TDD and JUnit

Authored and Presented by : Vijeta Angeer

This presentation is the intellectual property of Cybage Software Pvt. Ltd. and is meant for the usage of the intended Cybage employee/s for training purpose only. This should not be used for any other purpose or reproduced in any other form without written permission and consent of the concerned authorities.

Copyright © 2013. Cybage Software Pvt. Ltd. All Rights Reserved. Cybage Confidential.

Agenda

- Overview on TDD
- JUnit 4
- JUnit Concept

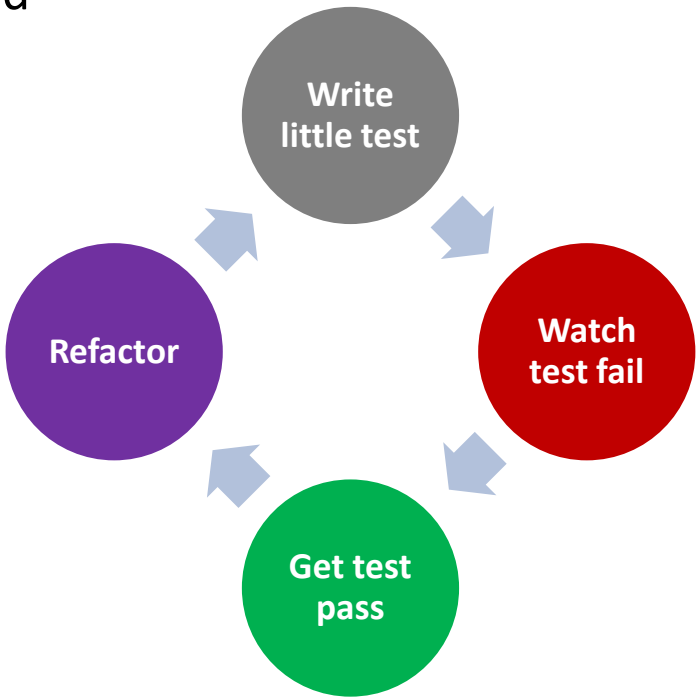
Copyright © 2013. Cybage Software Pvt. Ltd. All Rights Reserved. Cybage Confidential.

TDD is a Practice

- Write tests before coding
- Unit test should meet Customer requirements
- UT tells what a Class and its methods should do?
- Use Junit or TestNG frameworks
- UT is a test of a [Small functional piece of code]
- Use Mock objects for Dependencies
- Code Review of UT by Peer/TL
- Use Maven/Gradle – Build tools to run UTs
- Check your code coverage > 85%
- Use Code coverage tool – Emma/Clover

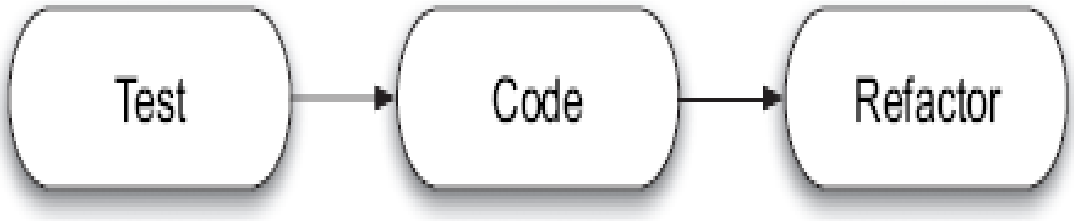
What is TDD ?

- **Test-driven development** (TDD) is a software development process
- That relies on the repetition of a very short development cycle.
- Approach for developing software by writing tests before writing the code being tested



Build it right: TDD

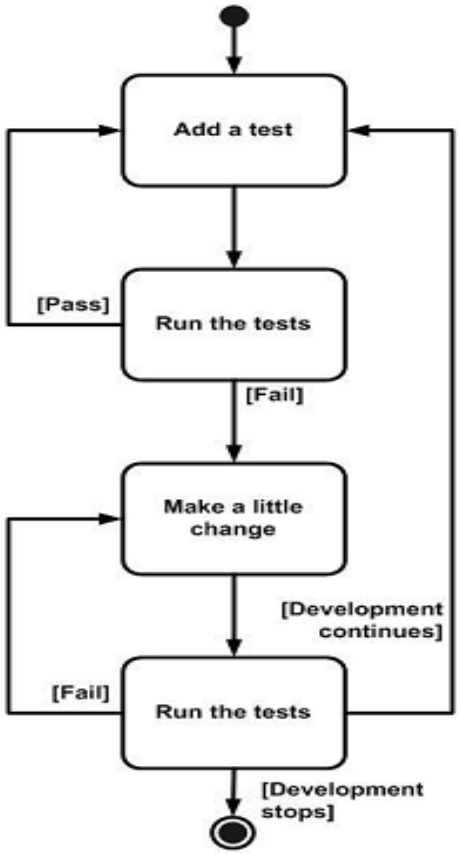
- **Test-Code-Refactor**



- The term refactoring is used to better communicate that the last step is about transforming the current design toward a better design.

Test-Driven Development Cycle

- Add a Test
- Run all tests and see if the new one fails
- Write some code
- Run tests
- Refactor code
- Repeat



What is Refactoring?

- Changing the structure of the code without changing its behavior

- Example refactoring's:
  - ✓ Rename
  - ✓ Extract method/extract interface
  - ✓ Inline
  - ✓ Pull up/Push down

- Some IDE’s (e.g. Eclipse) include automated refactoring's

TDD Misconceptions

- There are many misconceptions about TDD
  - They probably stem from the fact that the first word in TDD is “Test”
  - TDD is not about testing, TDD is about design
  - Automated tests are just a nice side effect
  - TDD does not mean “the TDD process”
  - TDD is a practice(like pair programming, code reviews, and stand-up meetings)
  - not a process(like waterfall, Scrum, XP, TSP)

JUnit 4

- JUnit is an open source Java testing framework used to write and run repeatable tests
- JUnit is integrated with several IDEs, including Eclipse
- Download from http://www.junit.org and place distribution jar in your classpath

JUnit Concepts

- **Test Case** – Java class containing test methods
- **Test Method** – a non-argument method of a TestCase class annotated with @Test

```
@Test
public void someTestMethod() { … }
```

- **Fixture** - the initial state of a Test Case
- Test method contains business logic and
  assertions – check if actual results equals expected results
- **Test Suite** – collection of several Test Cases

Fixture: setUp/tearDown

- Fixtures is a fixed state of a set of objects used as a baseline for running tests.
- The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.
- It includes
  - setUp() method which runs before every test invocation.
  - tearDown() method which runs after every test method.

Fixture: setUp/tearDown

- Before/After runs for each Test method
- Annotate a method with @Before to initialize the variables in that method
- Annotate a method with @After to release any permanent resources you allocated during set up

```
@Before public void setUp() {...}

@After public void tearDown() {...}
```

Test Lifecycle

- **@Before public void setUp() {...}**

  gets called once before each test method is executed
- **@After public void tearDown() {...}**

  gets called once after each test method has been executed – regardless of whether the test passed or failed (or threw some other exception)
- JUnit 4 introduces **suite-wide initialization** - class-scoped setUp() and tearDown() methods
- any method annotated **@BeforeClass** will run exactly once before the test methods in that class run
- any method annotated with **@AfterClass** will run exactly once after all the tests in the class have been run

Example

Calculation.java

```
package com.alm.tdddemo;

public class Calculation {

    public static int add(int a, int b) {
        return a + b;
    }

    public static int sub(int a, int b) {
        return a - b;
    }
}
```

CalculationTest.java

```
package com.alm.tdddemo;

import static org.junit.Assert.*;

public class CalculationTest {

    private int value1;
    private int value2;

    @Before
    public void setUp() throws Exception {
    value1 = 5;
    value2 = 3;
    }

    @After
    public void tearDown() throws Exception {
    value1 = 0;
    value2 = 0;
    }

    @Test
    public void testAdd() {
    int total = 8;
    int sum = Calculation.add(value1, value2);
    assertEquals(sum, total);
    }

    @Test
    public void testSub() {
    int total = 2;
    int sub = Calculation.sub(value1, value2);
    assertEquals(sub, total);
    }

    @Test
    public void testFailedAdd() {
    int total = 9;
    int sum = Calculation.add(value1, value2);
    assertNotSame(sum, total);
    }
}
```

Assertions

- Assertions are used to check that actual test results are the same as expected results
- A set of assert methods is specified in org.junit.Assert (see JUnit JavaDocs)
- The most often used assertions –assertEquals(), assertNull(), assertSame(), assertTrue() and their opposites – are enough for most situations
- Assert.fail() is used, if control should not reach that line in the test - this makes the test method to fail

Example

```
import org.junit.Assert;
import org.junit.Test;

public class TestCase{

  @Test public void myTest()
  {
  List<Customer> customers =
        customerService.getAllCustomers();
  Assert.assertEquals(12, customers.size());

  Customer customer = customerService.getById("123");
  Assert.assertNotNull("Customer not found", customer);

  boolean isActive = customerService.isActive(customer);
  Assert.assertTrue("Customer is not active", isActive);
  }
}
```

Testing Expected Exceptions

- In JUnit 4, you can write the code that throws the exception and simply use an annotation to declare that the exception is expected:

```
@Test(expected=ArithmeticException.class)
  public void divideByZero() {
    int n = 2 / 0;
}
```

- Test fails if exception will not be thrown

Ignored Tests

- Sometimes it's useful to mark test to be ignored by test runner
  - test that takes an excessively long time to run
  - test that access remote network servers
  - test is failing for reasons beyond your control

- Such tests can be annotated as @Ignore

```
@Ignored
@Test public void myTest() {
}
```

Timed Tests

- In JUnit 4 tests can be annotated with a timeout parameter
- If the test takes longer than the specified number of milliseconds to run, the test fails

```
@Test(timeout=500)
public void retrieveAllElementsInDocument() {
    doc.query("//*");
}
```

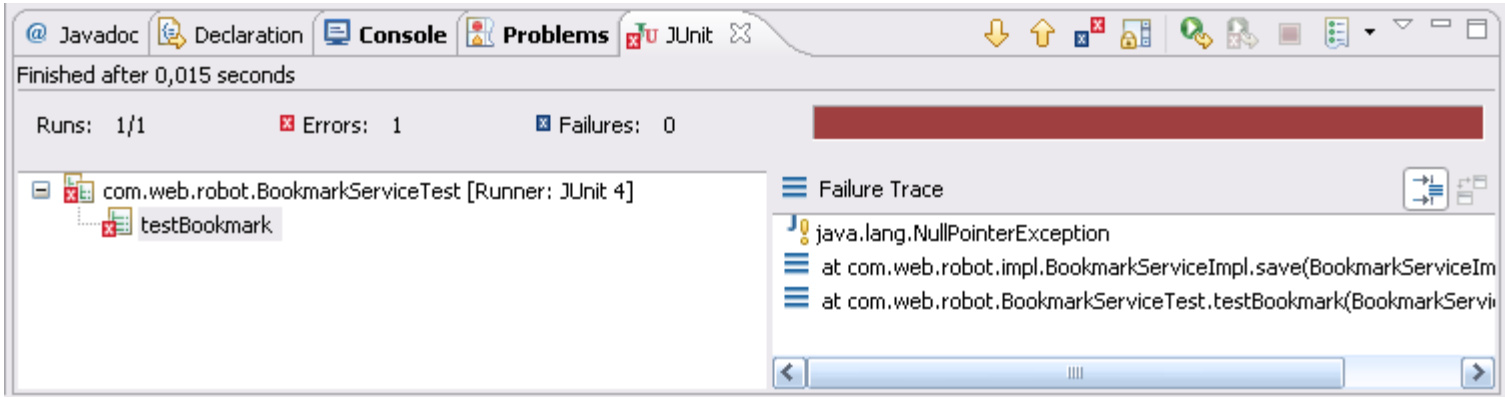
Test Suite

- Test Cases can be combined into suites

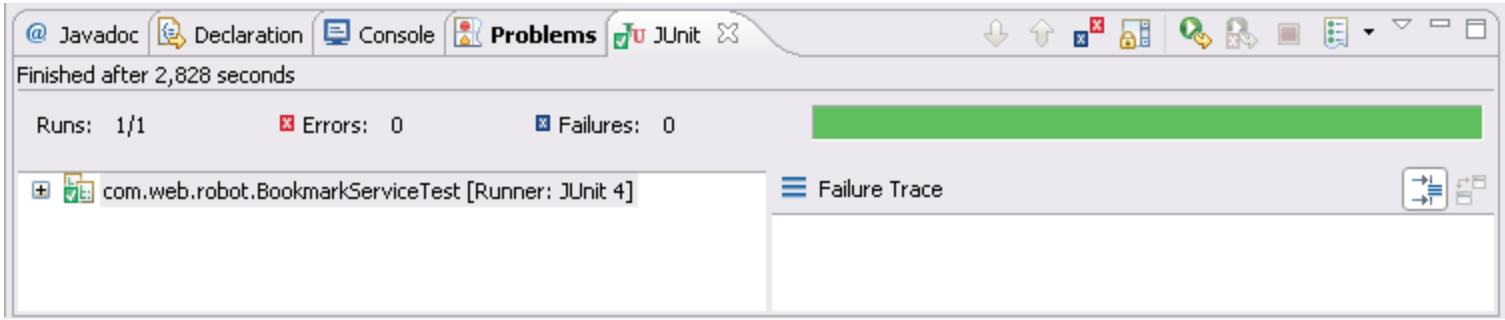
```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses(value =
    {CalculatorIntegerTest.class,
    CalculatorFloatingPointTest.class,
    CalculatorLogarithmsTest.class })
public class Calculator TestSuite {
   // Can leave empty
}
```

Running from Eclipse

- Right click test class → “Run As” → “JUnit Test”
- Failed run:



- Successful run:



Use Maven – > run Junit tests & Code coverage Tool

> mvn test

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building web-robot
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------------
. . .
[INFO] [surefire:test]
[INFO] Surefire report directory:
C:\tools\eclipse\workspace\java-eim-lab01-robot\target\surefire-reports
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.web.robot.BookmarkServiceTest
. . .
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 9 seconds
[INFO] Finished at: Thu Sep 20 09:55:04 EEST 2007
[INFO] Final Memory: 4M/8M
[INFO] ------------------------------------------------------------------------
```

JUnit Categories

- **Define the Marker Interface**

- The first step in grouping a test using categories is to create a marker interface.

- This interface will be used to mark all of the tests that you want to be run as integration tests.

```
public interface IntegrationTest {}
public interface CriticalTest {}
public interface SanityTest {}
```

JUnit Categories

- **Mark your test classes**

➢ Next we add the category annotation to the top of your test class. It takes the name of your new interface as a parameter.

```
import org.junit.Test;
import org.junit.experimental.categories.Category;
public class MyFeatureTest {
@Test @Category(FastTests.class)
public void testFast() {
System.out.println("fast");
}
@Test @Category(SlowTests.class)
public void testSlow() {
System.out.println("slow");
}
@Test @Category({SanityTests.class, SlowTests.class})
public void testSanity() {
System.out.println("sanity");
}
}
```

JUnit Categories

- We can run all the test under a category using the -Dgroups maven attribute. For example run all the fast and slow tests:

```
mvn test -Dgroups="com.test.groups.FastTests, com.test.groups.SlowTests“
```

- -Dtest to run only a specific test in a test class

```
mvn test -Dtest=AppTest.java  -Dgroups="com.test.groups.SanityTests"
```

Any Questions?

Thank You!

Copyright © 2014. Cybage Software Pvt. Ltd. All Rights Reserved. Cybage Confidential.