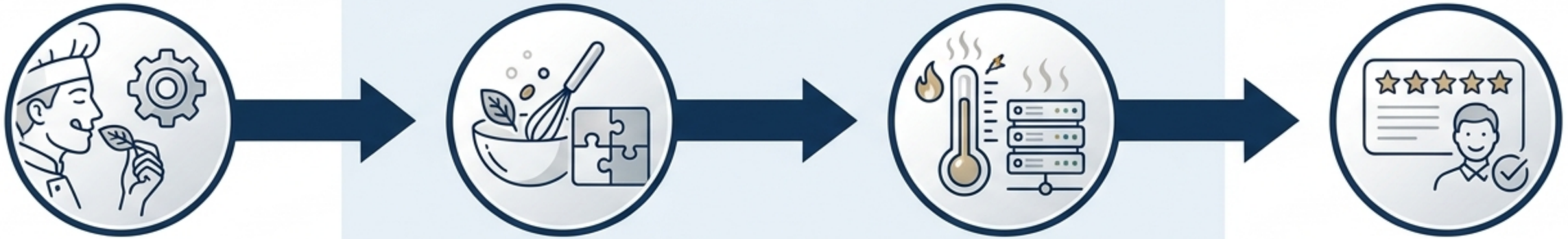
Software Testing and Techniques: From Zero to Mastery

軟體測試與技術：從零基礎到完全掌握

費曼學習法帶你拆解程式碼的邏輯藍圖 (Deconstructing the Logical Blueprint with the Feynman Technique)

The Big Picture: The 4 Stages of Testing (軟體測試的四個階段)



Stage 1: Unit Testing (單元測試)

Focus: A single method or class.

Restaurant Metaphor: Tasting individual ingredients (品嚐單一食材).

Stage 2: Integration Testing (整合測試)

Focus: Modules working together.

Restaurant Metaphor: Combining ingredients into a recipe (測試食譜的組合).

Stage 3: System Testing (系統測試)

Focus: Performance, security, overall system.

Restaurant Metaphor: Stress-testing the kitchen during a dinner rush (廚房高壓實戰測試).

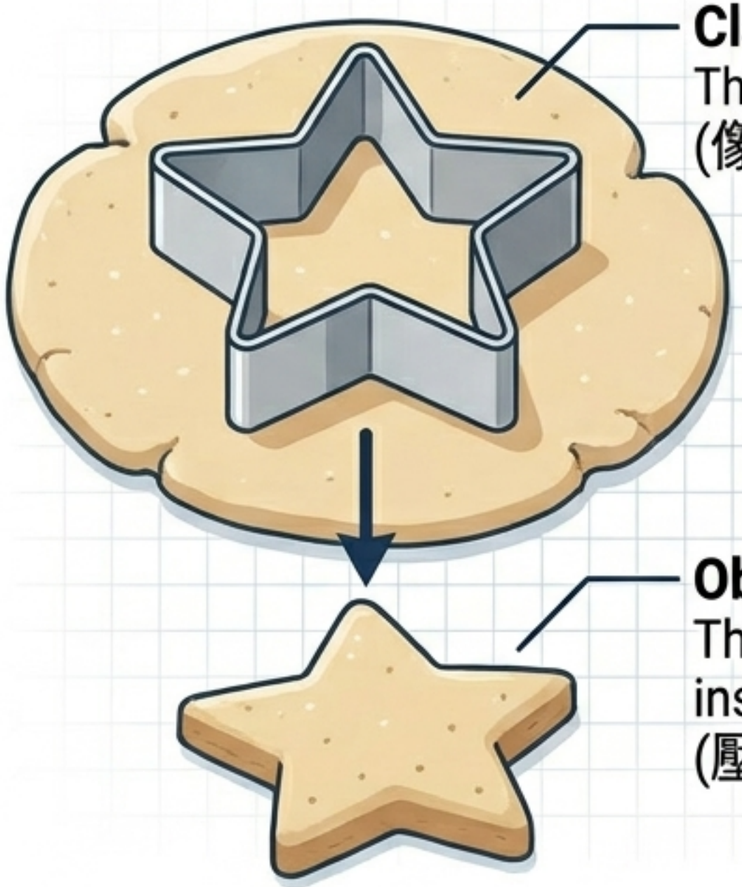
Stage 4: Acceptance Testing (驗收測試)

Focus: Meeting business needs.

Restaurant Metaphor: Serving the final dish to the customer (端給顧客試吃).

Foundation: What is a Unit? (基礎概念：什麼是「單元」？)

Zero to Mastery Note

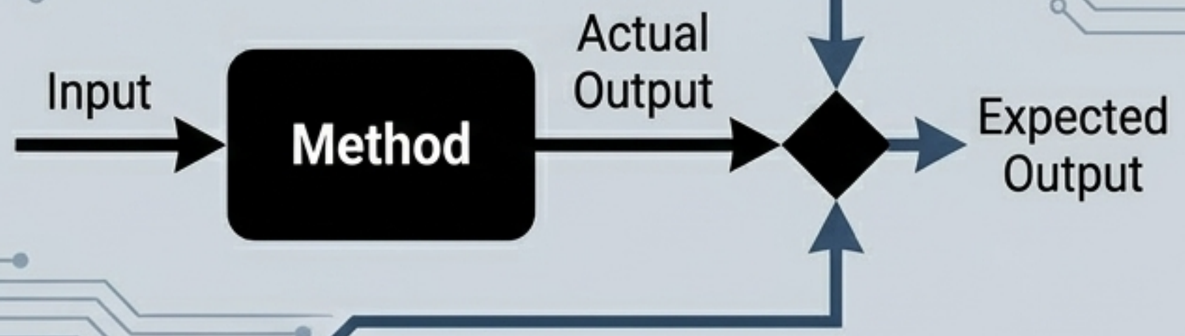


Class (類別):
The blueprint or mold
(像是餅乾模具)

Object (物件):
The actual working instance
(壓出來的實體餅乾)

Unit Testing (單元測試) verifies the smallest testable parts (methods) of these objects.

The 3 Verdicts (三種測試結果)



- **Pass (通過)**: Output matches expected (符合預期).
- **Fail (失敗)**: Output does not match expected (邏輯錯誤，產出不符).
- **Error (錯誤)**: The test crashed before finishing (程式崩潰或發生異常 Exception).

Mastering JUnit: Framework & Annotations (JUnit 測試框架解析)

The Prep (備料階段):
Initializes test fixtures and objects before *each* test runs.
(每次測試前重置環境).

```
public class RecipeTest {

    @Before
    public void setUp() {
        // Initialize objects
    }

    @Test
    public void testCooking() {
        // Execute and Assert
    }

    @After
    public void tearDown() {
        // Clean up resources
    }
}
```

The Execution (烹飪執行):
The actual test case method.
(獨立的測試案例).

The Cleanup (打掃廚房):
Releases resources, closes connections.
(測試後的環境清理).

TestSuite: The Menu (套餐組合).
Running multiple test cases together dynamically or statically.

The Heart of JUnit: Assertions (核心靈魂：斷言)

Assertions determine the verdict. If true, continue; if false, throw a ***Fail*** (斷言是主廚的味覺標準，不符標準直接宣告失敗).

Assertion Command	Logical Function	Everyday Analogy
assertEquals(expected, actual)	Are they exactly the same? (檢查數值或字串是否完全相同)	Does the soup taste exactly like the recipe dictates? ✅
assertTrue(condition)	Is the statement correct? (檢查條件是否為真)	Is the oven turned on? (Yes/No) [Success ✅
assertNull(object)	Is the container empty? (檢查物件是否尚未實例化/為空)	Is the ingredient jar completely empty? [Success ✅
assertSame(expected, actual)	Are they the exact same physical object? (檢查是否指向同一個記憶體實體)	Are these two plates literally the same plate? [Success ✅

JUnit in Action: The Student Class (實戰示範：JUnit 腳本撰寫)

Scenario Requirements

```
// code for Student class continues from the previous page

public class Student {
    private int student_mark;
    public void setStudent_mark(int inmarks) {
        student_mark = inmarks;
    }
    public int getStudent_mark() {
        return student_mark;
    }
    public boolean sameMarks(Student
        student
            data anotherStudent) {
        return student_mark ==
          anotherStudent.getStudent_mark();
    }
}
```

Completed JUnit Script

```
public void setUp() {
    student1 = new Student();
}

public void testCase1() {
    student1.setStudent_mark(80);
    assertEquals(80, student1.getStudent_mark());
}
```

Step 1: Instantiation (實例化)

Step 2: Execution & Assertion

The Blueprint: Control Flow Graph (CFG) Basics (邏輯藍圖：控制流程圖)

Translating invisible code into a visible mathematical graph (**將程式碼轉換為可見的數學圖形結構**).

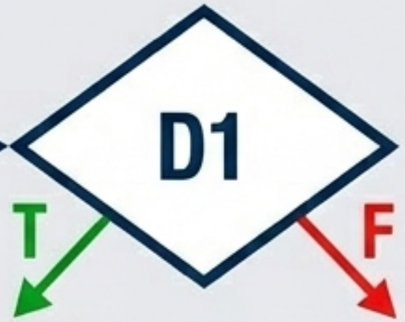
Nodes (節點 - S)

```
x = x + 1;
```

Represent sequential Statements (**循序執行的語句**).

Decisions (判斷 - D)

```
if (a > b)
```



Represent branch points like `if` or `while` loops (**產生分岔的條件判斷**).

Edges (邊)

Directed arrows representing control flow transfer (**執行方向**).

Code to Graph Translation (程式碼到圖形的轉換)

Feynman Note: Treat the CFG like drawing a map of all possible corridors a waiter could walk.

Pseudocode

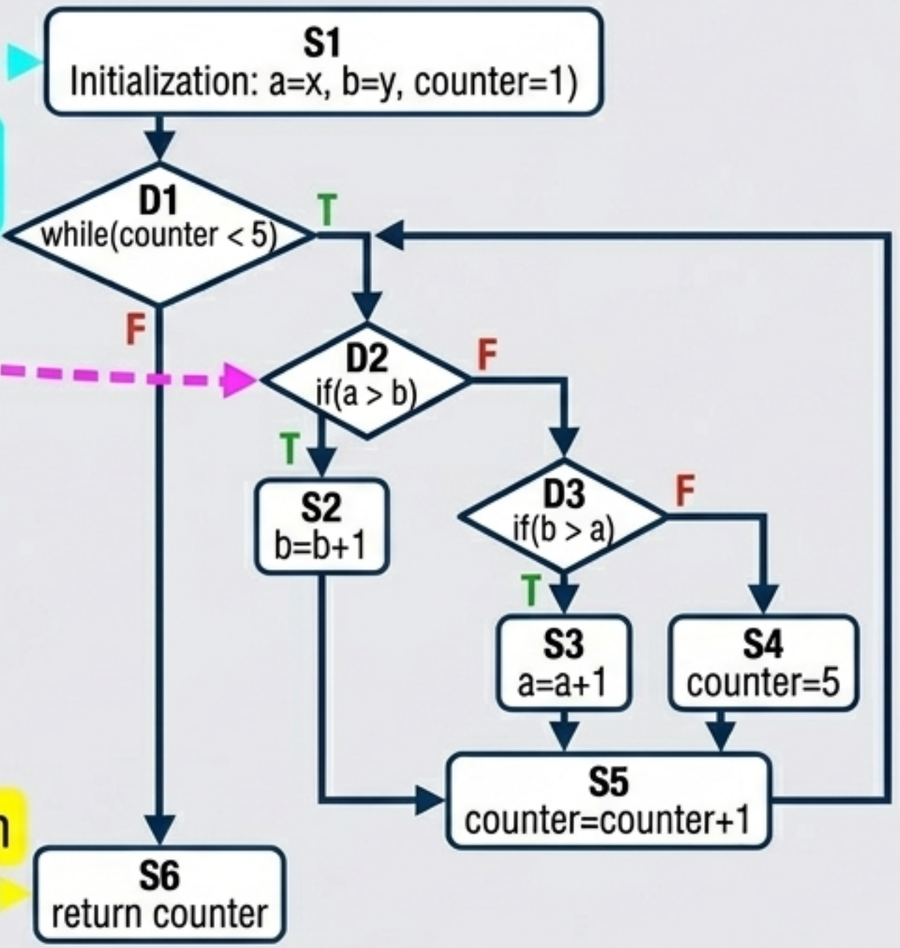
```
public int CompareMethod (int x, int y) {
    int a = x;
[S1] int b = y;
    int counter = 1;
    while (counter < 5) { <D1>
        if (a > b) <D2>
[S2]        b = b + 1;
        else {
            if (b > a) <D3>
[S3]            a = a + 1;
            else
[S4]            counter = 5;
        }
[S5]    counter = counter + 1;
    }
[S6] return counter;
}
```

Initialization (S1) connects to Loop Condition (D1).

If True, enter Loop

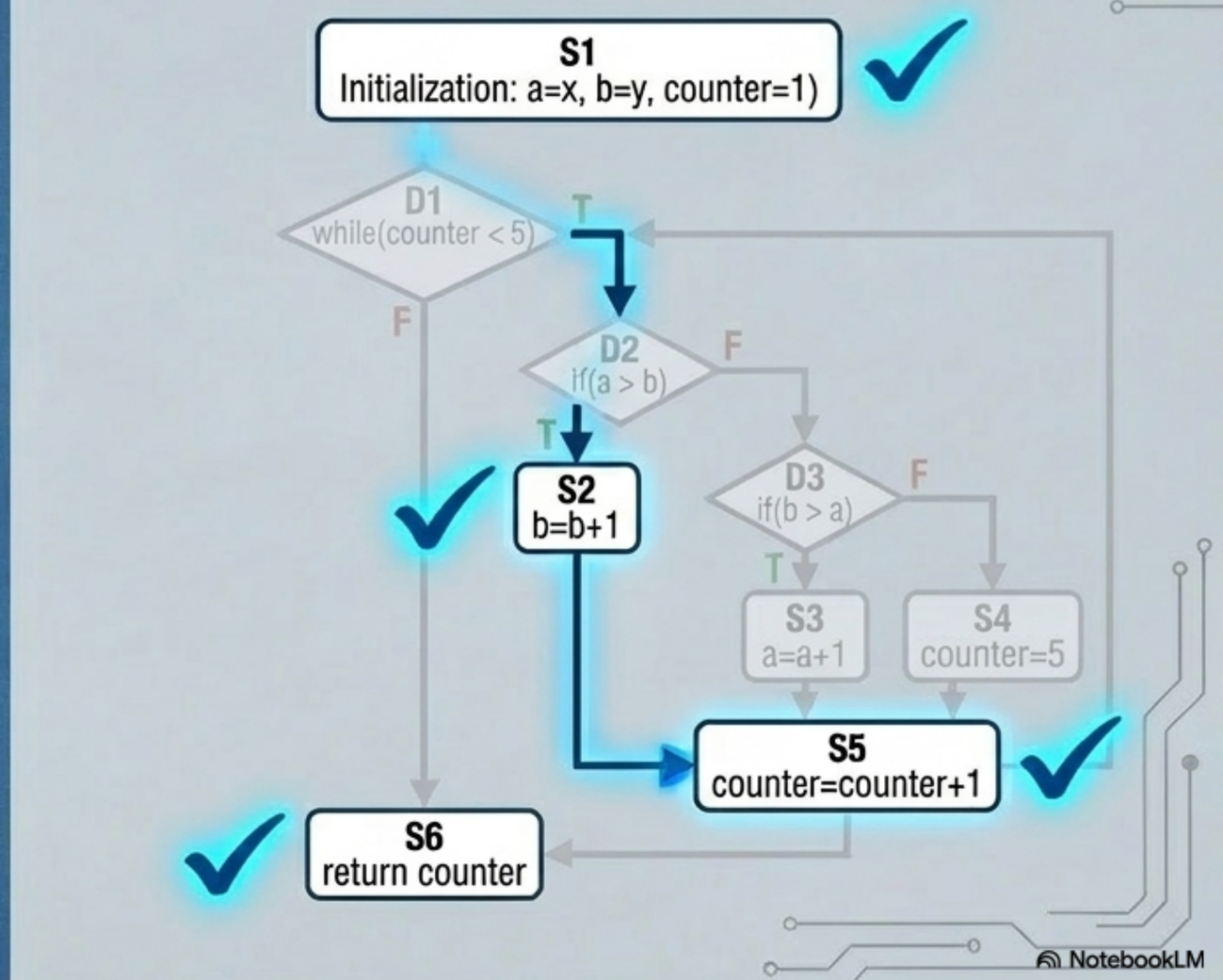
If False, exit Loop to Return

Mapped CFG



Coverage Criteria 1: Statement Coverage (語句覆蓋率)

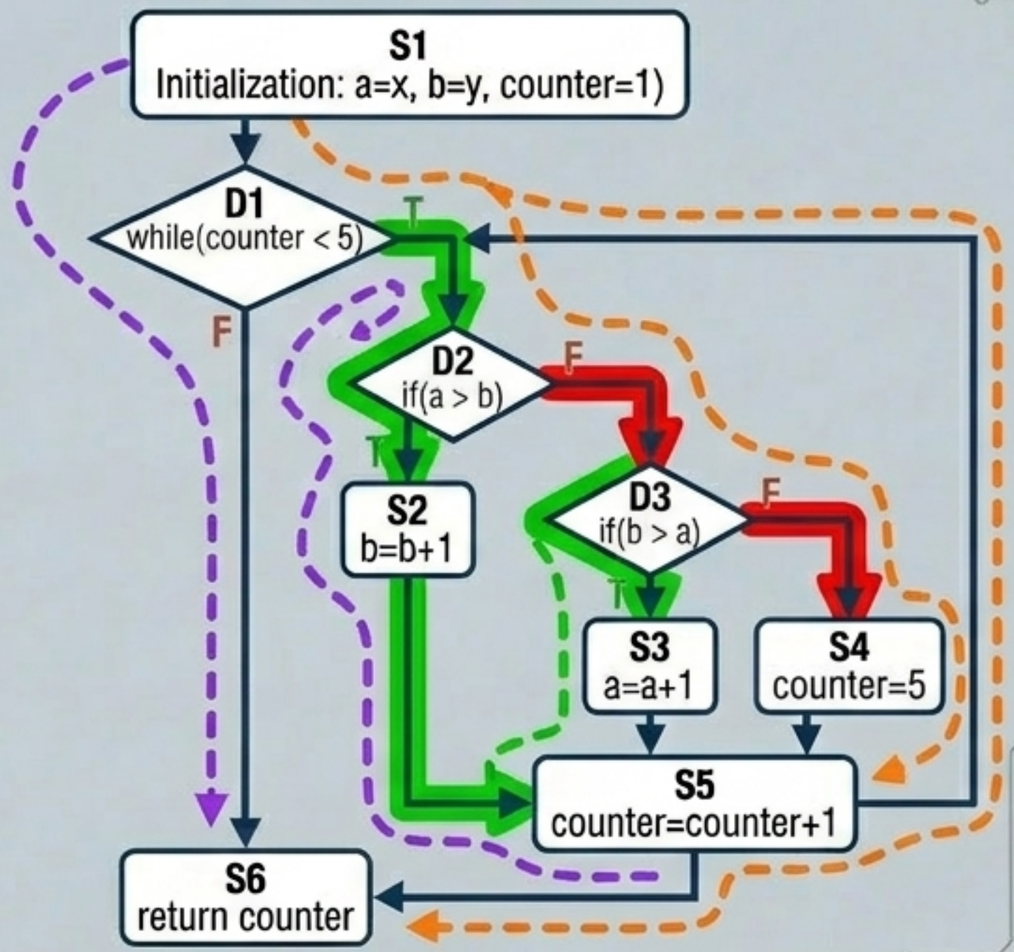
- **Definition:** Every statement (node) must be executed at least once (每一行程式碼都至少被執行過一次).
- **Metaphor:** 掃地機器人打掃房間 (A robot vacuum entering every room at least once).
- **Calculation:** Usually requires the minimum number of paths (Often just 1 path is enough).
- **Weakness:** Might miss hidden logical bugs in the False paths of an `if` statement!



Coverage Criteria 2: Branch/Decision Coverage (分支覆蓋率)

- **Definition:** Every branch direction (True and False) must be traversed at least once (每一個判斷條件的「真」與「假」路線都要走過).
- **Metaphor:** 走過迷宮裡所有的通道 (Walking down every possible connecting hallway, not just visiting the rooms).

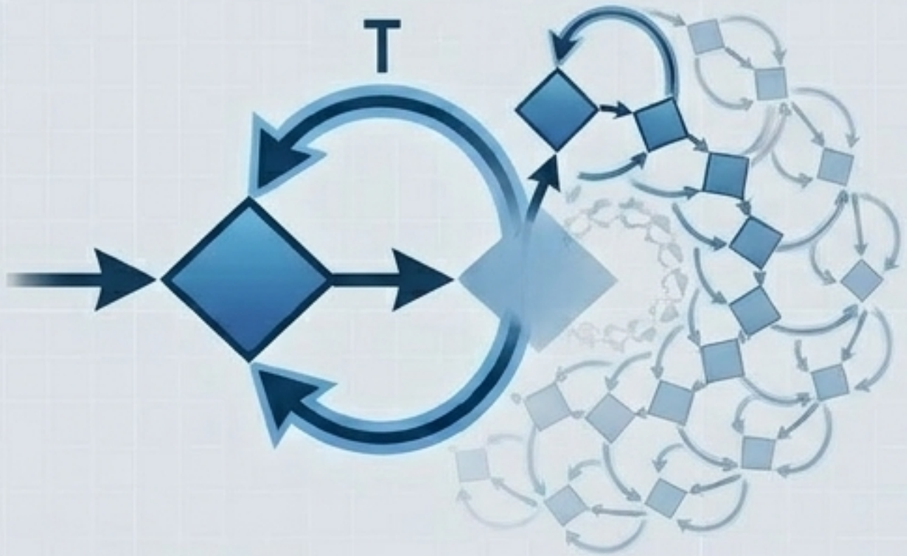
Fact: Achieving 100% Branch Coverage *automatically* guarantees 100% Statement Coverage.



Coverage Criteria 3: Path Coverage (路徑覆蓋率)

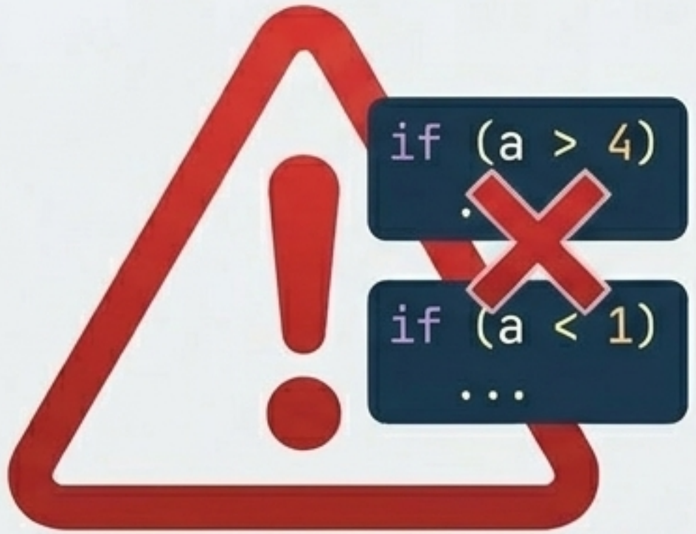
Definition: Every possible route from start to finish must be tested (測試從頭到尾的所有可能路線組合).

The Loop Problem



Loops (迴圈): A `while` loop can create an infinite number of paths (產生無限量組合).

The Logic Problem (Alert)

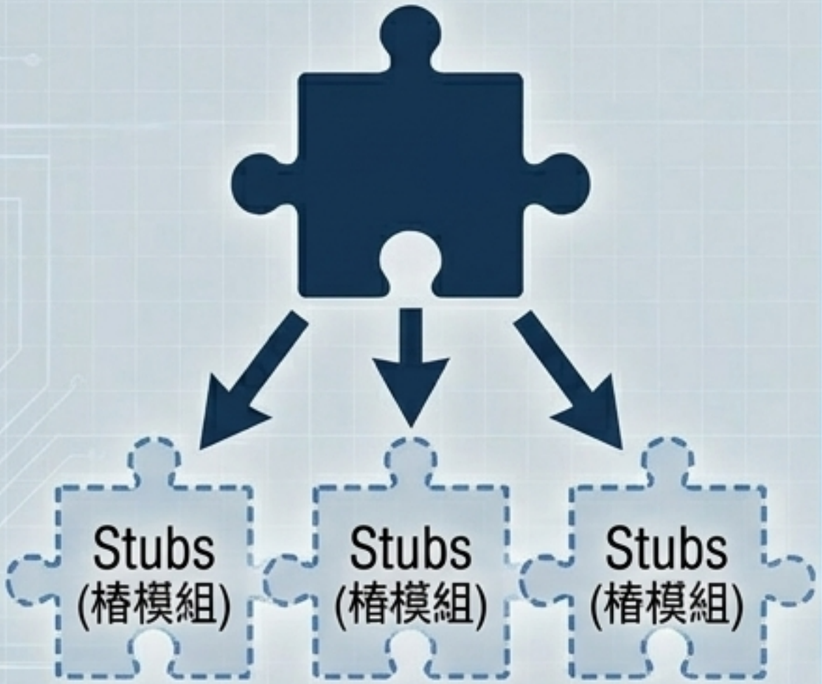


Infeasible Paths (不可行路徑): Logically impossible combinations.

Conclusion: 100% Path Coverage is often impossible in real-world systems.
(100% 路徑覆蓋在現實中幾乎不可能達成).

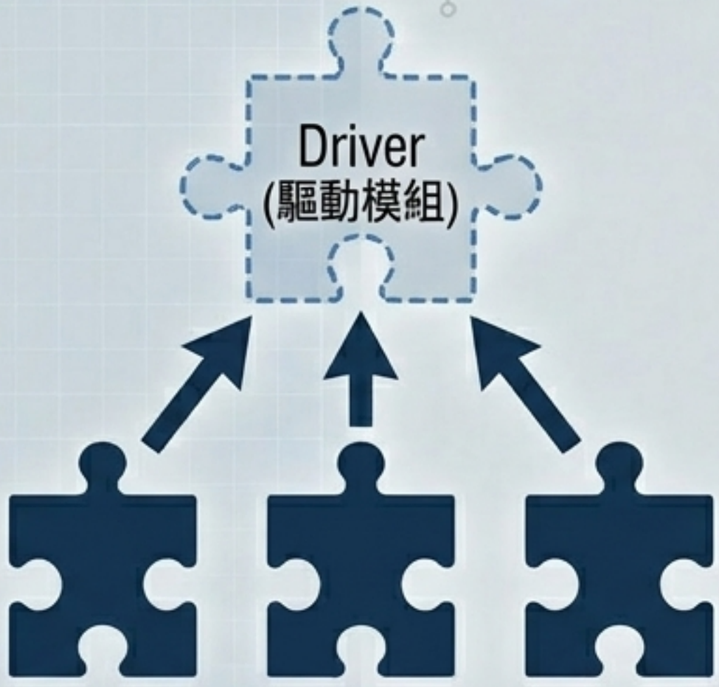
Stage 2: Integration Testing Strategies (整合測試策略)

Top-down (由上而下)



Tests main logic first.
Uses Stubs to simulate missing lower-level functions.
(Stub = 替身演員, providing fake data).

Bottom-up (由下而上)



Tests base utilities first.
Uses Drivers to simulate the missing main program.
(Driver = 代理導演, issuing commands).

Big-Bang (大爆炸)

Integrating everything at once
(Not recommended / 不推薦).

Stage 3: System Testing (系統測試)

Goal: Ensure all modules work together to meet business requirements without error
(由系統分析師執行，確保整體符合業務需求)

PERFORMANCE TESTING (效能測試)

- Evaluates Throughput, Response Time, and Memory Utilization (尋找資源瓶頸)

SECURITY TESTING (安全測試)

- Resistance to unauthorized access (防止未授權存取)

CONFIGURATION TESTING (配置測試)

- Hardware/OS compatibility (硬體與作業系統相容性)

Stage 4: Acceptance Testing (驗收測試)

Goal: Confirm the system is complete and acceptable to the users (確認系統完全滿足商業需求).

Alpha Testing (內部測試)

- Conducted by internal team in a controlled environment.
- Uses Made-up Data (虛擬假資料).

Beta Testing (公開測試)

- Conducted by real end-users in their own environment.
- Uses Real Data (真實資料).
- Monitors for unpredicted real-world errors.

The Ultimate Testing Cheat Sheet (期末總複習：測試階段矩陣)

Stage (階段)	Focus (測試焦點)	Tester (執行者)	Key Methods/Metrics (核心方法)
Unit (單元)	Single Method	Developer	JUnit, CFG, Statement/Branch Coverage (白箱測試 / White-Box)
Integration (整合)	Module interfaces 整合	Developer/Tester 發行者	Top-down (Stubs), Bottom-up (Drivers)
System (系統)	Entire application 全程	System Analyst 系統分析	Performance, Security, Load testing
Acceptance (驗收)	Business logic 繪的	End Users 終關學者	Alpha (Made-up data), Beta (Real data) (黑箱測試 / Black-Box)