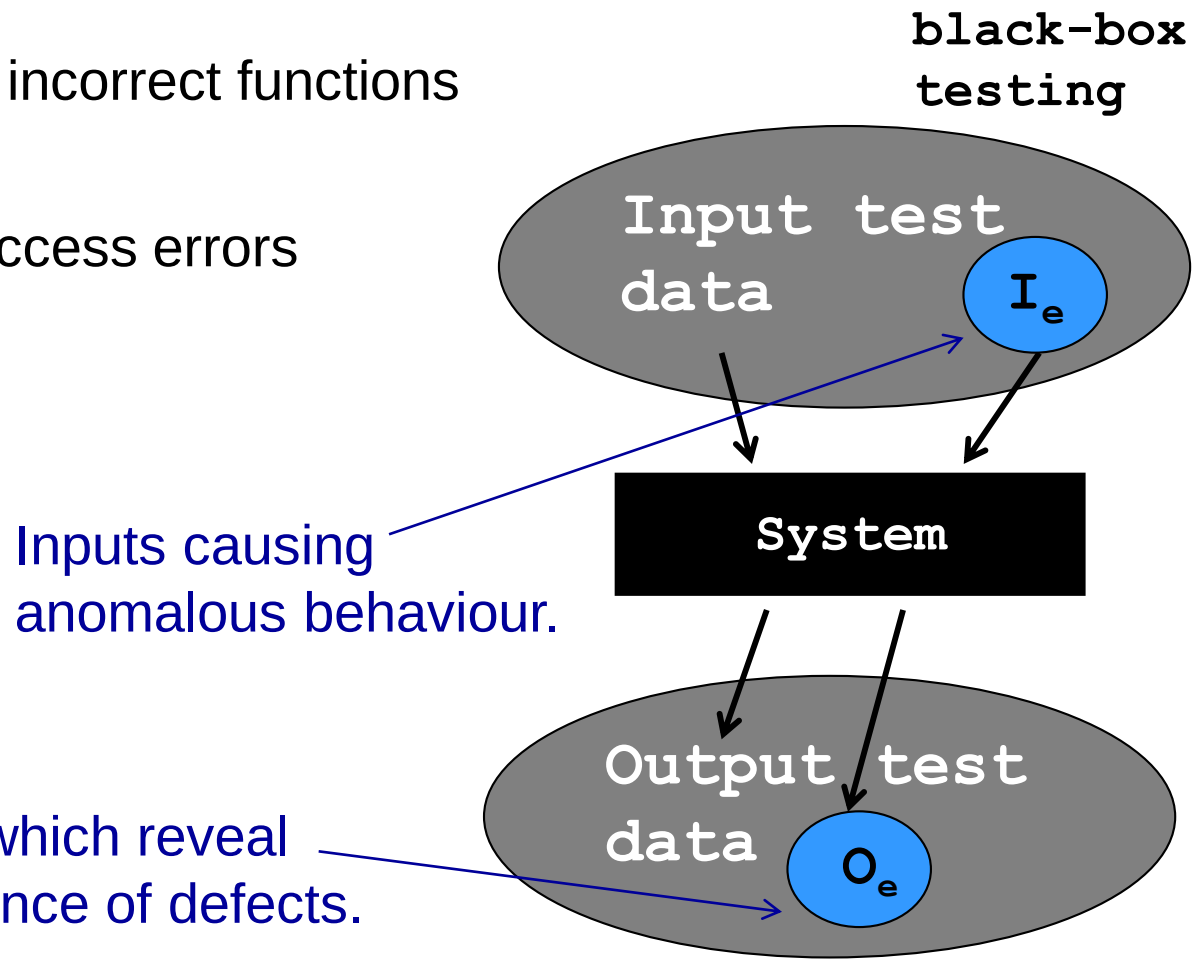
- Test cases must be designed using disciplined techniques:
  - **Black-box testing**
    - Partition testing
    - Scenario-based testing
    - Regression testing
  - **White-box testing**
    - Basis path testing

Black/White-box Testing

- In general, software is tested from two different perspectives:
  - **Black-box testing** → to test software requirements
    - Also called behavioural testing.
    - Focus is on the functional requirements of the software.
  - **White-box testing** → to test the internal program logic
    - Also called *glass-box testing* or *clear-box testing*.
    - Component level test case design.
    - Internal operations are performed according to specifications, and all internal components have been adequately exercised.

Black-box Testing

- Attempt to find errors:
  - Missing functions or incorrect functions
  - Interface errors
  - External database access errors
  - Behaviour errors

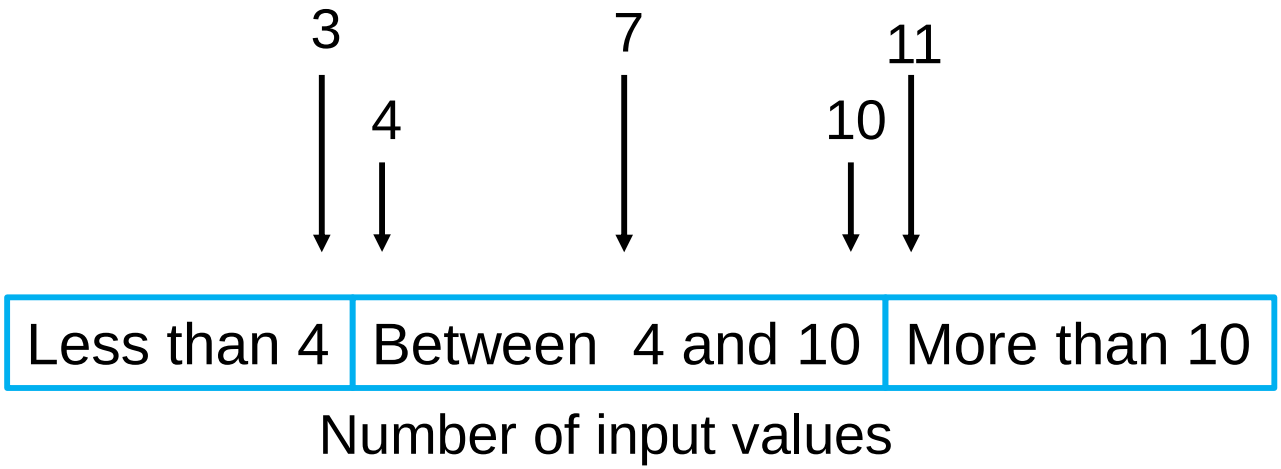


Partition testing

- A typical black-box testing technique
- Input data and output results often fall into different classes where all members of a class are related:
  - Positive numbers, negative numbers, etc
- Each of these classes is an equivalence partition *or* domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.
- Can be used in both component testing and system testing.

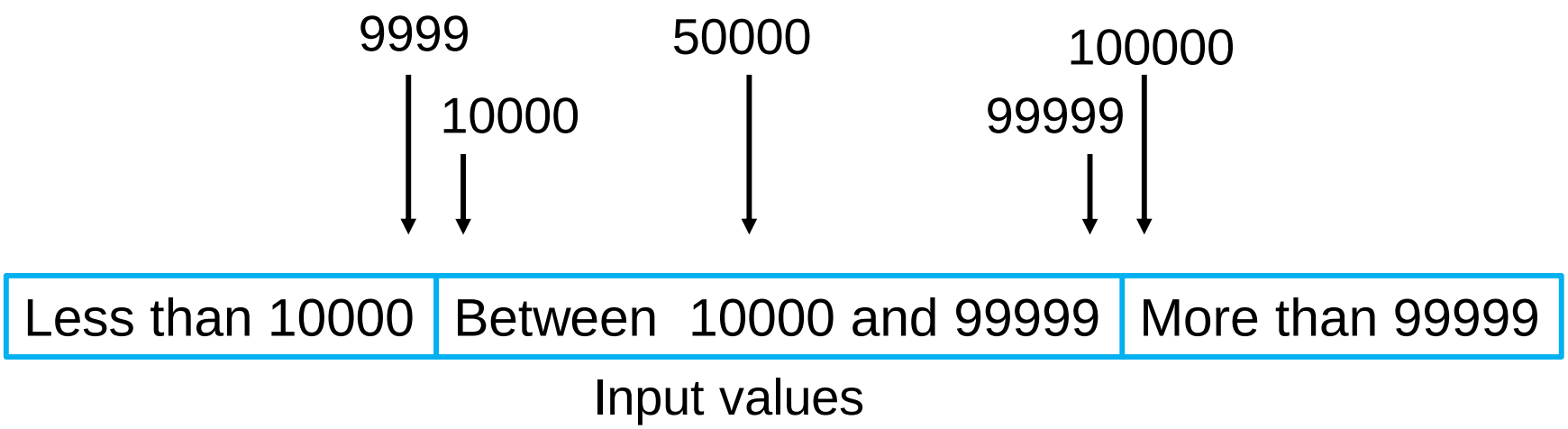
Partition testing: example 1

A program accepts 4-10 inputs.



Partition testing: example 2

A program accepts **inputs that are five-digit integers greater than 10,000.**



Scenario-based testing

- A typical black-box testing technique
- Requirements should be testable:
  - A test can be designed, observer can check that the requirement has been satisfied.
- A validation testing technique where you consider each requirement and derive a set of tests for that requirement.
- Used in system testing.

Scenario-based testing – Example

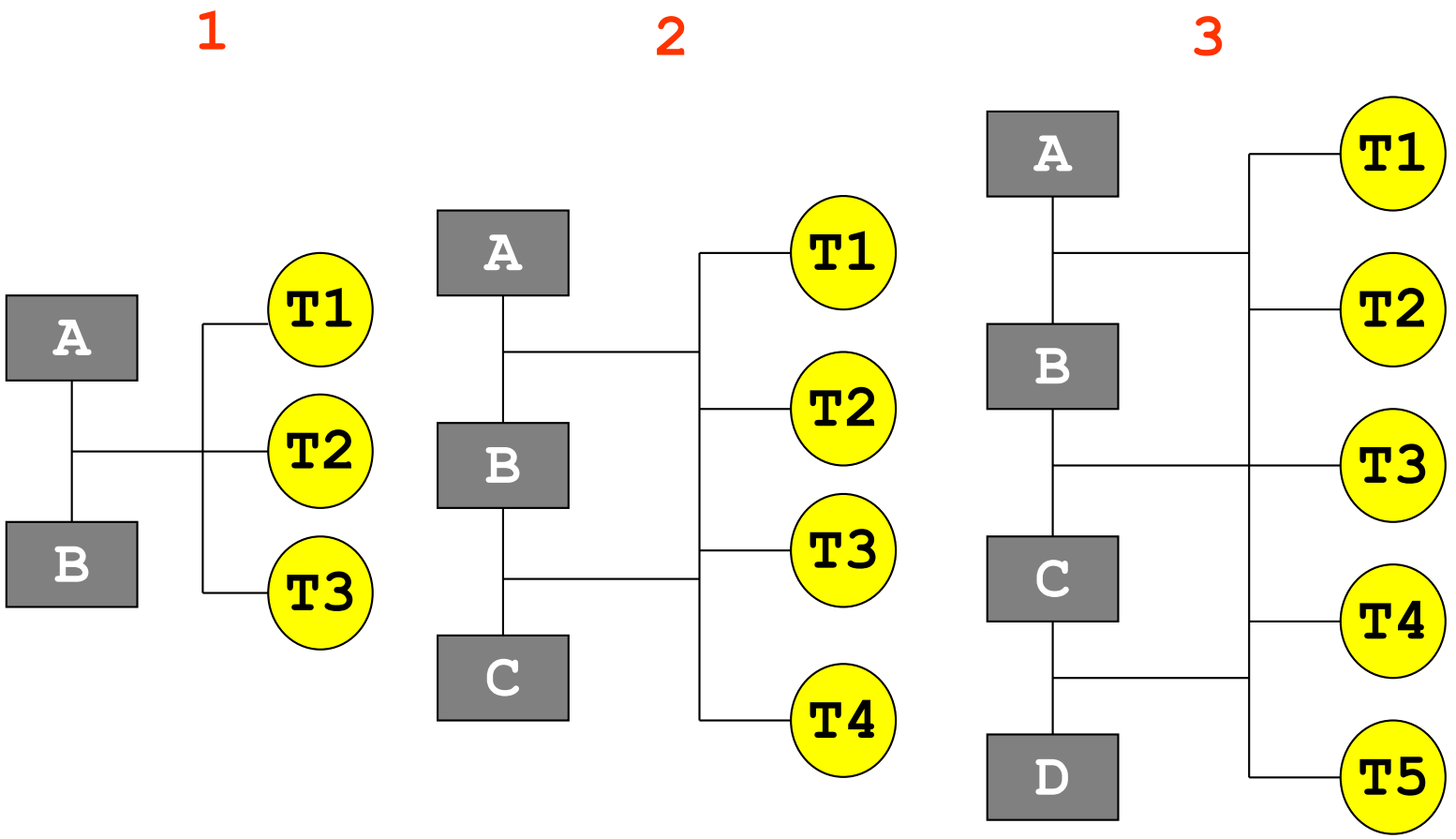
The user shall be able to search either all of the initial set of databases or select a subset from it.

- Initiate user search for searches of items that are known to be present and known not to be present, where the set of databases includes 1 database.
- Initiate user searches for items that are known to be present and known not to be present, where the set of databases includes 2 databases
- Initiate user searches for items that are known to be present and known not to be present where the set of databases includes more than 2 databases.
- …

Regression Testing

- For integration testing.
- At each build stage, test cases will be created to test the functionality of the build.
- As this development is incremental, as new functionality is added to the build, so there are increasing numbers of test cases.
- These test cases will be used at each stage of iteration and they make up what are known as Regression Tests.
- The concept is that you continue to run all of the tests for each build to ensure that earlier functionality is not broken by updates to the build.

Regression Testing (cont.)

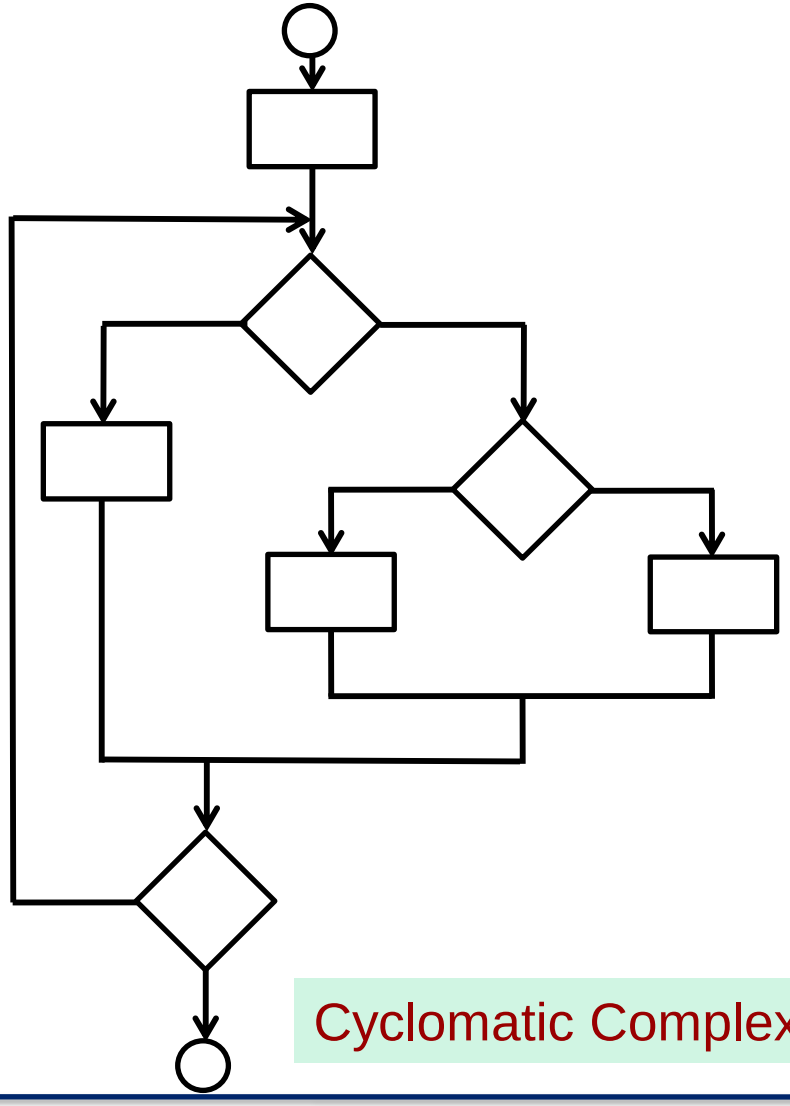


White-box testing

- Goal: to ensure that all statements and conditions have been executed at least once.
- White-box testing test cases
  - All independent paths within a module have been exercised at least once.
  - Exercise all logical decisions on their *true* and *false* sides.
  - Exercise all loops at their boundaries and within their operational bounds.
  - Exercise internal data structures to ensure their validity.

Basis Path Testing

A typical white-box testing

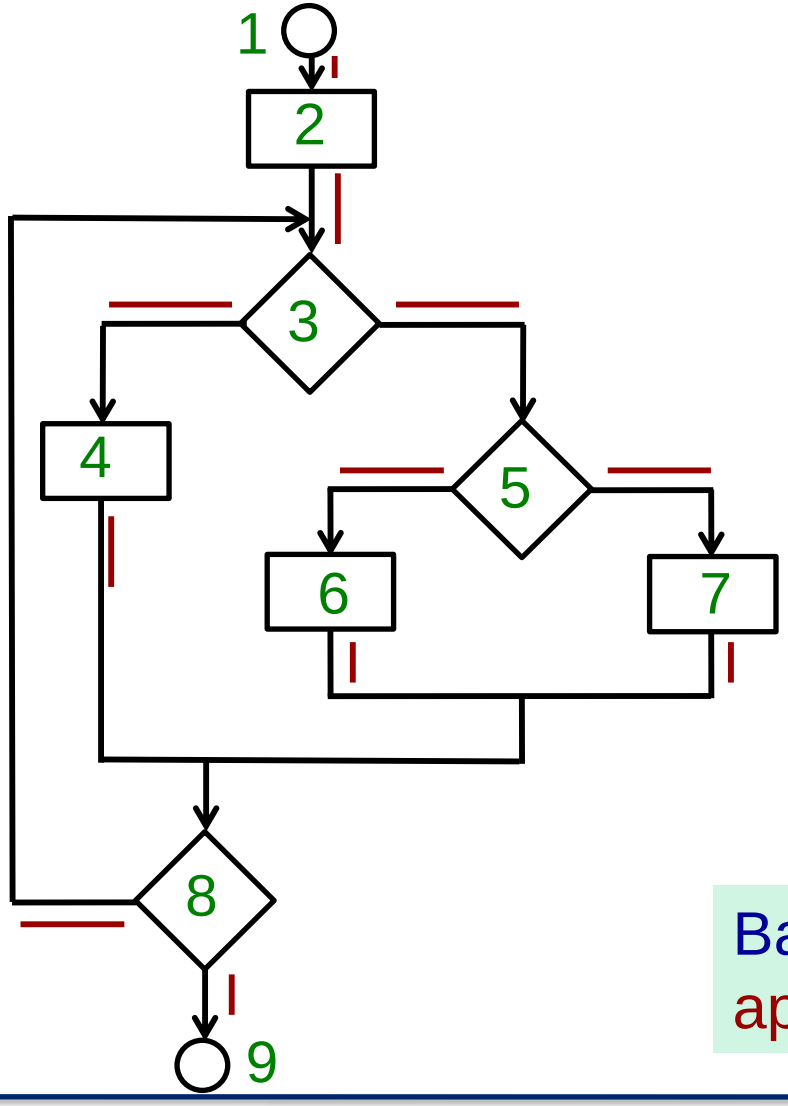


- Cyclomatic Complexity
  - A software metric the provides a quantitative measure of the logical complexity of a program.
  - Defines the number of independent paths in the basis set of a program.
  - Upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once.

Cyclomatic Complexity: number of simple decisions + 1

Basis Path Testing

A typical white-box testing



Cyclomatic Complexity = 4

Path 1: 1, 2, 3, 5, 6, 8, 9
Path 2: 1, 2, 3, 5, 7, 8, 9
Path 3: 1, 2, 3, 4, 8, 9
Path 4: 1, 2, 3, 4, 8, 3, 4, 8, 9

Then we derive test cases to exercise these paths.

Basis path testing should be applied to critical modules.

Implementation and Testing Steps

- Classes need to be implemented from least coupled to most coupled.
- At each step, each class should be fully unit tested using WHITE BOX testing. This involves:
  - Building a test harness.
  - Checking that the methods of the class behave as expected (as defined by their *operation descriptions*).
  - Checking that the methods are 'robust'.
- Integration Testing using BLACK BOX Testing.

Summary

- A “build” is an executable version of the system, normally a part of the system.
- An integration build plan describes the sequence of builds required in an iteration.
- Testing policies define the approach to be used in selecting system tests.
- Tests must be conducted systematically and test cases must be designed using disciplined techniques.
- Black-box testing is to test software requirements
- White-box testing is to test the internal program logic

References

- Chapters 8 – “Software Engineering” textbook by Ian Sommerville
- “Head First Object Oriented Analysis & Design” textbook by Brett McLaughlin *et al*