Core Test-Driven Development using JUnit (3 days)

Core Test Driven Development using JUnit **is a three-day, comprehensive hands-on test-driven development / JUnit / TDD training course geared for developers who need to get up and running with essential Test-driven development programming skills using JUnit and various open-source testing frameworks. Throughout the course students learn the best practices for writing great programs in Java, using test-driven development techniques. This comprehensive course also covers essential TDD topics and skills.**

Students who attend **Core Test-driven Development Using JUnit** will leave the course armed with the skills they require to develop solid Java programs, using sound coding testing techniques and best coding practices.

This course quickly introduces developers to the concepts of Agile Programming and how continuous testing is an integral part of those concepts. Developers are then shown the features of JUnit and educates them regarding JUnit's strengths and weaknesses. JUnit makes it possible to write higher-quality Java code. It is a powerful tool designed to support robust, predictable and automated testing development in the Java enterprise application arena.

Working within in a dynamic, learning environment, guided by our expert TDD team, attendees will:

- Understand JUnit.
- Understand and use the JUnit Test Runner interface.
- Use JUnit to drive the implementation of Java code.
- Test applications using native IDE support.
- Best practices and patterns for test development.
- Understand JUnit's strengths and weaknesses
- Understand the role of debugging when done in conjunction with tests.
- Understand not only the fundamentals of the TDD using Java, but also its importance, uses, strengths and weaknesses.
- Learn to better control the development and quality of Java code.
- Understand how JUnit affects your perspective on development and increases your focus on a task.
- Learn good JUnit coding style.
- Create well structured JUnit programs.
- Look at refactoring techniques available to make code as reusable/robust as possible.
- Discuss various testing techniques

The following JUnit-based testing frameworks are examined:

- JUnit 4.x
- EasyMock

Workshop Topics Covered / Course Syllabus

Session: Introducing Test-Driven Development (TDD)

Lesson: Test-Driven Development

- Overview of Test-driven Development
- Test, code, refactor, repeat
- The ROI of TDD
    - Rationale for Test-driven Development
    - The Process of TDD
    - Advantages to TDD
    - Side-effects of TDD
    - Observations About Tests
- Rationale
- Advantages
- Tools

Session: JUnit

Lesson: JUnit Overview

- What is Unit Testing?
- Purpose of Unit Testing
- Successful Unit Testing
- Good Unit Tests
- Test Stages
- Unit Test Stage
- Integration Test Stage
- Unit Testing Vs Integration Testing
- Functional Testing
- Non-Functional Testing

Lesson: Jumpstart: JUnit 4.x

- Understanding Unit Testing Frameworks
- JUnit Overview
- JUnit Design Goals
- JUnit Features
- Reasons to Use JUnit
- How JUnit Works
- Class to be Tested
- Test Case using JUnit
- Exploring JUnit
- Writing the TestCase
- Test Result Verification (Assertions)
- Assert
- Launching Tests
- Failures vs. Errors

- Introducing Class Message
- Creating Class MessageTest
- The First Test Implementation Steps
- The Initial Test Code
- Testing the Constructor
- Running a Test in an IDE
- Running a Test From the Command Line
- Seeing Results of a Test: JUnit View
- Using the Results of a Test
- Seeing Results of a Successful Test
- Test Suites
- Creating a Test Suite
- Composing Tests Using Suite
- JUnit Test Fixture
- Managing Resources with Fixtures
- Share Similar Objects
- Share Expensive Setups
- JUnit Method Lifecycle
- JUnit: Resources

Lesson: @Test Annotation

- Test Execution Cycle
- Checking for Exceptions
- Limitation of Expected Parameter
- Testing for an Expected Exception
- Testing Using a Timeout
- Using Timeouts
- Test Annotation: Resources

Lesson: Hamcrest

- About Hamcrest
- junit.assert.Assert.assertThat(...)
- The Hamcrest Matcher Framework
- Using assertThat
- Hamcrest Matchers - Logical
- Hamcrest Matchers - Object
- Hamcrest Matchers - Number
- Hamcrest Matchers - Collections
- Additional Hamcrest Matchers
- Hamcrest: Resources

Lesson: Parameterized Tests

- Parameterize a Test Case
- Injecting the Parameters
- Setting the Parameters
- Write the Test Method
- Writing a Parameterized Test
- Test Execution Cycle
- Observations
- Parameterized Tests: Resources

Lesson: Theories

- Writing Theory Enabled Tests
- Defining DataPoints
- Defining Theories
- Test Execution Cycle
- Observations
- Theories: Resources

Lesson: JUnit Best Practices

- So What is a "Good" Test?
- Good: Readable Equates to Maintainable
- Good: Proper Organization and Structure
- Good: Test the Right Thing
- Good: Run in Solitude
- Practices to Reduce and Manage Dependences
- Good: Reliability
- Importance of Quality of Assertions
- Bad Smell: Primitive Assertions
- Bad Smell: Broad Assertion
- Bad Smell: Hidden Beef
- Bad Smell: Split Personality
- Bad Smell: Split Logic
- Managing Data
- Bad Smell: Parameterized Mess
- Coding Practices
- Legacy Code
- Preparing Unit Test Environment
- Stubs -> Mocks
- White-Box Unit Testing
- Black-Box Unit Testing
- Keys to Success
- Boundary Between Unit and Integration Testing
- Integration Testing
- Purpose of Integration Testing
- Who Does Integration Testing
- The Lowest Bar for Unit Testing
- Automated Testing
- Automation and Coverage
- Working With Coverage Analysis

Session: Advanced Topics

Lesson: Mocking of Components

- Why We use Test Dummies
- Isolation
- Improving Speed and Reliability
- Handling Special Conditions and Hidden Data
- Types of Test Dummies
- Stubs
- Mock Objects
- Working with Mock Objects
- Challenges of Testing User Interfaces
- Using Mocks with the User Interface
- Mock Object Strategies

Lesson: Mock Objects and EasyMock

- EasyMock Description and Features
- EasyMock Object Lifecycle
- Types of Mock Objects (in EasyMock)
- Create Phase: Creating EasyMock Mock Objects
- Create Phase: Creating Mock Objects Directly
- Create Phase: Creating Mocks Using a Control
- Expect Phase: Recording Methods Returning Void
- Expect Phase: Record Methods Returning Value
- Expect Phase: Record Throwing Exception
- Expect Phase: Call Repetition
- Expect Phase: Matchers
- Replay Phase: Preparing Mock Objects
- Verify Phase: Checking that Expectations are Met
- Mocking Complex Objects
- Testing with Dependencies
- Decoupling Dependent Classes
- Testing with Mocks
- EasyMock HOWTO

Lesson: Improving Code Quality Through Refactoring

- Refactoring Overview
- Sample of Refactorings
- Refactoring and Testing
- Suggested Refactoring
- The Impact of Refactoring
- Refactoring to Design Patterns
- Sample Refactorings
- Best Practices
- Refactoring
- Naming conventions
- Organizing test suites

TT3510/j13