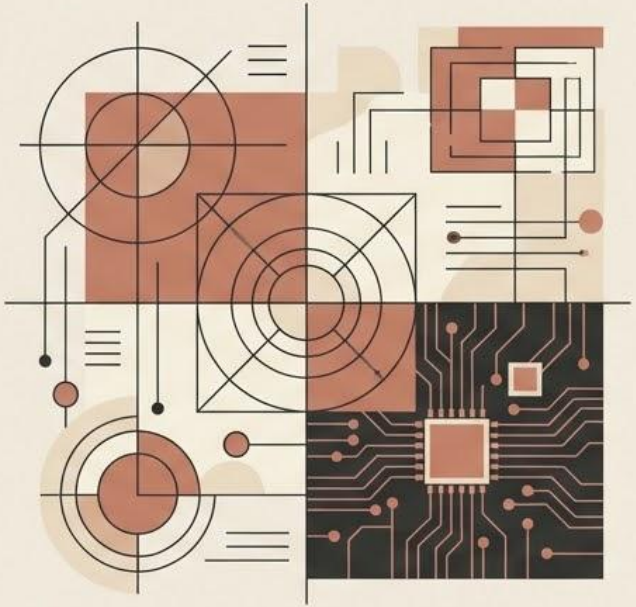
Chapter 18: Testing Conventional Applications

Software Engineering



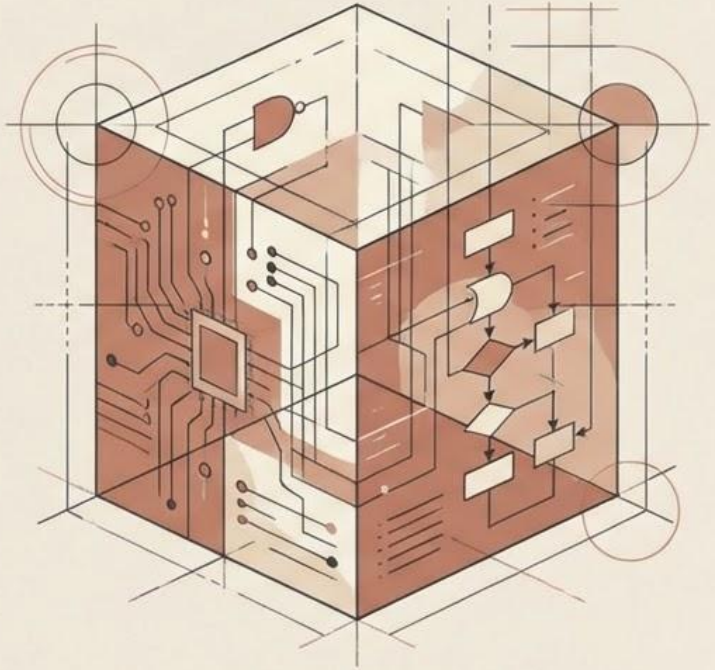
BTech Computer Science and Engineering
Duration: 1 Hour

The Engineer's Approach to Verification

Looking Inside the Code to Design Systematic Tests

Slide 1: Introduction to White-Box Testing

Context: Introduction - Transitioning from high-level strategy to concrete test design.



Slide 2:

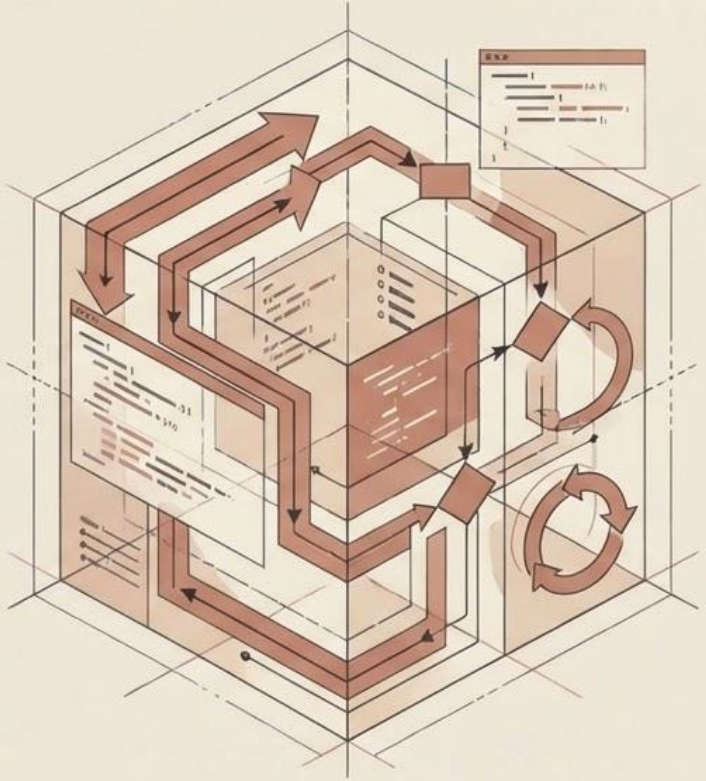
The “How” of Testing

Designing Tests That Actually Find Bugs

The Shift in Focus: We have already discussed the “where” and “when” of overall testing strategies. Today, we dive into the “how.”

The White-Box Perspective: Instead of treating the software as a closed system, we are going to look directly inside the source code.

The Goal: To design systematic, quantifiable test cases that explicitly exercise specific execution paths, logical conditions, and internal loops.



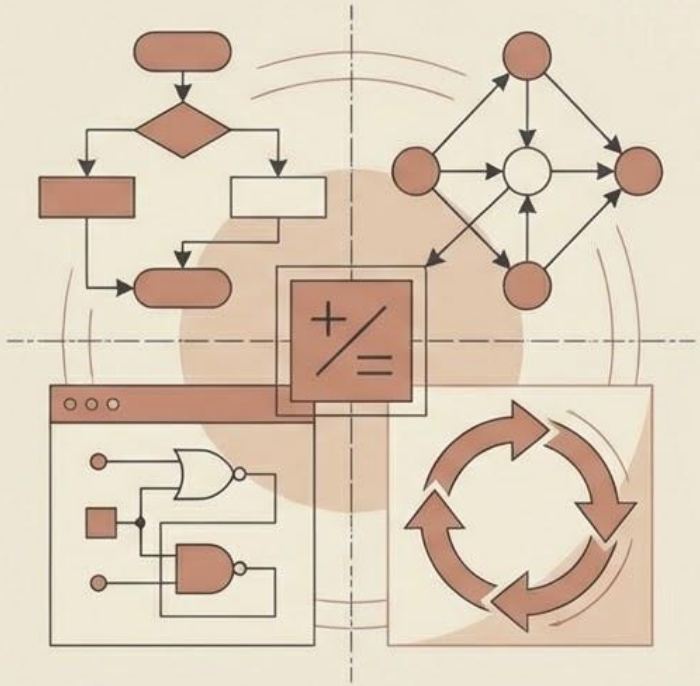
Slide 3:

Learning Objectives

What You Will Master Today

By the end of this lecture, you will be equipped to:

- **Distinguish the Views:** Clearly differentiate between the internal (white-box) and external (black-box) approaches to software testing.
- **Map the Flow:** Apply basis path testing to successfully derive test cases directly from program flow graphs.
- **Measure the Code**: Calculate the cyclomatic complexity of a program and identify its mathematically independent execution paths.
- **Test the Structures**: Describe and apply targeted control structure techniques, including condition testing, data flow testing, and loop testing.



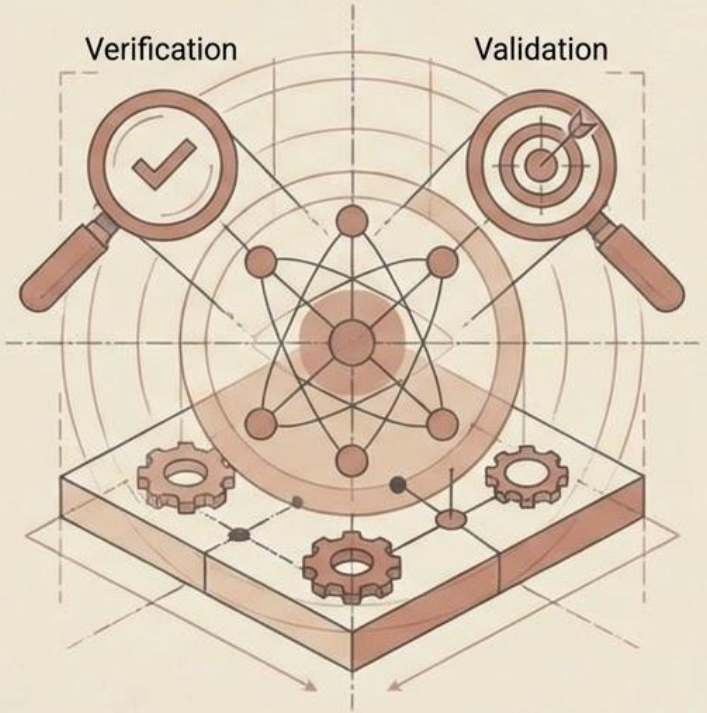
Slide 1:

Part 1: Testing Fundamentals and Views

Establishing what testing actually is before looking at specific techniques.

The Philosophy of Testing

Setting the Stage for Verification and Validation



Slide 2:

Software Testing Fundamentals

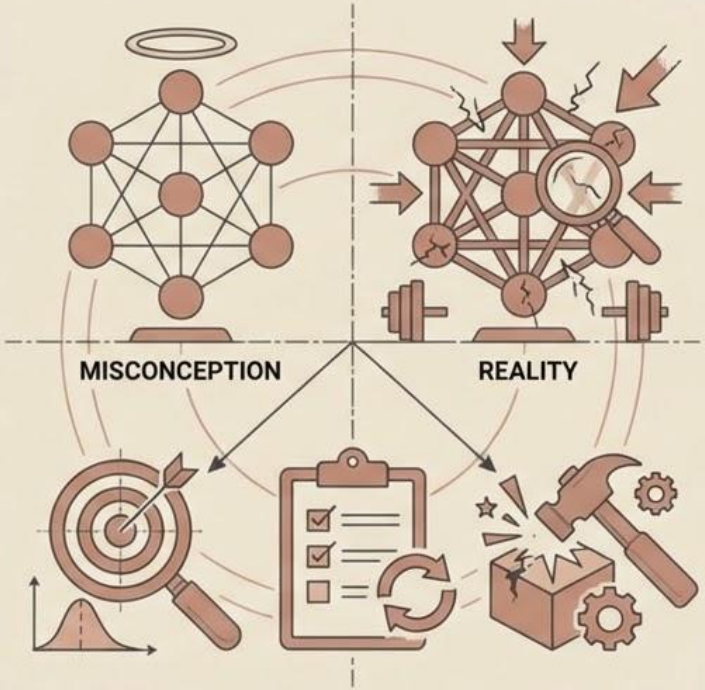
The Primary Goal

The Misconception: Many believe testing is about proving that the software works perfectly or proving the absence of errors.

The Reality: The primary goal of testing is actively trying to find errors that have not yet been discovered.

Key Principles:

- A 'good' test case is one that has a high mathematical and logical probability of finding an as-yet-undiscovered error.
- Tests must be strictly repeatable and thoroughly documented.
- Testing is fundamentally a creative and destructive activity—your job is to try to break the software.



Slide 3:

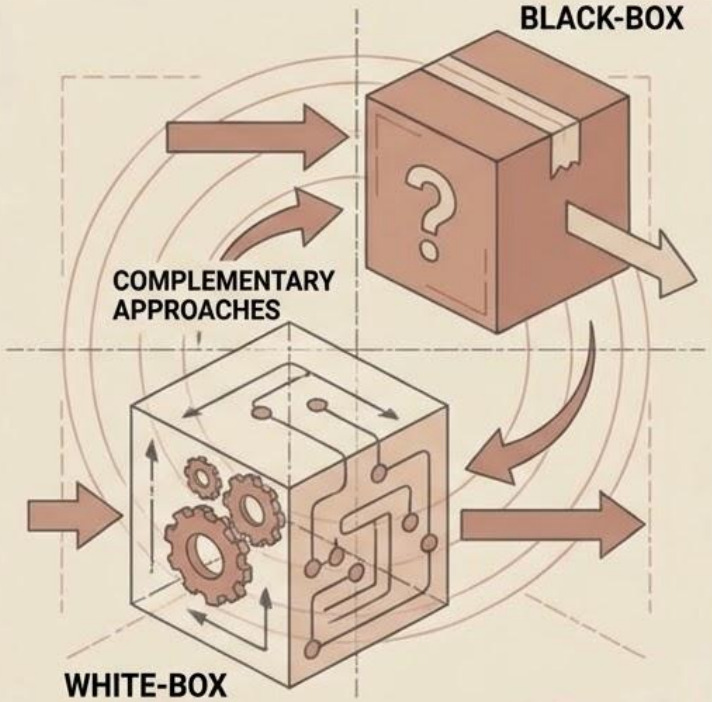
Internal vs. External Views of Testing

Two Complementary Approaches

To thoroughly test software, we must look at it from two completely different perspectives:

Black-Box Testing (The External View): Treating the software as a closed box. We only care about what goes in and what comes out, completely ignoring the internal mechanisms.

White-Box Testing (The Internal View): Also known as "Glass Box" testing. We look directly at the internal structure, logic, and code to see exactly how the data is being processed.



Slide 4:

The Core Differences: White-Box vs. Black-Box

White-Box Testing (Internal)

Focus: Tests internal structure, logic, and paths.

The Question: "How does it work?"

Origin of Tests: Derives tests directly from source code and design.

Defects Found: Finds logic errors, loops, and path problems.

Timeline: Typically used in early testing (Unit, Integration).

Black-Box Testing (External)

Focus: Tests against specified user requirements.

The Question: "What does it do?"

Origin of Tests: Derives tests from requirement documents.

Defects Found: Finds missing functions and interface errors.

Timeline: Typically used in later testing (Validation, System).

Slide 5:

The Synergy of Both Views

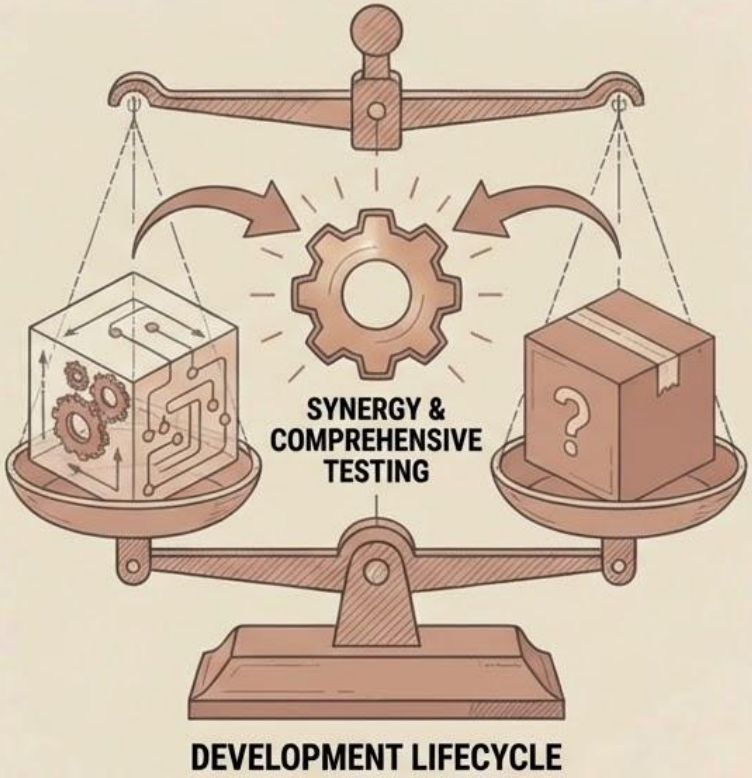
Why We Need Both

The Complete Picture: Neither testing method is sufficient on its own. They are highly synergistic.

The Balance:

- White-Box ensures the code structurally works exactly as the engineer implemented it.
- Black-Box ensures the software actually does what the end-user requested.

The Result: Comprehensive, enterprise-grade software testing requires the strategic application of both views across the development lifecycle.

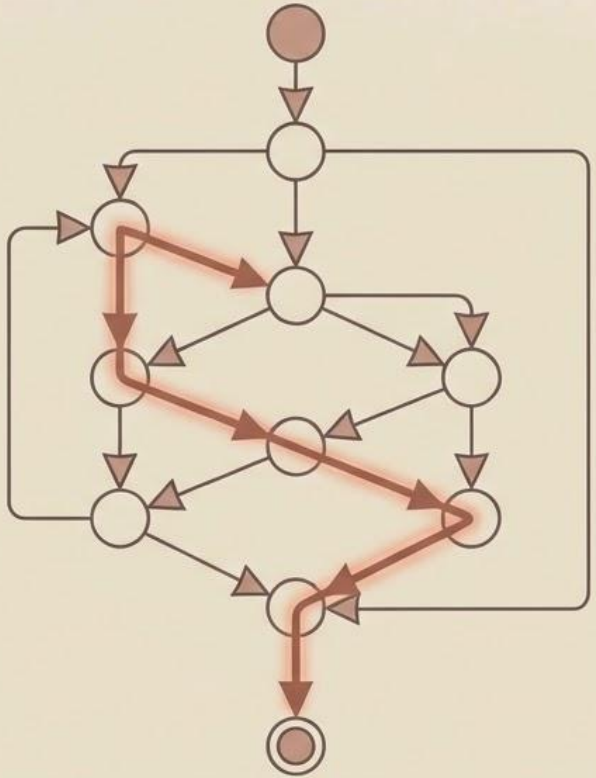


Slide 1: Part 2: White-Box & Basis Path Testing

Examining the Internal Structure

Systematic Test Case Design

Part 2 - Moving from theory to the mathematical derivation of test cases.



Slide 2:

White-Box Testing (Structural Testing)

Why Look Inside the Box?

The Assumption: By knowing the internal structure of the software, we can create precise tests that exercise specific paths, conditions, and data flows.

Why White-Box is Essential:

- Complex Logic: Logic errors are statistically much more likely to hide in complex execution paths.
- False Assumptions: Developer assumptions about how execution paths flow are frequently incorrect.
- Human Error: Typographical errors are common and can quietly alter logic.
- Coverage: We need to ensure that absolutely every statement in the code executes at least once during testing.

Slide 3: Basis Path Testing

Guaranteeing Path Coverage

The Technique:

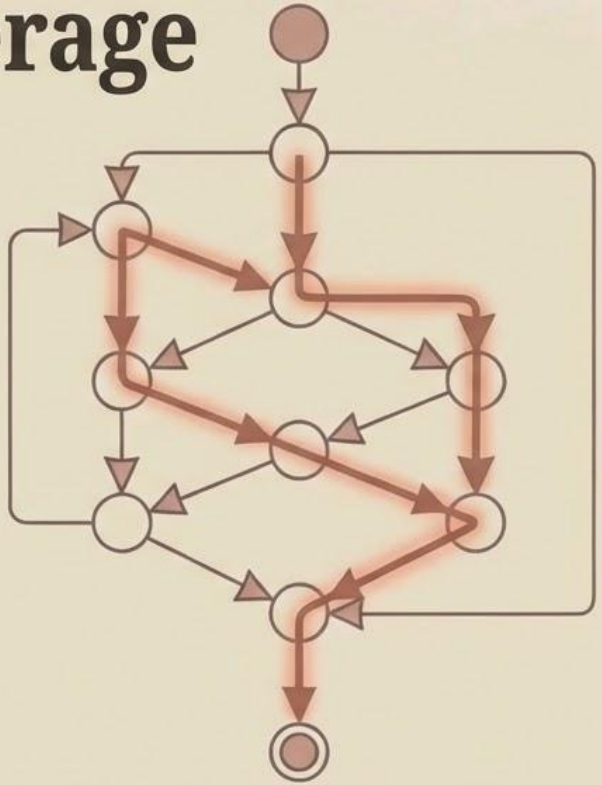
Proposed by Tom McCabe, Basis Path Testing is a foundational white-box technique.

The Goal:

To derive a mathematical set of test cases that completely guarantees every independent path through a program is executed at least one time.

What is an Independent Path?

Any path through the program that introduces at least one completely new set of processing statements or a new logical condition.



Slide 4:

Flow Graph Notation

Mapping the Logic

To calculate paths, we first map the code to a simplified Flow Graph.

- Node (Circle): Represents one or more procedural statements. (Sequential statements can be collapsed into a single node).
- Edge (Arrow): Represents the flow of control between nodes.
- Region: An area completely bounded by edges and nodes (Note: the space outside the graph also counts as one region).
- Predicate Node: A node containing a conditional statement (meaning it has multiple outgoing edges).

Basic IF/THEN Example:

Pseudocode:

```
1. INPUT a,b
2. IF a>b THEN
3.    c = a + b
4. ELSE
5.    c = a - b
6. ENDIF
7. PRINT c
```

Flow Graph Structure:

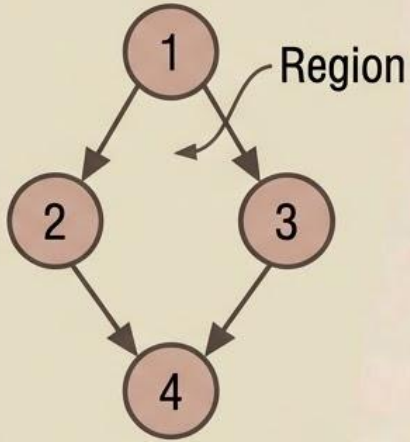


FIGURE 18.1

Flow graph notation

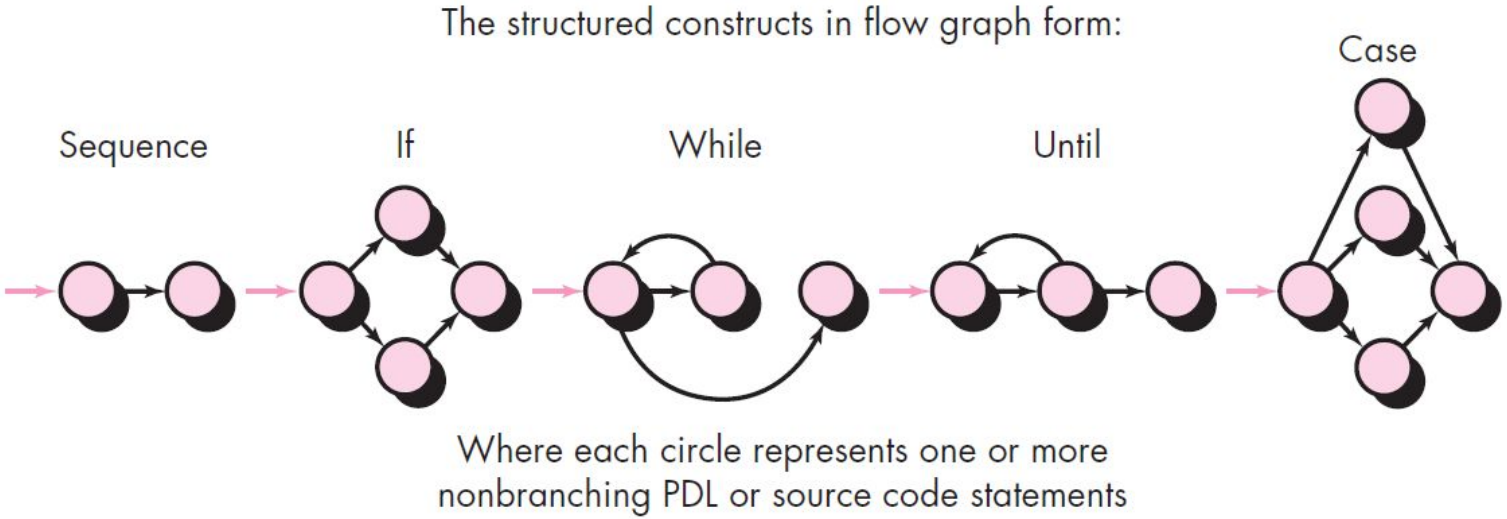


FIGURE 18.2 (a) Flowchart and (b) flow graph

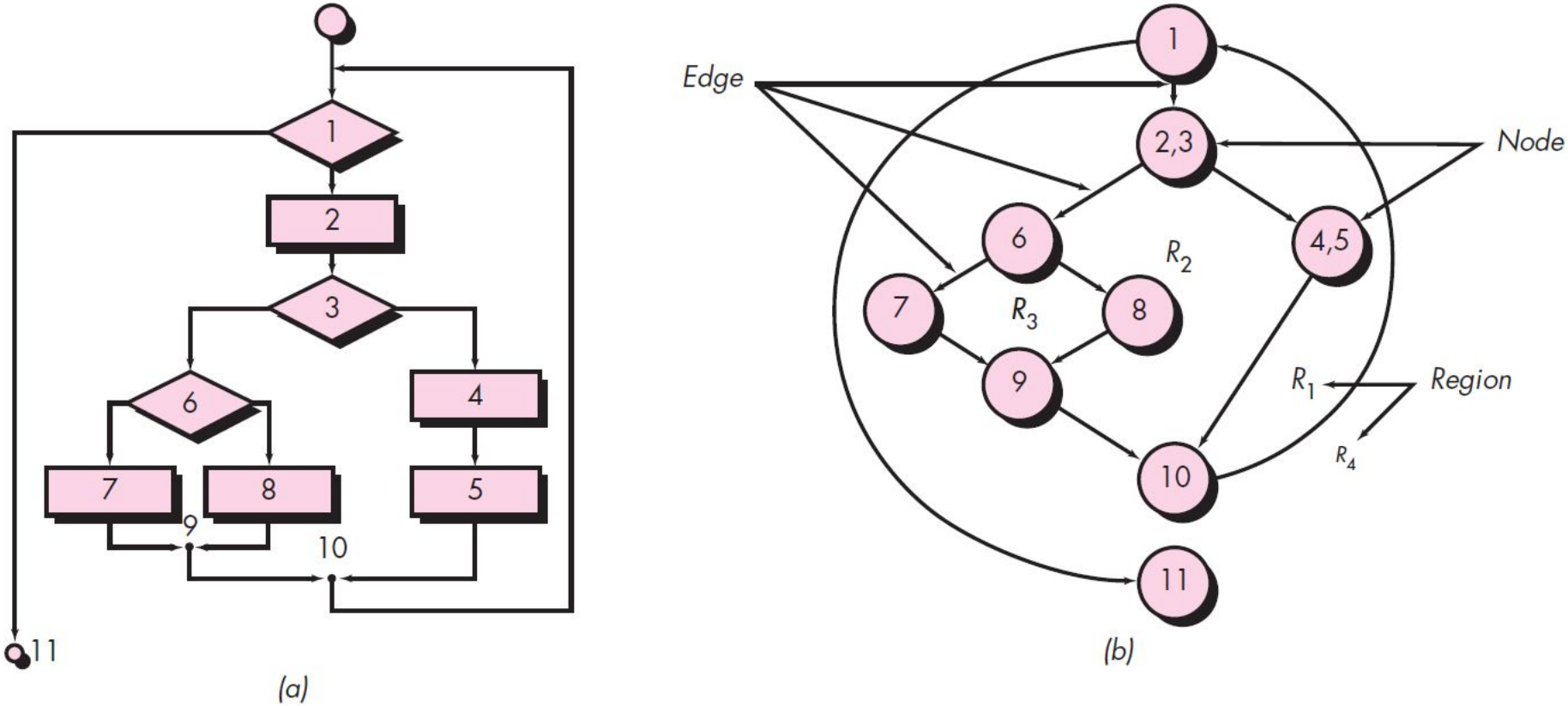


FIGURE 18.3

Compound logic

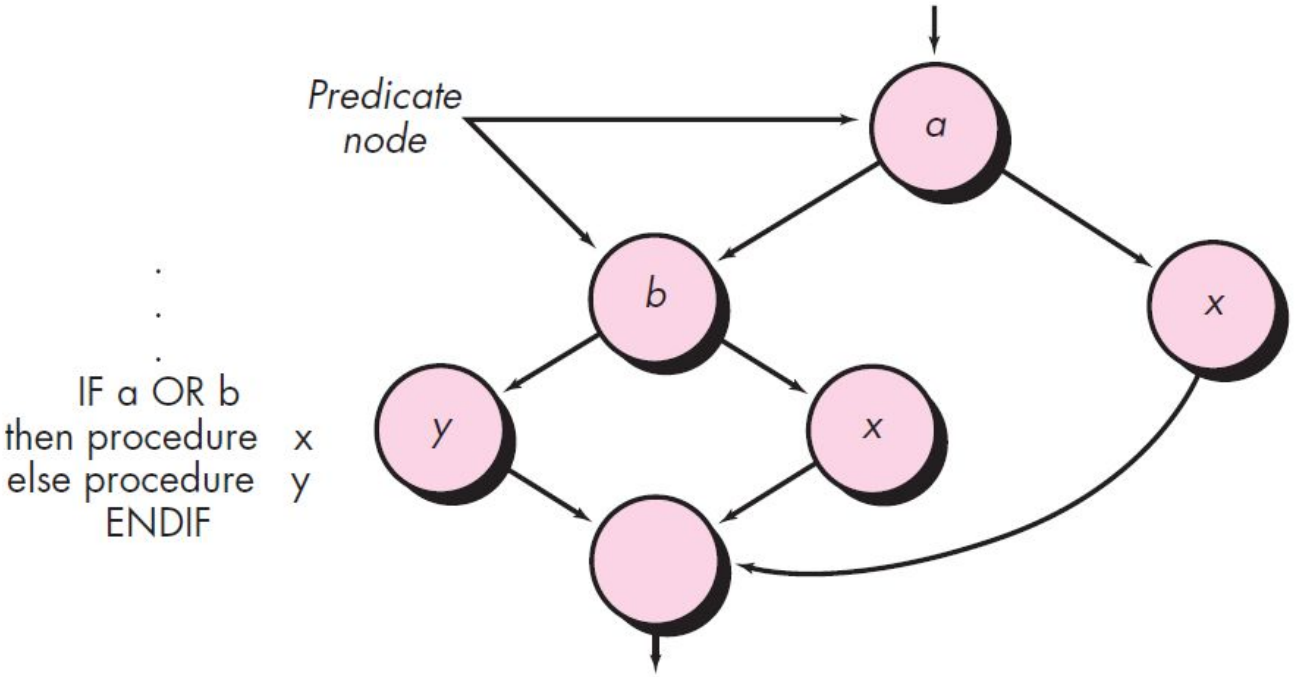


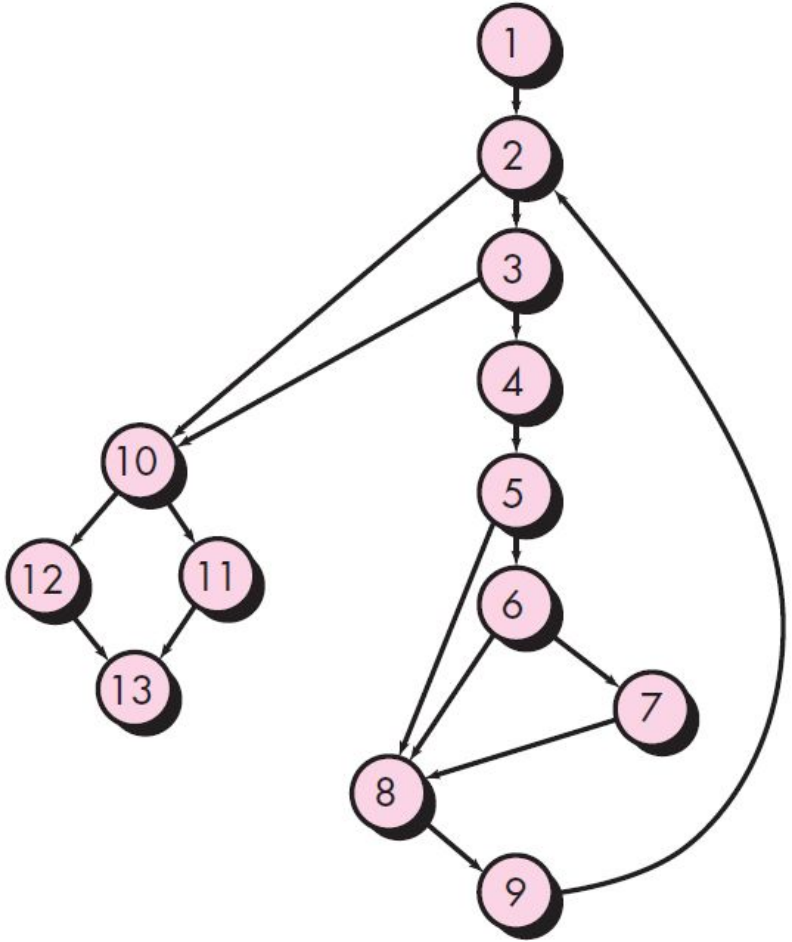
FIGURE 18.4

PDL with nodes identified

```
PROCEDURE average;

    *  This procedure computes the average of 100 or fewer
       numbers that lie between bounding values; it also computes the
       sum and the total number valid.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;                    2
    sum = 0;
    DO WHILE value[i] <> -999 AND total.input < 100   3
4      increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum   6
5         THEN increment total.valid by 1;
7                    sum = s sum + value[i]
          ELSE skip
       ENDIF
8      increment i by 1;
9   ENDDO
    IF total.valid > 0   10
11     THEN average = sum / total.valid;
12     ELSE average = -999;
13  ENDIF
END average
```

FIGURE 18.5

Flow graph for the procedure *average*



Slide 5: Cyclomatic Complexity

Calculating V(G)

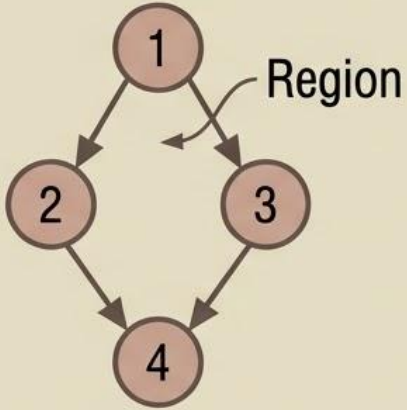
The Metric:

Cyclomatic Complexity, or V(G), is a software metric providing a quantitative measure of a program's logical complexity.
Critically, V(G) exactly equals the number of independent paths in the program.

Three Equivalent Formulas:

- Regions: V(G) = Number of regions in the flow graph.
- Edges & Nodes: V(G) = E - N + 2 (Where E = edges, N = nodes)
- Predicates: V(G) = P + 1 (Where P = number of predicate/decision nodes)

Example Calculation (Standard IF/THEN):



Nodes (N) = 4, Edges (E) = 4
Using formula 2: V(G) = 4 - 4 + 2 = 2 independent paths.

Slide 6:

Deriving Test Cases

The Step-by-Step Process

- Draw: Create the flow graph from the source code or design document.
- Calculate: Determine the cyclomatic complexity V(G).
- Determine Paths: Map out the exact set of independent paths (you will have exactly V(G) paths).
- Design Tests: Create test cases that provide the inputs necessary to force the execution of each specific path.
- Execute: Run the tests and verify the results.

While Loop Example (V(G) = 3):

1. Path 1: Skip the loop entirely.
2. Path 2: Execute the loop exactly one time.
3. Path 3: Execute the loop two times (covering the loop boundary).

Slide 7: Graph Matrices

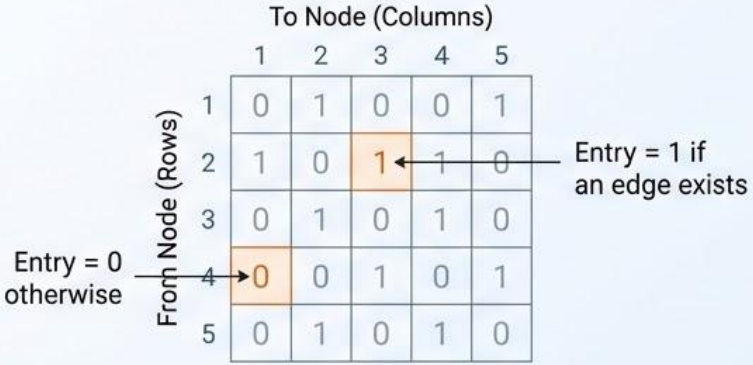
Automating the Complexity Calculation

The Concept:

A Graph Matrix is a tabular representation of a flow graph used to automate the complexity calculation process.

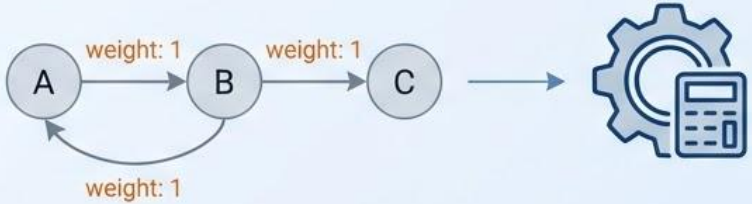
The Structure:

A square matrix where both rows and columns equal the number of nodes.



Connection Matrix:

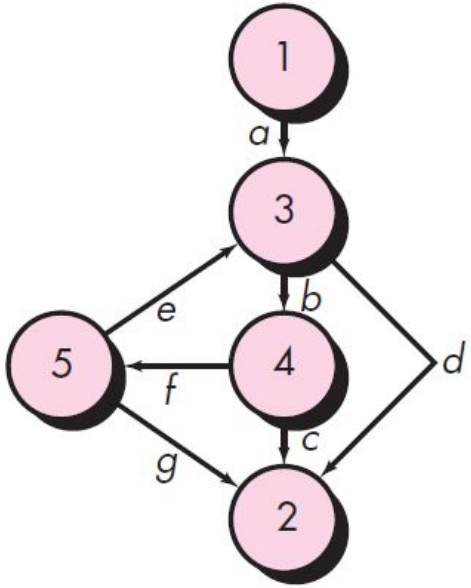
Adds “weight” to the edges (usually 1 for existence).



By summing these weights logically, a computer tool can automatically calculate the Cyclomatic Complexity and generate the required test paths without manual drawing.

FIGURE 18.6

Graph matrix



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Slide 1: Part 3: Control Structure Testing

Beyond the Basis Path

Targeting Specific Logical Structures

Context: Part 3 - Why path coverage isn’t enough, and how we test specific code structures.

Slide 2: The Need for Control Structure Testing

Zooming In on the Details

The Limitation of Basis Paths: While Basis Path testing is incredibly powerful for overall coverage, it is not always sufficient on its own.

The Solution: We must zoom in and deliberately test the specific control structures that make up those paths.

Core Areas of Focus:

- Condition Testing
- Data Flow Testing
- Loop Testing

Slide 3: Condition Testing

Validating the Logic

The Goal:

To rigorously test logical conditions (boolean expressions) to catch subtle logical errors.

Common Types of Condition Errors:

- **Boolean Operator Errors:** Mixing up AND vs. OR.
- **Boolean Variable Errors:** Incorrectly tracking state flags.
- **Parentheses Errors:** Mistakes in the order of mathematical operations.
- **Relational Operator Errors:** Mixing up < vs. ≤ or > vs. ≥.
- **Arithmetic Expression Errors:** Flaws in the underlying math.

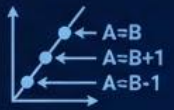
Slide 4: Test Approaches for Conditions

Strategies for Coverage

How do we systematically test these boolean conditions?

Branch Testing:

The simplest form. Ensure you test both the strictly True and strictly False outcomes of every single condition.



Domain Testing:

For relational expressions (e.g., A > B), you must test the exact boundary and the values immediately surrounding it: test A=B, A=B+1, and A=B-1.

Multiple Condition Testing:

Testing all possible combinations of boolean sub-conditions. (Note: This requires 2^n tests for n conditions, which scales poorly but is highly thorough).

Slide 5: Data Flow Testing

Following the Variables

The Goal:

To track the “lifecycle” of variables—specifically their definition and their use—to ensure they are properly initialized, utilized, and destroyed.

Key Concepts:

- **Define (def):** The exact location where a variable receives a value (e.g., an assignment, an input, or a parameter).
- **Use (c-use / p-use):** The location where that variable’s value is actively used.
- **c-use (computation use):** Used in a calculation (e.g., the right side of an equation).
- **p-use (predicate use):** Used in a conditional expression (e.g., inside an if statement).

Slide 6: Managing Data Flow Anomalies

Catching the Invisible Bugs

Data Flow testing specifically hunts for ‘anomalies’ that might not crash the program immediately but indicate deeper flaws:

The Anomalies:

- **Defined but never used**: Indicates dead code or an unnecessary variable wasting memory.
- **Used without being defined**: A critical bug that will likely cause a crash or garbage data output.
- **Defined multiple times before use**: May be perfectly fine, but requires investigation (did a loop overwrite important data?).

Test Criteria:

- **All-defs**: Ensure every single definition reaches at least one use.
- **All-uses**: Ensure every definition reaches all of its possible uses.
- **All-du-paths**: Test every possible path from each definition to each of its uses.

Slide 7: Loop Testing

Breaking the Cycles

The Goal:

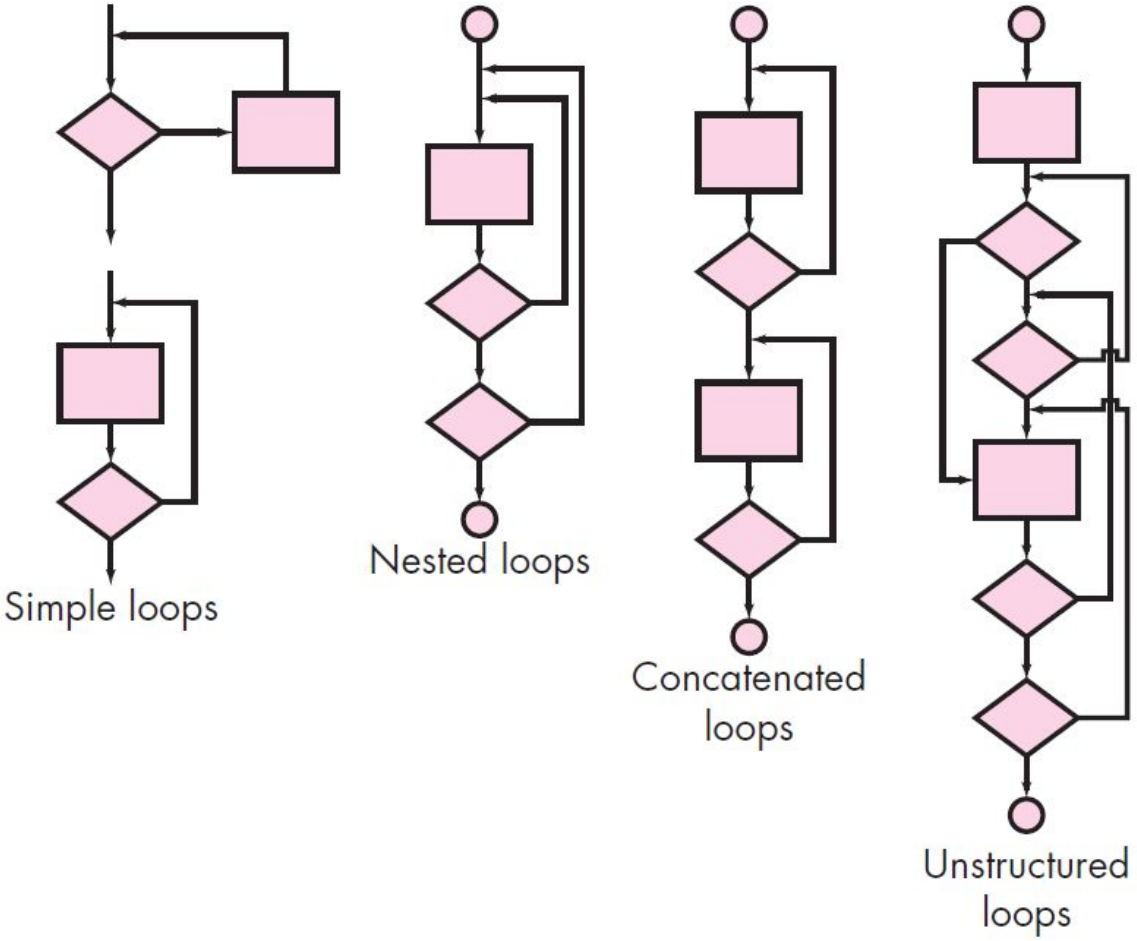
Loops are notoriously one of the most common sources of software bugs (off-by-one errors, infinite loops, premature termination). We must test them specifically.

Loop Types:

- **Simple Loops:** Standard for or while loops with a single, standalone structure.
- **Nested Loops:** Loops placed inside other loops.
- **Concatenated Loops:** Sequential, independent loops that run one after the other.

FIGURE 18.7

Classes of Loops



Slide 8: Executing Loop Tests

The Testing Framework

Testing Simple Loops

(Where n = max allowed iterations):

- Skip the loop entirely (0 iterations).
- Make exactly one pass (1 iteration).
- Make exactly two passes (2 iterations).
- Make a "typical" number of passes (m iterations, where $m < n$).
- Boundary testing: Run $n - 1$, n, and $n + 1$ iterations.

Testing Nested Loops

- Start at the innermost loop and set all outer loops to their minimum values.
- Test the inner loop thoroughly.
- Work outward, holding inner loops at typical values until all loops are tested.

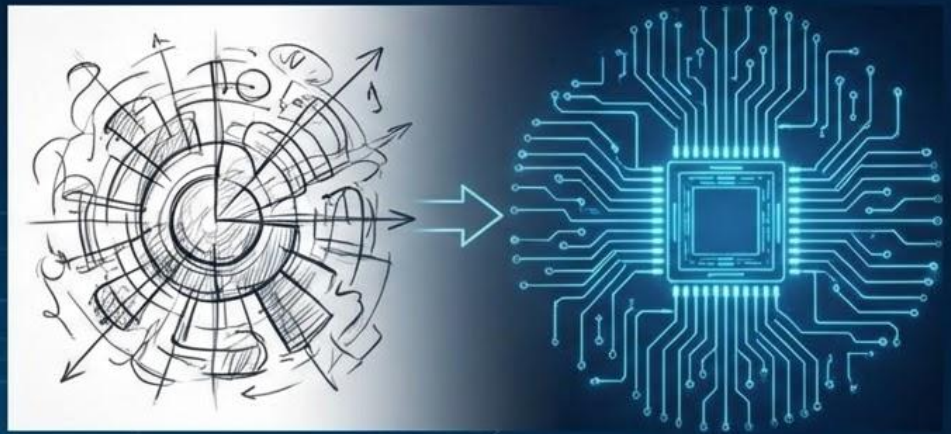
Testing Concatenated Loops

- If independent, test each separately as a simple loop.
- If dependent, test them as if they were nested.

Slide 1: Conclusion & Key Takeaways

From Art to Engineering

Systematizing Test Case Design



Context: Part 5

- Summarizing the core methodologies and their impact on software quality.

Slide 2: The Core Pillars of White-Box Testing

Looking Inside the Code

The White-Box Philosophy

- We use our internal knowledge of the software's structure to design tests that deliberately exercise specific paths, logical conditions, conditions, and data flows.

Basis Path Testing

- By utilizing flow graphs, we can identify a strict mathematical set of independent paths through the code. This guarantees we generate at least $V(G)$ test cases to achieve baseline coverage.

Cyclomatic Complexity ($V(G)$)

- This vital metric quantifies program complexity. Remember: the complexity score exactly equals the number of independent paths that must be tested.

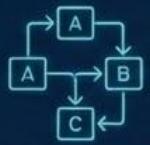
Slide 3: Extending the Coverage

Targeting Specific Control Structures

Basis paths give us the skeleton, but Control Structure Testing ensures the finer details are flawless:

Condition Testing

- Rigorously tests boolean expressions to catch operator errors and logic flaws.



Data Flow Testing

- Tracks exactly where variables are defined and where they are used, hunting for anomalies like dead code or uninitialized values.

Loop Testing

- Systematically tests the boundaries and behaviors of simple, nested, and concatenated loops to prevent infinite cycles and off-by-one errors.

Slide 4: The Final Thought

The Engineer’s Microscope

The Evolution of Testing

Together, these techniques provide a highly systematic, quantifiable approach to test case design. They officially move software testing away away from being a guessing “art” and turn it into a rigorous engineering discipline.

> “White-box testing is the engineer’s microscope—it lets us examine the internal workings of our code with precision. Combined with black-box testing, we achieve both structural coverage and functional validation.”