How to TDD

Summary of TDD Process

- Start with the simplest failing test
- Write simplest code to pass the test quickly
- Move on to next failing test if no refactoring needed
- Refactor test code as well
- Use parameterized tests to consolidate similar test methods
- Allow duplication if it improves test readability
- One test method per distinct rule, name tests to convey rules clearly
- Tests should read like a specification
- Localize dependencies on objects under test
- Done when no more failing tests can be thought of
- TDD is a design discovery process
- Tests make changes safer and easier

Exercises

- Exercise #1
  - Test-drive code generating prime numbers less than 1,000
  - Test-drive code converting integers 1 to 4,000 into Roman Numerals
- Exercise #2
  - Test-drive code calculating total net value of shopping cart items
  - Items represented as list of unit price and quantity
  - Apply discounts:
    - 5% discount if gross value > £100
    - 10% discount if gross value > £200

TDD Philosophy

- Baby steps and many tests may seem overkill
- Upfront investment pays off by reducing future costs of change

Improving Performance

- Problem
  - Recursive solution is slow for higher indexes due to repeated calculations
- Iterative Solution
  - Use array to store sequence values
  - Calculate iteratively to reuse previously computed numbers
- Safer to change with existing automated tests

Benefits of TDD

- Enables safe and easy code changes
- Provides fast-running automated tests
- Helps discover design through incremental examples

Example: Fibonacci Sequence Calculator

- Fibonacci Sequence Definition
  - Starts with 0 and 1
  - Each subsequent number is sum of previous two
  - Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, etc.
- Step 1: First Failing Test
  - Test that first number is zero
  - Simplest code to pass: always return 0
- Step 2: Second Failing Test
  - Test that second number is one
  - Simplest code to pass both: return index
- Step 3: Parameterized Test for First Two Numbers
  - Combine tests for index 0 and 1 using parameterized test
  - Use JUnitParams for easier parameterized tests
- Step 4: Third Failing Test
  - Test that third number is one
  - Simplest code to pass: if index < 2 return index else return 1
  - Keep separate test method for this rule for clarity
- Step 5: Refactor Test Code to Reduce Duplication
  - Extract method to get Fibonacci number to centralize object creation
  - Limit test knowledge of object interface
- Step 6: Fourth Failing Test
  - Test that fourth number is two
  - Simplest code to pass: return index - 1 for index >= 2
- Step 7: Parameterized Test for Third and Fourth Numbers
  - Combine tests for index 2 and 3 with a parameterized test
- Step 8: Fifth Failing Test
  - Test that sixth number is five
  - Simplest code to pass: recursive call summing previous two numbers
- Step 9: Merge and Rename Parameterized Test for Indices >= 2
  - Merge tests for indices 2, 3, and 5
  - Rename test method to reflect sum of previous two rule
- Step 10: Sixth Failing Test - Negative Index
  - Test that negative index throws IllegalArgumentException
  - Add input validation in implementation
- Final Implementation
  - Recursive Fibonacci with input validation
  - Tests serve as clear specification with distinct rules communicated via test names