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

Duration: 4 Days *(Face-to-Face & Remote-Live)*, or 28 Hours *(On-Demand)*

Price: $2095 *(Face-to-Face & Remote-Live)*, or $1495 *(On-Demand)*

Discounts: We offer multiple discount options. Click here for more information.

Delivery Options: Attend face-to-face in the classroom, remote-live or on-demand training.

Students Will Learn

- Agile development and the test-driven development paradigm
- Creating tests from use cases and/or Agile methodology
- Unit testing using JUnit
- Testing code that interacts with databases
- Using mocks, fakes, and stubs
- Automating tests, builds and check-ins using a continuous integration server
- Refactoring existing code to improve clarity, readability and maintainability
- Identifying patterns useful in TDD including the SOLID principles
- Identifying and eliminating dependencies that make code difficult to maintain and extend
- Tracking code coverage and analyzing other code metrics to improve code maintainability
- Using the seam model to identify appropriate places in the code to make changes safely
- Identifying and correcting various types of code smells
- Using effect sketches and pinch points to identify optimal places for tests
- Using feature sketches to identify opportunistic refactoring

Course 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.

Comprehensive Java labs throughout the course provide facilitated hands on practice that is crucial to developing competence and confidence with the new skills being learned.

Course Prerequisites

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

Course Overview

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

Mocks, Fakes, Stubs and Drivers

- TDD Development Patterns
- Naming Conventions for Better Code
- Using Mock Objects
- Using Fakes
- Using Stubs
- Test Doubles
- Manual Mocking
- Mocking with a Mock Framework
- Self-Shunt Pattern

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

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

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

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

Code Coverage

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

Risks Changing Legacy/Production Systems

- Refactoring
- Coupling and Cohesion Issues
- Taking Small Tested Steps