Foundations of Software Testing

2.5 Test-Driven Development (TDD) Basics

Learning objectives

- What is Test-Driven Development (TDD)?
- The Red → Green → Refactor cycle
- Why TDD improves code quality
- Step-by-step TDD example using Calculator

What Is Test-Driven Development

- TDD is a software design approach where tests are written before code
- Focus on small, incremental steps
- Goal: drive the implementation through tests
- Each test defines expected behavior
- Ensures immediate validation of logic
- Promotes cleaner, more maintainable code

The TDD Cycle

- Three simple steps:
- 1. 🔴 Red – Write a test that fails (no implementation yet)
- 2. 🟢 Green – Write minimal code to make it pass
- 3. 🔵 Refactor – Clean up both code and tests
- Repeat for every small piece of functionality.

Why Practice TDD

- Encourages thinking about design first
- Ensures test coverage grows naturally
- Makes refactoring safer
- Catches logic errors early
- Builds confidence in code changes

Example: Adding a power() Method (Step 1 – Red)

- Scenario: Extend our Calculator with a new feature: raising a number to a power.
- 1. Write a failing test first:
  - The method power(int base, int exponent) should compute base^exponent.
- 2. The test will fail since the method doesn’t exist yet.
- Talking Point:
  - “In TDD, failing first means we’re defining what we want – not guessing after implementation.”

Example: Step 2 – Green

- Now create a minimal implementation to make the test pass.
- Don’t optimize or over-engineer.
- Just return the correct result for the tested case.
- Example: using Math.pow() is fine for now.

Example: Step 3 – Refactor

- Once the test passes, clean up your code.
- Remove duplication or simplify logic.
- Ensure all tests still pass after refactoring.
- Keep your test names descriptive and focused.

Demo: Implementing Power Function with TDD

- 1. Start with the existing Calculator project from Module 2.4.
- 2. Add a new test first in CalculatorTest to test the power method:
  - assertEquals(8.0, calculator.power(2, 3));
- 3. You need to create the power method for the code to compile, but keep it just as simple as possible:
  - Use the correct method signature, but just return zero
- 4. Run the test and you have completed step 1 - Red.

Demo: Implementing Power Function with TDD

- 5. Create a working implementation for the power method
- 6. Run the test and you have completed step 2 - Green.
- 7. Refactor if necessary and you have completed step 3 - Refactor.

Lab: Building Features with TDD

- Goal: Apply TDD by adding new operations step-by-step.
- 1. Continue in the existing project demo project with the Calculator.
- 2. Choose one or more new features for your Calculator, e.g.:
    - modulus(int a, int b) – remainder after division
    - isEven(int number) – returns true/false
    - absolute(int number) – returns positive value
- For each:
    - Write a failing test first.
    - Implement just enough code to make it pass.
    - Once the test pass, refactor the code.

Lab: Reflection

- How did starting with tests change your thinking?
- Did you ever write too much code before running tests?
- How did it feel to see red → green → refactor?
- What challenges did you face designing testable code?

Key Takeaways

- TDD drives design through tests, not code guesses
- Red → Green → Refactor keeps focus on small steps
- Tests define behavior, not implementation
- Writing failing tests first clarifies requirements
- You can use TDD for logic, APIs, and even REST endpoints later