Test Driven Development with JUnit

Mikhail Andrenkov

Department of Computing and Software
McMaster University

Week 2: Sept 19 - 23

Outline

TDD Introduction

- **Test Driven Development:** A software development technique involving short development cycles
  1. Requirements are converted into a set of tests
  2. Software is only improved to pass these tests

- **Unit Testing:** Testing small "units" of code independently
  1. Facilitates debugging (finding bugs and catching unintended side effects)
  2. Promotes modular coding

TDD Process

1. Write a *test case* based on the requirements
2. Run all the test cases to check if the new one fails
3. If it fails, modify the code until the new test passes
4. Improve the code until *all* the tests are passing
5. Repeat

JUnit Introduction

- **JUnit:** A Java unit testing framework
  1. Excecutes a *flow* that independently runs all the test cases
  2. Displays the result of each test case (pass or fail)

- The Eclipse IDE has a convenient JUnit plugin

JUnit Download

Windows/OS X/Linux

1. Download and install JUnit from: `https://github.com/junit-team/junit4/wiki/Download-and-Install`
   - Download the latest `junit.jar` and `hamcrest-core.jar`
   - The newest versions are `junit-4.12.jar` and `hamcrest-core-1.3.jar`

Setting up JUnit Tests

- Assume the following setup:
  1. The class to be tested is `Design.java`
  2. The class containing the JUnit tests is `Tester.java`
  3. The path to `junit-4.13.jar` is `<junit_path>`
  4. The path to `hamcrest-core-1.3.jar` is `<hamcrest_path>`
     - Make sure these paths are absolute paths! For example, the `javac` and `java` command-line tools will not expand shell shortcuts like `*` or `~`

Running Unit Tests

- To run the tests:
  1. Compile `Design.java`: `javac Design.java`
  2. Compile `Tester.java`:
     - **Windows:** `javac -cp .;<junit_path> Tester.java`
     - **OS X/Linux:** Same as Windows, except replace ";" with ":"
  3. Run the tests:
     - **Windows:** `java -cp .;<junit_path>;<hamcrest_path> org.junit.runner.JUnitCore Tester`
     - **OS X/Linux:** Same as Windows, except replace ";" with ":"

JUnit Skeleton

```
import static org.junit.Assert.*;
import org.junit.BeforeClass;
import org.junit.Test;

public class JUnitSkeleton {
  @BeforeClass  // Runs once before all the test cases
  public static void setUp() {
    System.out.println("First");
  }

  @Test  // A test case
  public void testCase1() {
    assertEquals(val1, val2); // Assert that val1 = val2
  }

  @Test  // Another test case
  public void testCase2() {
    fail("Not yet implemented"); // Automatic failure
  }
}
```

Example: Initial Method Setup

- **Requirement:** The method must return the value of $f(x)$, where f is a line with the following parameters:
  1. The slope of the line is 2
  2. The y-intercept of the line is 3

- First attempt:

```
public class Line {

  public static int calculateLine(int x) {
    return 0;
  }
}
```

Example: JUnit Skeleton

- Begin with a skeleton of the JUnit testing code:

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

public class LineTest {

  @Test
  public void test() {
    fail("Not yet implemented");
  }
}
```

Example: Initial Test Case

- Implement a basic test:

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

public class LineTest {

  @Test
  public void testCalculateLine() {
    assertEquals(Line.calculateLine(4), 11);
  }
}
```

- Since `Line.calculateLine(int x)` always returns 0, this test should fail

Example: Improve the Implementation

- Improve the method implementation to satisfy the test case:

```
public class Line {

  public static int calculateLine(int x) {
    // Slope of line
    int m = 2;
    // Y-intercept of line
    int b = 3;

    // Compute f(x) = m*x + b
    return m*x + b;
  }
}
```

- Check if the new implementation passes the test case

Example: Improve the Test

- Improve the coverage of the JUnit test by adding additional checks:

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

public class LineTest {

  @Test
  public void testCalculateLine() {
    assertEquals(Line.calculateLine(-8), -13);
    assertEquals(Line.calculateLine(0), 3);
    assertEquals(Line.calculateLine(4), 11);
  }
}
```

- Are these checks sufficient to conclude method correctness?

Conclusion

- Test Driven Development (TDD) helps produce code that is:
  1. Meaningful
  2. Modularized and maintainable
  3. More *likely* to be correct
- JUnit is a Java framework that facilitates unit testing
  - Allows for the easy creation and execution of a test suite

Further Reading

- JUnit Annotions: `https://www.mkyong.com/unittest/junit-4-tutorial-1-basic-usage/`
- TDD Best Practices: `https://technologyconversations.com/2013/12/24/test-driven-development-tdd-best-practices-using-java-examples-2/`