

Test Driven Development and Refactoring

Course Summary

Description

This is a practical and hands on workshop that focuses on two core Agile practices; refactoring and test driven development (TDD). Agile software development methods require a high degree of skill in keeping code robust as it evolves incrementally through iterations and changes in response to feedback from stakeholders and users.

TDD is an Agile practice to support continuous testing of development code using tools such as JUnit and XUnit, which are used to develop "self-testing" code for unit testing, and FitNesse, which is used for acceptance testing. By following the principle of writing the tests first, design and development are driven to cleaner architectures that meet the user expectations and keep code clean and easier to maintain.

Refactoring is the practice of keeping the code base clean and robust by changing it without harming it – changing its structure without causing unexpected changes in its functionality. Refactoring works in conjunction with TDD in the Agile methods by allowing test results from the TDD process to drive code changes and, in turn uses TDD to ensure that refactorings are robust.

Topics

- Recognize poor code constructs
- Improve designs in small safe steps
- Use the xUnit framework (JUnit, NUnit, or CppUnitLite)
- Use "intention" to drive object interface design
- Perform test -first programming of object clusters
- Know how and when to use Mock objects and other testing patterns
- Know how and when to refactor when growing systems test-first
- Write executable requirements
- Understand Unit and Acceptance Tests
- Write Acceptance Tests using FitNesse
- Keep the system running

Audience

This course is designed for experienced OO programmers.

Prerequisites

Students should have experience writing C++ or Java code. Experience with testing concepts is useful but not necessary.

Duration

Three days

Test Driven Development and Refactoring

Course Outline

- I. Agile Development Overview**
 - A. Basic concepts of Agile development
 - B. Extreme programming principles
 - C. Developing Iteratively and incrementally
 - D. Informal and continuous design
- II. TDD: why write tests first**
 - A. Refactoring as controlled code changes
 - B. TDD and Testing Concepts Overview
 - C. Traditional software testing
 - D. Core testing best practices
 - E. Unit, component, system, acceptance and regression testing
 - F. and regression testing
 - G. TDD: testing early, often and automatically
 - H. TDD: driving development through testing
 - I. Testing is not debugging
 - J. White-box and black-box testing
 - K. Functional versus operational testing
 - L. Executable requirements
- III. Unit Testing**
 - A. Defining units to test
 - B. Interface and implementation
 - C. Using assertions
 - D. Testing units at the interface
 - E. Organizing unit tests – simplest first
 - F. Incremental testing
 - G. Unit test coverage
 - H. Using unit tests to understand a system
- IV. Working with JUnit/Xunit**
 - A. The JUnit and the xUnit framework and family
 - B. Setting up and using JUnit
 - C. Framework structure
 - D. Test cases, suites and runners
 - E. Assertion methods
 - F. Testing exceptions
 - G. Defining common fixture code
 - H. JUnit pattern usage
 - I. Extending the framework
- V. Writing Automated JUnit Tests**
 - A. Writing good test cases
 - B. Deciding what test cases need to be written
 - C. Keeping the test cases organized and documented
 - D. XUnit/JUnit best practices
 - E. Data-driven testing approaches
 - F. Using the test results and feedback
- VI. The TDD Process**
 - A. Red, green, refactor
 - B. Declare, prepare, assert
 - C. Testing by method, state and scenario
 - D. Testing patterns
 - E. Compile-time constraints
 - F. Executing full test suites
 - G. Testing object clusters
- VII. Mock Objects**
 - A. What mock objects are
 - B. When to use and not use mock objects
 - C. Using mock object effectively
 - D. Using mock object tools
- VIII. Refactoring Overview**
 - A. Changing structure without changing functionality
 - B. When to refactor
 - C. Refactoring versus code changes
 - D. Refactoring driver – code smells
 - E. Refactoring for simplicity in design and style
 - F. Refactoring to good design practices – increasing cohesion and reducing coupling
 - G. Refactoring to patterns
 - H. Refactoring for testing
- IX. Refactoring Patterns**
 - A. Renaming variables, methods, classes and packages
 - B. Restructuring class hierarchies
 - C. Partitioning classes
 - D. Changing implementation representations
 - E. Refactoring for decoupling
 - F. Refactoring to break dependencies
 - G. Basic refactoring guidelines
 - H. Refactoring patterns and catalog
- X. Acceptance Testing**
 - A. The importance of acceptance testing
 - B. Acceptance testing and user stories
 - C. Acceptance testing and use cases
 - D. Using FitNesse for acceptance testing
- XI. Overview of Other Tools**
 - A. DbUnit: Testing the database layer
 - B. HtmlUnit: Testing web applications
 - C. Cactus: Testing server-side J2EE components
 - D. JunitPerf: Continuous performance testing