Testing in Software Development

CSE260, Computer Science B: Honors

Paul Fodor

Stony Brook University

http://www.cs.stonybrook.edu/~cse114

Today's Topics

- Everybody makes mistakes/errors - We must deal with errors in programming
- Prevention of errors, Thin interfaces
- Software Testing Goals: Verification and Validation
- Unit testing, Test automation, Integration testing, System testing
- Pure Java Test Drivers
- **`assertions`** and Test Drivers with assertions
- Unit Testing with **`JUnit`**
  - Arrange-Act-Assert (AAA) pattern
- Exhaustive vs. Sampled Testing
  - Black-box testing
    - Boundary Conditions
  - Glass-box testing
    - Structural Analysis & Path Completeness Table
- Notes on **`static import`**

Everybody makes mistakes/errors

- We must deal with errors in programming
- Early errors are usually syntax errors
  - The compiler will spot these
- Later errors are logic errors - also known as bugs
  - The compiler cannot help with these
    - Some logical errors have no immediately obvious manifestation
- Prevention (stop bugs before they happen) vs Detection (find bugs after they happen)
  - As developers, we can lessen the likelihood of errors by using software engineering techniques, like modularization, encapsulation, documentation, etc.

Prevention of errors

- Clear Requirements: Many bugs aren't technical; they are misunderstandings of what the software should do. Behavior-Driven Development (BDD) writes requirements in "Given-When-Then" scenarios to aligns designers/product managers, developers, and QA testers before code is even written.
- Design for Testability ("design to test") (DFT) means designing your code to be testable, such as, you architect your software from the ground up in a way that makes it easy to isolate and test.
- Test-Driven Development (TDD): Writing your tests before writing the actual code forces you to think through edge cases, inputs, and outputs ahead of implementation.

Prevention of errors

- Keep It Simple, Stupid (KISS): Avoid over-engineering. Complex code breeds hidden edge cases.
- Don't Repeat Yourself (DRY): Duplicate code means duplicate bugs. If you find a bug in one place, you have to remember to fix it everywhere else it was pasted.
- Avoid "Magic Numbers": use named constants instead of hardcoded strings or numbers.

Prevention of errors

- **Design and Develop for simplicity:**
  - ***Thin interfaces***:
    - Large applications often consist of different modules, so that different teams can work on them
    - The interface between modules must be clearly specified and Tested. Reducing Cognitive Load & Clear Specifications.
    - Each module does not need to know implementation details of the other. Information Hiding (Encapsulation).

Software Testing

- Software Testing goals:
  - Verification: "Are we building the product right?"
    - Did we build the code correctly according to specs?
    - It focuses on code correctness—ensuring your functions, algorithms, and edge cases produce the exact output promised by your technical design without any bugs.
    - When you write a test driver or run unit tests, you are performing testing. When that test confirms your code strictly matches the written specification, design rules, or math formulas, you have achieved verification.
  - Validation: "Are we building the right product?"
    - Catches misunderstood requirements and missing features

Prevention of errors through Testing, Detection of errors through Debugging

- *Testing* searches for the presence of errors.
  - Testing techniques:
    - Unit testing
    - Test automation
- Once errors are found, *Debugging* searches for the source of errors and fixing them.
  - Debugging techniques (covered at the beginning of the semester):
    - Manual walkthroughs
    - Print statements
    - Debuggers

Testing

- *Defensive programming* ensures your code handles bad inputs and unexpected states.
  - Never Trust Outside Input: data coming from users, file reads, or external APIs and it is unverified by default.
    - Always validate and sanitize it before using it in core logic.
  - Fail Fast & Explicitly: If something goes wrong, fail immediately at the boundary rather than letting corrupt data slip deeper into your application where it's much harder to debug.
  - Practice Graceful Degradation: when a failure occurs, choose a sensible fallback (e.g., Exceptions) so the entire application doesn't collapse.

Unit testing

- Each unit of an application may be tested.
  - Methods, classes, and modules (packages in Java).
- Can (and should) be done during development.
  - Finding and fixing errors early
    - Lowers development costs (e.g. programmer time) in long term
  - A test suite is built up.
- For each unit, understand what the unit should do (its contract)
  - You will be looking for violations
    - Use positive tests and negative tests
    - Test boundaries: Zero, One, Full.
      - Example: for sorting, try an empty array, an array of 1 element, an array with lots of elements.

Test automation

- Testing is repetitive (rerun after each update or daily updates)
- ***Regression testing*** involves re-running tests.
  - Human analysis of the results may be still required.
  - Fuller automation: intervention only required if a failure is reported.
    - If tests fail, explore the changes/updates/commits to debug/find the introduction of the error.

Testing

- Integration testing: verifies that components work correctly together when combined.
  - Occurs: Right after Unit Testing (where individual functions/classes are tested in isolation) and before System Testing
  - Consider two separate functions in an e-commerce app:
    - calculateTotal(price, taxRate) — Calculates the total cost.
    - chargeCard(cardDetails, totalAmount) — Charges the customer's credit card.
    - Integration testing: in a processPayment(price, taxRate, cardDetails) pass the output of calculateTotal() into chargeCard() as the totalAmount argument and verify that the credit card get charged correctly.
- System testing:
  - Evaluates the complete, fully integrated application as a whole to ensure it meets specified requirements.
  - Example: trigger the entire checkout flow from start to finish: addToCart() -> calculateTotal() -> chargeCard() -> updateInventory()
    - Does the whole process run successfully without crashing?
    - Is the item deducted from the database inventory?

Pure Java Test Drivers

- Framework-Free Testing - Example: consider a class Calculator to test:

```
public class Calculator {
  public static int max(int num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
  public static double divide(int numerator,
      int denominator) {
    if (denominator == 0)
      throw new IllegalArgumentException(
      "Cannot divide by 0");
    return (double) numerator / denominator;
  }
}
```

```java
public class CalculatorTest {
  public static void main(String[] args) {
    System.out.println("Calculator Tests:\n");
    // Test Case 1: Positive numbers
    testMax(10, 5, 10, "Test Max: First argument larger");
    // Test Case 2: Negative numbers
    testMax(-3, -1, -1, "Test Max: Negative numbers");
    // Test Case 3: Equal numbers
    testMax(7, 7, 7, "Test Max: Both numbers equal");
    // Test Case 4: Division
    testDivide(10, 2, 5.0, "Test Divide: Normal division");
    // Test Case 5: Division by 0
    testDivide(10, 0, 0, "Test Divide: Division by 0 exception");
  }
  public static void testMax(int num1, int num2, int expected, String testName) {
    int actual = Calculator.max(num1, num2);
    if (actual == expected)
      System.out.println("[PASSED] " + testName);
    else
      System.out.println("[FAILED] " + testName + " -> Expected: " + expected
          + ", But Got: " + actual);
  }
  public static void testDivide(int num, int denom, double expected, String testName) {
    try {
      double actual = Calculator.divide(num, denom);
      if (Math.abs(actual - expected) < 0.0001) { // Using a tolerance for doubles
        System.out.println("[PASSED] " + testName);
      else
        System.out.println("[FAILED] " + testName + " -> Expected: " + expected + ", But Got: "
          + actual);
    } catch (IllegalArgumentException e) {
      // The correct exception was caught -> Test Passed
      System.out.println("[PASSED] Test Exception: Correct caught IllegalArgumentException.");
    }
  }
}
```

Assertions

- An assertion is a Java statement that enables you to assert an assumption about your program.
- An assertion contains a Boolean expression that should be true during program execution.
- Assertions can be used to assure program correctness and avoid logic errors.

Declaring Assertions

- An assertion is declared using the Java keyword **`assert`** in JDK 1.5 as follows:

```
    assert assertion;    //OR
    assert assertion : detailMessage;
```

where assertion is a Boolean expression and detailMessage is a primitive-type or an Object value.

Executing Assertions Example

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i==10;
    assert sum>10 && sum<5*10 : "sum is " + sum;
  }
}
```

Running Programs with Assertions

- By default, the assertions are disabled at runtime. To enable it, use the switch –enableassertions, or –ea for short, as follows:

  java –ea AssertionDemo

```
public class AssertionDemo {
  public static void main(String[] args){
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i!=10;
  }
}
```

```
Exception in thread "main" java.lang.AssertionError
at AssertionDemo.main(AssertionDemo.java:7)
```

Running Programs with Assertions

- Enable assertions in Eclipse
  - Right-click your Java file → Select Run As → Run Configurations…
  - Go to the Arguments tab
  - In the VM arguments box, add:
  - -ea
  - Click Run

Executing Assertions

- When an assertion statement is executed, Java evaluates the assertion.
  - If it is false, an AssertionError will be thrown.
  - The AssertionError class has a no-arg constructor and seven overloaded single-argument constructors of type int, long, float, double, boolean, char, and Object.
  - For the first assert statement with no detail message, the no-arg constructor of AssertionError is used.
  - For the second assert statement with a detail message, an appropriate AssertionError constructor is used to match the data type of the message.
  - Since AssertionError is a subclass of Error, when an assertion becomes false, the program displays a message on the console and exits.

Using Exception Handling or Assertions?

- Assertion should not be used to replace exception handling.
  - Exception handling (try-catch) make your program robust against predictable, real-world user errors or environmental issues.
  - Assertions are to assure the correctness of the program.

Using Exception Handling or Assertions?

- Do not use assertions for argument checking in public methods:
  - Valid arguments that may be passed to a public method are considered to be part of the method's contract.
  - The contract must always be obeyed whether assertions are enabled or disabled.
  - For example, the following code in the Circle class should be rewritten using exception handling:

```
public void setRadius(double newRadius) {
   assert newRadius >= 0;
   radius =  newRadius;
}
```

Using Exception Handling or Assertions?

- Use assertions to reaffirm assumptions.
  - This gives you more confidence to assure correctness of the program.
  - A common use of assertions is to replace assumptions with assertions in the code.
  - A good use of assertions is place assertions in a switch statement without a default case. For example:

```
switch (month) {
  case 1: ... ; break;
  case 2: ... ; break;
  ...
  case 12: ... ; break;
  default: assert false : "Invalid month: " + month;
}
```

Java Test Drivers with assertions

- Example: consider a class Calculator to test:

```
public class Calculator {
  public static int max(int num1, int num2) {
    if (num1 > num2)
      return num1;
    else
      return num2;
  }
  public static double divide(int numerator,
      int denominator) {
    if (denominator == 0)
      throw new IllegalArgumentException(
      "Cannot divide by 0");
    return (double) numerator / denominator;
  }
}
```

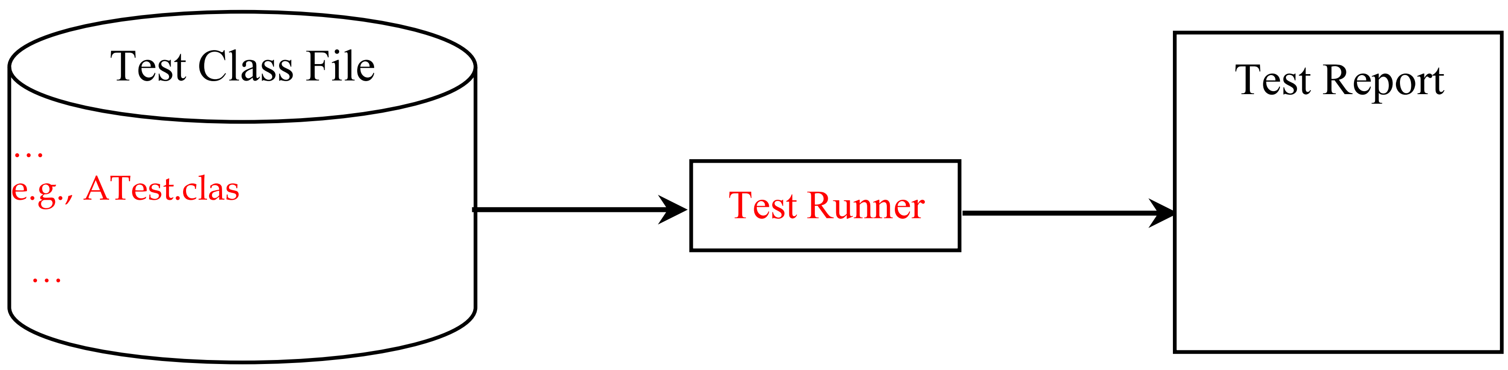
```java
public class CalculatorAssertTest {
  public static void main(String[] args) {
    System.out.println("Running assertions...");

    // Syntax: assert (condition) : "Error message if false";

    // Test Case 1
    assert (Calculator.max(10, 5) == 10) : "Failed: 10 vs 5 should be 10";

    // Test Case 2
    assert (Calculator.max(-3, -1) == -1) : "Failed: -3 vs -1 should be -1";

    // Test Case 3
    assert (Calculator.max(7, 7) == 7) : "Failed: 7 vs 7 should be 7";

    System.out.println("All assertion tests PASSED successfully!");
  }
}
```

Unit Testing with JUnit

- *JUnit* is the de facto framework for testing Java programs
- JUnit is a third-party open source library
- It contains a tool called *test runner*, which is used to run test programs.

Unit Testing with JUnit

- Suppose you have a class named A.
- By convention, if the class to be tested is named A, the test class should be named ATest.
- This test class, called a *test class*, contains the methods you write for testing class A.
- The test runner executes ATest to generate a test report



Using JUnit in Eclipse

- Eclipse come with JUnit (including JUnit 5 and JUnit 6) **pre-bundled.**
  - **Create a Test Class:** In your Package Explorer, right-click on your project or source folder and select **New > JUnit Test Case**.
  - **Select the JUnit Version:** In the wizard that pops up, you will see radio buttons to select the JUnit version on the top - select **New JUnit Jupiter test** for JUnit 5/6
  - In the next screen, select a class that you want to test
  - In the next screen, select **"Add JUnit 5 to build path"**. Eclipse will instantly attach the required libraries to your project.
  - Run Your Tests: Right-click your test file, Select Run As > JUnit Test
    - The JUnit View tab will open up, showing you if your tests pass!

```java
public class A {
  public int add(int a, int b) {
    return a + b;
  }
}

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
class ATest {
  @Test
  @DisplayName("Testing standard addition of two positive integers")
  void testAddPositiveNumbers() {
    // 1. Arrange (Set up your object and data)
    A calculator = new A();
    // 2. Act (Execute the method under test)
    int result = calculator.add(5, 3);
    // 3. Assert (Verify the result matches expectations)
    // assertEquals(expected, actual, optional_failure_message)
    assertEquals(8, result, "5 + 3 should equal 8");
  }
  @Test
  @DisplayName("Testing addition with a negative integer")
  void testAddNegativeNumbers() {
    A calculator = new A();
    assertEquals(-2, calculator.add(3, -5), "3 + (-5) should equal -2");
    // you can add more tests
    assertEquals(0, calculator.add(0, 0), "0 + 0 should equal 0");
  }
}
```

Arrange-Act-Assert (AAA) pattern

- Part of Test-Driven Development (TDD)
- Arrange is the setup phase. Here, you initialize the objects, prepare the data, and configure the environment needed for your test.
  - Goal: Put the system into the exact state required for the test to run.
  - Examples: Creating instances of classes or defining input variables.
- Act is the execution phase. You invoke the specific method or action that you want to test.
- Assert is the verification phase. You check the outcome of the action against your expectations by comparing the returned value to an expected value, checking if an object's state changed, or verifying that a specific side effect occurred (like a file being written).

Exhaustive vs. Sampled Testing

- How do we generate test cases?
  - Exhaustive
    - Consider all possible combinations of inputs
    - Often infeasible
  - Sampled
    - A small but representative subset of all input combinations
      - ***Black-box testing*** - Test cases generated from program specifications and not dependent on the implementation
      - ***Glass-box testing*** - Test cases generated from program's code

Black-box testing

- Test cases should cover all paths (not all cases) through the **specification**, including exceptions.
- Test cases based on program's specification, not on its implementation.
- It is the best place to start when attempting to test a program thoroughly
- Test cases are not affected by:
  - Invalid assumptions made by the programmer
  - Implementation changes
    - Use same test cases even after program structures has changed
- Test cases can be generated by an "independent" agent, unfamiliar with the implementation.

Boundary Conditions

- A boundary condition is an input that is "one away" from producing a different behavior in the program code
- Such checks catch 2 common types of errors:
  - Logical errors, in which a path to handle a special case presented by a boundary condition is omitted
  - Failure to check for conditionals that may cause the underlying language or hardware system to raise an exception (ex: arithmetic overflow)

Boundary Conditions

- Examine the method specifications (preconditions) to generate unique test cases for testing.

```
/* REQUIRES: x >= 0 && y >= 10 */
public static int calc(int x, int y) { … }
```

Translate conditions to test success cases:

```
x =  0, y = 10 (x == 0 && y == 10)
x =  5, y = 10 (x >  0 && y == 10)
x =  0, y = 15 (x == 0 && y >  10)
x =  5, y = 15 (x >  0 && y >  10)
```

Add fail test cases:

```
x = -1, y = 10 (x <  0 && y == 10)
x = -1, y = 15 (x <  0 && y >  10)
x = -1, y =  9 (x <  0 && y <  10)
x =  0, y =  9 (x == 0 && y <  10)
x =  1, y =  9 (x >  0 && y <  10)
```

Boundary Conditions

- A JUnit test class for the boundary conditions for withdrawals from a BankAccount
  - Standard Withdrawal: Arrange an account with $100, withdraw $30. Assert that the method returns true and the new balance is $70.
  - Boundary Condition (Exact Balance): Arrange an account with $100, withdraw exactly $100. Assert that the method returns true and the balance becomes $0.
  - Boundary Condition (Overdraft): Arrange an account with $100, attempt to withdraw $100.01 (one cent over). Assert that the method returns false and the balance remains $100.

```java
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.DisplayName;
class BankAccountTest {
  @Test
  @DisplayName("Withdrawing a valid amount within the balance limit")
  void testStandardWithdrawal() {
    // 1. Arrange
    BankAccount account = new BankAccount(100.0);
    // 2. Act
    boolean result = account.withdraw(30.0);
    // 3. Assert
    assertTrue(result, "Withdrawal should return true");
    assertEquals(70.0, account.getBalance(), "Balance should be 70.0");
   }
  @Test
  @DisplayName("Boundary Condition: Withdrawing the exact total balance")
  void testExactBalanceWithdrawal() {
    BankAccount account = new BankAccount(100.0);
    boolean result = account.withdraw(100.0);
    assertTrue(result, "Withdrawal of exact balance should return true");
    assertEquals(0.0, account.getBalance(), "Balance should drop exactly to 0.0");
  }
  @Test
  @DisplayName("Boundary Condition: Requesting an overdraft by just one cent")
  void testOverdraftBoundary() {
    BankAccount account = new BankAccount(100.0);
    boolean result = account.withdraw(100.01);
    assertFalse(result, "Withdrawal exceeding balance should return false");
    assertEquals(100.0, account.getBalance(), "Balance should remain unchanged at 100.0");
  }
```

Boundary Conditions

- **Test the Edge cases:** Don't just test the exact boundary.
  - If a method checks if someone is a minor under 18, test 17, 18, **and 19**.
    - Bugs almost always hide right at the transition points.

method name / parameters	description
`assertTrue(`***test***`)` `assertTrue("`***message***`",` ***test***`)`	Causes this test method to fail if the given `boolean` test is not `true`.
`assertFalse(`***test***`)` `assertFalse("`***message***`",` ***test***`)`	Causes this test method to fail if the given `boolean` test is not `false`.
`assertEquals(`***expectedValue***`,` ***value***`)` `assertEquals("`***message***`",` ***expectedValue***`,` ***value***`)`	Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the `equals` method to compare them.) The first of the two values is considered to be the result that you expect; the second is the actual result produced by the class under test.
`assertNotEquals(`***value1***`,` ***value2***`)` `assertNotEquals("`***message***`",` ***value1***`,` ***value2***`)`	Causes this test method to fail if the given two values *are* equal to each other. (For objects, it uses the `equals` method to compare them.)
`assertNull(`***value***`)` `assertNull("`***message***`",` ***value***`)`	Causes this test method to fail if the given value is not `null`.
`assertNotNull(`***value***`)` `assertNotNull("`***message***`",` ***value***`)`	Causes this test method to fail if the given value *is* `null`.

`assertSame(`***expectedValue***`,` ***value***`)`
`assertSame("`***message***`",` ***expectedValue***`,` ***value***`)`
`assertNotSame(`***value1***`,` ***value2***`)`
`assertNotSame("`***message***`",` ***value1***`,` ***value2***`)`
	Identical to `assertEquals` and `assertNotEquals` respectively, except that for objects, it uses the `==` operator rather than the `equals` method to compare them. (The difference is that two objects that have the same state might be `equals` to each other, but not `==` to each other. An object is only `==` to itself.)

`fail()` `fail("`***message***`")`	Causes this test method to fail.

Notes on Static import

```
import static org.junit.jupiter.api.Assertions.*;
```

- Static import is a feature introduced in the Java programming language that allows members (fields and methods) defined in a class as public static to be used in Java code without specifying the class in which the field is defined.
- The mechanism can be used to reference individual members of a class:

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
```

- or all the static members of a class:

```
import static java.lang.Math.*;
```

Static import example

```
import static java.lang.Math.*;
// OR
// import static java.lang.Math.PI;
// import static java.lang.Math.pow;

import static java.lang.System.out;

public class HelloWorld {
    public static void main(String[] args) {
        out.println("Hello World!");
        out.println("A circle with a diameter of 5 cm has:");
        out.println("A circumference of " + (PI * 5) + " cm");
        out.println("And an area of " + (PI * pow(2.5,2))
          + " sq. cm");
    }
}
```

Glass-box testing

- Black-box testing is generally not enough.
- For Glass-box testing, the code of a program being tested is taken into account
- Path-completeness:
  - Test cases are generated to exercise each path through a program.
  - May be insufficient to catch all errors, but you can compare the test cases with the ones obtained from Black-box testing
    - Infer if tests developed at the design phase were incomplete or implementation of all cases is incomplete
  - Can be used effectively only for a program fragment that contains a reasonable number of paths to test.

Glass-box testing

- Consider a method that calculates a user's shipping discount based on their loyalty tier and order total:

```
public double calculateShipping(String tier, double totalOrder) {
  if ("Platinum".equals(tier)) {
    return 0.00; // Free shipping for Platinum
  } else if ("Gold".equals(tier)) {
    if (totalOrder >= 50.00)
      return 0.00; // Free shipping for Gold if order is $50+
    else
      return 5.00;  // $5 shipping for Gold under $50
  } else
    return 10.00; // Flat $10 shipping for everyone else
}
```

Glass-box testing

- How to Perform the Structural Analysis:
  - Identify Decision Points: Look for every conditional operator or control block (if, switch, loops).
  - Trace Combinations: Chart every unique outcome of those decisions. For example, if you have a nested if structure, you must account for when the outer condition is true/false, and when the inner conditions are true/false.
  - Find the Input Configurations: Work backward from the path to deduce the exact input values needed to force the execution stream down that specific logical line.

Glass-box testing

- Structural Analysis & Path Completeness Table map out every single independent path through a program fragment to achieve path-completeness meaning that test cases are systematically generated to exercise every possible branch route from the method's entry point to its exit point.
  - Path ID: A unique identifier (e.g., Path A, Path 1) for tracking and cross-referencing with automated test cases.
  - Input Variables: Separate columns for every parameter or configuration variable that influences an if, else if, else, or loop condition.
  - Expected Output: The exact return value, object state change, or exception that the code is structured to produce when following that specific path.
  - Core Branch Condition / Internal Logic Path: A brief explanation detailing exactly which internal branch conditions are evaluated as true or false to route execution along that path.

Glass-box testing

- Structural Analysis & Path Completeness Table

Path ID	Input tier	Input totalOrder	Expected Return Value	Core Branch Condition / Internal Logic Path
Path 1	"Platinum"	30.00 (any value)	0.00	Evaluates the first if statement as true. Returns free shipping immediately.
Path 2	"Gold"	60.00	0.00	First if is false, else if ("Gold") is true. Inner boundary condition totalOrder >= 50.00 evaluates to true.
Path 3	"Gold"	20.00	5.00	First if is false, else if ("Gold") is true. Inner boundary condition totalOrder >= 50.00 evaluates to false (else branch).
Path 4	"Regular" (or any other)	100.00 (any value)	10.00	All previous if and else if checks evaluate to false. Execution falls entirely through to the final catch-all else block.

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
public class ShippingCalculatorTest {
  private ShippingCalculator calculator = new ShippingCalculator();
  @Test
  public void testPlatinumTier_GetsFreeShipping() {
    // Arrange
    String tier = "Platinum";
    double totalOrder = 30.00;
    // Act
    double actualShipping = calculator.calculateShipping(tier, totalOrder);
    // Assert
    assertEquals(0.00, actualShipping, 0.001);
  }
  @Test
  public void testGoldTier_WithOrderOverFifty_GetsFreeShipping() {
    String tier = "Gold";
    double totalOrder = 60.00;
    double actualShipping = calculator.calculateShipping(tier, totalOrder);
    assertEquals(0.00, actualShipping, 0.001);
  }
  @Test
  public void testGoldTier_WithOrderUnderFifty_PaysFiveDollars() {
    String tier = "Gold";
    double totalOrder = 20.00;
    double actualShipping = calculator.calculateShipping(tier, totalOrder);
    assertEquals(5.00, actualShipping, 0.001);
  }
  @Test
  public void testSilverTier_PaysFlatTenDollars() {
    String tier = "Silver";
    double totalOrder = 100.00;
    double actualShipping = calculator.calculateShipping(tier, totalOrder);
    assertEquals(10.00, actualShipping, 0.001);
  }
}
```

Glass-box testing

- Why 0.001 is inside the assertEquals?

```
assertEquals(0.00, actualShipping, 0.001);
```

  - When testing double or float types in Java, you should always pass a third argument called a delta (tolerance). Because computers handle floating-point math with minor precision quirks, checking if a double exactly equals another double can cause flaky tests. 0.001 means "as long as the numbers are within $0.001 of each other, pass the test."

Glass-box testing

- Why Use a Path Completeness Table?
  - Discovers Hidden Logic Errors: It catches edge cases where a particular path or handling of a special condition was omitted by the programmer.
  - Ensures Code Coverage: It ensures that no dead code blocks exist and that every line of code written is verified at least once.