Training:

Automated Testing for Agile developers (TDD, BDD, Spec. by Example, patterns, tool)

General information

Name: Automated Testing for Agile developers (TDD, BDD, Spec. by Example, patterns, tool)
Code: Test-Auto
Category: Testing
Target audience: Developers, designers
Duration: 3 days
Form: 50% lectures / 50% workshop

During the training, participants will learn techniques for programming and testing to facilitate the daily work with the code.

During the workshop, participants will gain practical ability to write and maintain tests, and work using Test-Driven Development technique.

The training program integrates popular tools (JBehave, Selenium, Spring remoting, JUnit). The training program integrates popular tools (JBehave, Selenium, Spring remoting, JUnit) to support the process of creating the executable specification.

By executable specifications we understand literally the text of Acceptance Scenario (part of the User Story), that can be run to make sure if it is currently fulfilled.

Running a scenario can occur at any level of the system architecture: through the GUI or API layer.

Advantages:

» Automation tools
» Application architecture to support testable code
» Best patterns and practices
» Aspects of Behavior Driven Development and Domain Driven Design
1. Fundamentals of testing

1.1. Methods for testing the system

1.2. Types of tests and examples of their use

1.2.1. The scope of testing

1.2.1.1. Unit tests

1.2.1.2. Integration tests

1.2.1.3. End 2 End

1.2.2. The role of testing

1.2.2.1. Acceptance tests

1.2.2.2. Regression tests

1.2.3. The target of testing

1.2.3.1. Functional tests

1.2.3.2. Security tests

1.2.3.3. Performance tests

1.3. Automating the testing process

1.4. Selection of testing strategies in the project

1.5. The strategy of building a pyramid of tests

2. Designing the test cases

2.1. Approach to documenting test

2.1.1. User Story and Acceptance Scenario

2.1.2. Executable specifications - Behavior Driven Development techniques

2.2. Testing borderline cases

2.2.1. The focus on unit tests to achieve high test coverage
### 2.2.2. Architecture that supports high test coverage

### 2.2.3. Dividing the logic into application logic and domain logic

### 2.2.4. Modeling the login using Building Blocks within Domain Driven Design

### 2.3. Repeatability of the tests, eliminating the randomness of the tests

### 2.4. Best practices for creating test cases

#### 3. Application architecture that is open for testing

##### 3.1. Dividing the logic into application logic and domain logic

##### 3.2. Pyramid of test - how to interpret in the context of layers

- **3.2.1. Application logic - End 2 End tests**
- **3.2.2. Domain logic - unit test**

##### 3.3. When is it worth to use Mock and when is it redundant

### 4. Best testing techniques - designing tests that can be maintained in the future

#### 4.1. Introduction

- **4.1.1. What not to test?**
- **4.1.2. The structure of the test cases**

#### 4.2. Organizing the test code

- **4.2.1. Test class per production class**
- **4.2.2. Test class per functionality**
- **4.2.3. Test class per setup**
- **4.2.4. Parameterized tests**

#### 4.3. Test fixture setup

- **4.3.1. Tests that use data sources (data-driven testing)**
- **4.3.2. Using Assembler pattern (variation of Builder Design Pattern)**
- **4.3.3. Patterns and templates**

#### 4.4. Verification
4.4.1. Value Object, verification by equals

4.4.2. Custom assertions

4.4.3. Verification by specification (Matcher object)

4.4.4. Simplification of assertion by using Assert Object

4.4.5. The correct verification of negative cases

4.4.6. Patterns and templates

4.5. Cleaning after the test (fixture teardown)

4.5.1. When to use

4.5.2. Manual

4.5.3. Automatic

4.5.4. Patterns and templates

4.6. Anti-patterns in testing (over 20 typical mistakes and traps)

4.7. Detection of the smells in the wrong test code

4.7.1. Fragile tests

4.7.2. Unreadable tests

4.7.3. Slow tests

4.7.4. Nondeterministic tests

5. Unit tests

5.1. Templates in xUnit

5.2. Creating custom assertions

5.3. Techniques: Mock, Stub, Fake

5.3.1. The choice of techniques to meet the needs - what is the direction

5.3.2. Samples of implementation in Mockito

5.4. Mocking

5.4.1. Advantages of testing in isolation
5.4.2. Recording the behavior

5.4.3. Verification of requests

5.4.4. Anti-patterns of tests using mocking

5.5. What makes the time to write the test returns a the benefit

5.6. Testability - susceptibility of the code for test

5.6.1. How to write code that can be tested

5.6.2. Best practices: SOLID, GRASP

5.6.3. The selected design patterns that enhance testability: Factory, Strategy, Value Object

5.6.4. Traps and common mistakes

5.6.5. Code smell - "smells" non-testable code

6. Test Driven Development

6.1. Cycle red-green-refactoring

6.2. Advantages and disadvantages of TDD

6.3. When it is worth, for whom it is TDD

6.4. Evolutionary development of a code

6.5. Basic techniques of refactoring

7. Integration testing

7.1. Testing event-driven-architecture

8. Acceptance testing

8.1. User Story technique

8.2. Creating acceptance tests based on User Story

8.3. Benefits and costs of different techniques acceptance testing

8.4. Effective tools to support:

8.4.1. Testing through the GUI layer
### 8.4.2. Testing by a layer of services

### 8.4.3. Preparation of the initial state

### 8.5. Acceptance testing of GUI layer (end-to-end)

### 8.6. Approaches

#### 8.6.1. Testing through GUI

##### 8.6.1.1. Selenium

#### 8.6.2. Testing through the service layer (system API)

##### 8.6.2.1. Object remote call (Spring Remoting/Remote EJB/Seam - your choice)

#### 8.6.3. Unification of approaches through BDD Agents

### 9. The nature and problems specific to each kind of test

#### 9.1. Functional tests

#### 9.2. Performance tests

#### 9.3. Security tests

#### 9.4. Integration tests

### 10. Behavior Driven Development

#### 10.1. The advantages of close cooperation with the customer

##### 10.1.1. Vendor role, client role in acceptance tests

#### 10.2. BDD approach to create applications

#### 10.3. 2-layered approach

##### 10.3.1. Flow layer - User Story

##### 10.3.2. Layer for automation with the system

#### 10.4. Tools and patterns

##### 10.4.1. JBehave - best practices

##### 10.4.1.1. Integration with Selenium and Spring/EJB

##### 10.4.2. Page Object - modeling the user
### 11. Specification by Example

11.1. Patterns and techniques for creating executable specifications

11.2. 3-layered approach

   11.2.1. Specification Layer - business goals

   11.2.2. Flow Layer - User Story

   11.2.3. Layer for automation the interaction with the system

11.3. Automation tools

### 12. Techniques for test code organization - standards, patterns and best practices

12.1. Support for refactoring with IDE

12.2. Techniques for refactoring the testing code

12.3. Effective ways to maintain a large number of tests

### 13. Code review

13.1. Collecting and interpreting metrics

13.2. Detection of critical points of the system

13.3. Identifying code smells

13.4. Introduction to pair programming

   13.4.1. The Psychological Bases

   13.4.2. Model Dreyfus

   13.4.3. Using the potential of the two hemispheres of the brain

### 14. Techniques specific for frameworks / architectures (your choice)

14.1. Java

   14.1.1. Spring framework

   14.1.2. JBoss Seam

   14.1.3. Hibernate
14.2. .NET

14.2.1. Using ServiceLocator and Dependency Injection

14.3. Architecture n-layered

15. Testing 'end 2 end' by the presentation layer

15.1. Using Selenium and JBehave

15.2. Integration with Spring framework (on request)

16. The comprehensive process

16.1. Strategic design - Domain Driven Design

16.2. Using the user story, BDD and TDD in every day work

16.3. Automation - using Continuous Integration server

16.4. Tools (your choice)

16.4.1. Java: JUnit/TestNG, JBehave (plugins Eclipse), Selenium

16.4.2. .NET: nUnit, NDbUnit, nBehave, nSubstitute, nCover, Selenium

16.5. Using best practices, introducing typical mistakes and traps

16.6. Sample project