Quality Development Testing and TDD for Java

**3 days duration**

**Course Description**

This course covers Test Driven Development and Behavior Driven Development (BDD) concepts of “write your tests first” and “fail first”. It will illustrate how you can drive your development process through “test-implement-test pass-refactor” cycles. Students will see the link between Agile Stories and Acceptance Test Driven Development (ATDD) through the definition of Acceptance criteria that can then be implemented via Cucumber-JVM together with JUnit. Through a series of practical exercises, students will translate scenarios into runnable acceptance tests using Cucumber-JVM. As such various testing patterns and approaches will be explored including the utilization of isolated testing via Stubs, Fakes and Mock objects. Students will leverage the evolution of User Stories and Business Flows into code in order to test Web Applications via the Selenium API. This course is about 50% lecture & 50% discussion / hands-on exercises.

**Objectives**

* Capture the core JUnit syntax
* Use BDD and the test-fail-first approach
* Translate business requirements into user stories with acceptance tests
* Learn how to Refactor to clean up your code
* What makes your code testable
* Use Fakes and Mocks for isolated unit tests
* Use Cucumber-JVM to turn User Stores into Code
* Explore the Selenium API

**Audience**

This training session is for developers, testers, business analysts and product owners, to enable them to apply BDD principles and practices to steer their projects with acceptance tests at the level of requirements and user stories.

**Prerequisites**

It is recommended that participants have experience working in Java.

**Course Details**

1. How Stories and Testing fit into the Agile Process

* What is Agile?
* The Role of the Product Owner
* Criteria for evaluating stories
* The Testing Pyramid
* Agile Ttesting Quadrants

1. JUnit Overview/ Primer

* What is Unit Testing?
* JUnit, the test lifecycle - @Before, @After, @BeforeClass, @AfterClass, @Test
* Writing assert methods – assertEquals, assertSame, assertTrue, assertFalse, assertNull
* Hamcrest Matchers and using assertThat
* Parameterized and Theory Testing
* Code Coverage

1. Test Driven Development (TDD)
   * Principals and Techniques
   * TDD Metaphors
   * Benefits, Challenges and Limitations
   * Handling Requirements Change
   * Characteristics of good tests
   * Revisit Anti Patterns
2. Testable Designs (Mocks, Fakes and Stubs)

* Creating testable Code, If you cannot test it what use is it?
* Strategies for Testable Code
* Test Unfriendly features
* Stubs, Fakes and Mocks
* Mocks as Collaborators
* Mocks and return values, void methods, frequency calls and ordering
* Using the EasyMock and Mockito testing frameworks

1. Behavior Driven Development (BDD) with Cucumber

* Writing User Stories
* Acceptance criteria
* Acceptance Test Driven Development (ATDD), Specification By Example
  + Creating software based on realistic examples,
  + Bridging the communication gap between business stakeholders
* Introduction to Cucumber
* Writing scenarios with Gherkin
  + Gherkin syntax
  + Using given, when, then
  + Java Fixtures generated from scenarios
* Using JUnit to run your Cucumber stories and scenarios
* Writing set-up and tear down code using hooks (@Before and @After)
* Using Data Tables, Scenario Outlines
* Automate Cucumber steps to drive your application through and below the user interface
* Refactor Cucumber step definitions to make them more readable and maintainable
* Cucumber Reports

1. Testing Web sites with Selenium with Cucumber

* Using a WebDriver to test web applications and the Selenium API
* Interrogating a response page
* Simulating links and form submissions
* Simulate multiple page navigation
* Continuous Integration

A 4 Day version is as above except that between the sections on TDD and Testable Designs (sections 3 and 4) we would focus on Refactoring Techniques

To quote Richard Helm of the Gang of Four “The goal for most software developers still remains to design for change—and there the debate is do you do it early (given foreknowledge) or later (once more is known and you know you need it)? In many cases the design should be refactored, and the patterns provide a target to do this.”

1. Refactoring to Patterns

* Code Smells
  + Bloaters
  + Abusers
  + Change Preventers
  + Dispensible
  + Tight Coupling
* SOLID Principals of OO Software Development
* Simple Refactoring Strategies
* Critical Design Patterns
  + Factories, Factory Method, Abstract Factory
  + Adapter
  + Façade
  + Strategy
  + Proxy and Dynamic Proxy