

Implementing Agile Test Driven Development for Java Developers (TT3535)

Duration: 3 days

Course Description:

Test-driven development (TDD) is an evolutionary approach to development where you must first write a test that fails before you write new functional code. This process was developed by Kent Beck and Ward Cunningham. It is primarily an Agile approach to software development and is one of the core principles of Extreme Programming.

Audience:

Experienced Java programmers

Prerequisites:

This is an **intermediary** course, designed for software developers. A working knowledge of Java is required. Knowledge of current development processes, such as structured top-down development and the waterfall method is beneficial.

Topics

SESSION: AGILE DEVELOPMENT

LESSON: AGILE SOFTWARE DEVELOPMENT

- What This Course is Really About
- Controlling Risk
- Agile Development
- Motivation: Risk Reduction
- The Discipline of Timeboxing
- Incremental Delivery and Evaluation

LESSON: AGILE SOFTWARE DEVELOPMENT MANIFESTO

- Same Approach with Documentation
- Change Wins Out Over Following a Set Plan
- Refactoring is Artifact of Change
- Rules
- Extreme Values
- The XP Practices
- Continuous Integration (CI)
- Agile Testing
- The Lowest Bar of Unit Testing
- Agile Testing Stages
- Test First

- Acceptance Tests
- Test-Driven Development
- General Agile Principles
- Adopting or Trying Agile
- Setting User Expectations

SESSION: TEST-DRIVEN DEVELOPMENT

LESSON: UNIT TESTING

- What is Unit Testing?
- Purpose of Unit Testing
- Successful Unit Testing
- Unit Testing Frameworks
- XUnit: JUnit, NUnit, etc.
- Reasons to Use XUnit
- How XUnit works
- Lesson The ROI of TDD
- Rationale for Test-driven Development
- The Process of TDD
- Advantages to TDD
- Side-effects of TDD
- Observations About Tests
- Tools to support TDD

Implementing Agile Test Driven Development for Java Developers (TT3535)

- Automation and Coverage
- Working With Coverage Analysis
- The Concept of Test “Close” Development
- Dependencies vs. mock objects
- Interaction-based Testing
- JUnit and Ant
- Running JUnit Tests from Ant
- Generating a JUnitReport

SESSION: IMPROVING CODE QUALITY THROUGH REFACTORING

LESSON: REFACTORING

- Refactoring Overview
- Sample of Refactorings
- Refactoring and Testing
- Suggested Refactoring
- The Impact of Refactoring

LESSON: ADVANCED REFACTORING

- Design Patterns
- Code That Feels Wrong
- Refactoring to Design Patterns
- Abstract Factory Design Patterns
- Sample Refactorings
- Adapter Design Patterns
- Sample Refactorings
- Strategy Design Patterns
- Sample Refactorings

SESSION ADVANCED TOPICS

LESSON: ADVANCED TDD TOPICS

- Mock Objects and EasyMock
- Decoupling with Mock Objects
- Mock object frameworks
- EasyMock and JUnit
- Dependency Injection, Spring and Testing
- Dependency Injection and IoC
- The Spring Framework
- Mock Objects and Spring
- State-based vs. Interaction-based Testing
- State-based testing
- Interaction-based testing

LESSON: CONTINUOUS INTEGRATION

- Overview of Continuous Integration
- Typical Continuous Integration Process
- Local Development Environment
- CI Server
- Potential Benefits of Continuous Integration
- Continuous Integration Best Practices Overview
- Automate Source Code Management
- Automate Build Process
- Goal: Consistent Build Process
- Why Use Build Management in an IDE?
- Decoupling the IDE from Code
- Automate Testing
- Automate Deployment
- Commit Code Early and Often
- Manage the Build Process
- Separate Integration Environment
- Mimic Production Environment
- Increase Visibility

LESSON: AGILE TESTING BEST PRACTICES

- Coding Practices
- State- vs. Interaction-based Testing
- Source Control
- Pair Programming and Code Reviews
- Continuous Integration
- Legacy Code