Open Source Training

978-250-4983

www.open-source-training-courses.com

# Test Driven Development (TDD), and Refactoring Legacy Code Using Java

Duration: 4 Days | Price: $2095

Description: This course provides students with hands on experience learning Test Driven Development (TDD) using JUnit. Students will build unit tests using mocks, fakes, stubs and drivers, and address issues working with databases and other systems. Student will create tests and code that will be more likely to meet and exceed requirements. Code that receives “test coverage” will not break existing systems, because tests are passed before code is checked in.

Students will spend time working with the issues involved in refactoring legacy code, safely cutting into an already deployed system. Students will work on looking for, or creating “seams” to more safely improve code or add features, and work on identifying “code smells” that need attention in a productive system.

Finally, students will explore dependency issues as well as techniques to better understand and improve complex systems.

Students will also examine TDD and refactoring legacy code in other languages like C# to gain a broader view of options and issues working in a multi-language shop. Comprehensive labs using Java provide facilitated hands on practice crucial to developing competence and confidence with the new skills being learned.

Prerequisites: Java SE programming experience and an understanding of object-oriented design principles. HOTT's Java Programming course or equivalent knowledge provides a solid foundation.

## Overview of Topics Covered:

### Why TDD? Think Twice, Write Production Code Once

- Utilizing a Safety Net of Automated Testing
- Agile Development Concepts
- Eliminating Bugs Early
- Smoothing Out Production Rollouts
- Writing Code Faster via Testing
- Reducing Technical Debt
- Practicing Emergent Design
- Making Changes More Safe
- The Importance of Regression Testing

### Basic Unit Testing

- JUnit
- 3.x vs 4.0 JUnit Testing
- Adding Complexity to Initial Simple Tests
- Making Tests Easy to Run
- The TDD Pattern: Red, Green Refactor
- Using Methods of the Assert Class
- Boundary Testing
- Unit Test Limitations

### Comprehensive Unit Testing Concepts

- Using Declarative-Style Attributes
- Using Hamcrest Matchers for More Complex Scenarios
- Using Test Categories

### Mocks, Fakes, Stubs and Drivers

- TDD Development Patterns
- Naming Conventions for Better Code
- Using Mock Objects
- Using Fakes

Exception Handling in Tests
- JUnit Test Initialization and Clean Up Methods
- Writing Clean and Dirty Tests
- Testing with Collections, Generics and Arrays
- Negative Testing

## Database Unit Testing

- Mocking the Data Layer
- Identifying what Should Be Tested in Databases
- Stored Procedure Tests
- Schema Testing
- Using NDbUnit to Set Up the DB Test Environment

## Patterns and Anti-Patterns in TDD

- The SOLID Principles
- Factory Methods
- Coding to Interface References
- Checking Parameters for Validity Test
- Open/Closed Principle: Open to Extension, Closed to Change
- Breaking Out Method/Object
- Extract and Override Call
- Extract and Override Factory Method
- Singleton Pattern
- Decorator Pattern
- Facade Pattern
- State Pattern
- MVP, MVC and MVVM Patterns
- Finding and Removing Code Smells/Antipatterns

## Code Coverage

- White Box vs Black Box Testing
- Planning to Increase Code Coverage Over Time
  - Goal 80% or More Test Coverage
  - Statement Coverage

Using Stubs
- Test Doubles
- Manual Mocking
- Mocking with a Mock Framework
- Self-Shunt Pattern

## Refactoring Basics

- Refactoring Existing Code
- Restructuring
- Extracting Methods
- Removing Duplication
- Reducing Coupling
- Division of Responsibilities
- Improving Clarity and Maintainability
- Test First - then Refactor
- More Complex Refactoring Considerations

## Refactoring Legacy Code

- Reducing Risk of Change
  - Eliminating Dependencies
  - Characterization Tests as a Safety Net
  - Introducing Abstractions to Break Dependencies
- Analyzing Legacy Code
  - Identifying Pinch Points with Effect Analysis
  - Identifying Seams for Expansion and Testing
  - Listing Markup
- Minimizing Risk of Adding New Behavior
  - Sprout Method
  - Sprout Class
  - Wrap Method
  - Wrap Class
- Dealing with Code that's Difficult to Test
  - Globals and Singletons in Tests
  - Inaccessible Methods and Fields
- Using Smells to Identify What to Refactor
  - Dealing with Monster Methods
  - Dealing with Excessively Complex, Large Classes
  - Identifying and Eliminating Duplication
  - Other Smells
- Dealing with Large Legacy Systems
  - Preserving Signatures

## System, Regression and Acceptance Testing

- Statistical Sampling
- Usability Testing
- Test Protocols
- Regression Testing

    - Condition Coverage
    - Path Coverage
- Test Harnesses
- Unit-Testing Harnesses

## Continuous Integration Servers/Automated Testing

- Early Warning of Conflicts
- Metrics and Tools
- Checking into Repository
    - Team Foundation Server (TFS)
    - Subversion
- Continuous Integration Servers
    - CruiseControl.NET
    - Hudson
- Automating the Build/Deployment
- Acceptance Testing

## Risks Changing Legacy/Production Systems

- Refactoring
- Coupling and Cohesion Issues
- Taking Small Tested Steps
- Anti-Pattern: Big Bang, Boom