Flexible, Reliable Software – Anno 2020

Introduction

The book *Flexible, Reliable Software* celebrates its tenth anniversary in 2020. Looking back, I am happy to say that all core contents of the book is still valid: the principles, the techniques, the patterns – they are all just as sound and useful today as they were in 2010.

However, the technological platforms on which we develop software is in constant flux, and over the years I have each year updated the provided codebase, scripts, and advice, to keep the core exercises, projects, and example code more in line with state-of-the-art development environments, and advances in the Java language and its libraries.

Ideally the book should have followed along and been constantly updated. However, a printed book (for all its merits) is not ideal for that purpose.

Therefore, I have decided to write this document that provides a 'delta' to figures and references in the book. It is a bit tedious, I agree, as you have to have this document open while reading the book. But – read the introduction of each 'delta chapter' in this document before reading the book's chapter, and then cross reference once you stumble into 'old stuff' in the book.

Summer 2020 – Henrik Bærbak Christensen

Chapter 2 – Reliability and Testing

This chapter is about terminology, and nothing has changed there. However, JUnit and the way we use it, has changed since 2010.

Automated Testing tools have grown in numbers and features since 2010. However, I will stick to JUnit, but use it in version 5, and utilize the 'hamcrest' matcher library, instead of the 'assertEqual()' method that is used throughout the FRS book.

Hamcrest matchers provide two benefits

- It uses a 'fluent API' way of expressing test cases that makes them *almost* readable directly.
- The output generated in the IDE in case a test case fails is more specific about what went wrong.

Changing to JUnit 5 and Hamcrest means new libraries to include in the Java classpath as well as new imports in your Java code. Please refer to Chapter 5 below for these details.

2.4 JUnit: An Automated Test Tool

The modern version of the TestDayOfWeek test case, using modern Java date classes and the Hamcrest matchers would look like:

```
@Test
public void shouldGiveSaturdayFor25Dec2010() {
  LocalDate date = LocalDate.of( year: 2010,  month: 12,  dayOfMonth: 25);
  assertThat(date.getDayOfWeek(), is(DayOfWeek.SATURDAY));
}
```

The main thing to highlight is the *assertThat* which takes two parameters: The first is the *computed value*, and the next is a *matcher expression* that is designed to be just about readable English: "Assert that date.getDayOfWeek() is Saturday."

Sidebar 2.2's long list of Junit 4.4 assert methods can then be rephrased using the Hamcrest matchers like this

```
@Test
public void shouldDemoSidebar2_2() {
  assertThat( actual: true, is( value: true));
  assertThat( actual: null, is(nullValue()));
  assertThat( actual: "fish", is(not(nullValue())));
  assertThat( actual: "fish", is( value: "fish"));
  assertThat( actual: 7.123, closeTo( operand: 7.124,  error: 0.005));

  assertThat( actual: "This is a fish", containsString( substring: "fish"));

  List<String> l = Arrays.asList("Bimse", "Bumse");
  assertThat(l, hasItem("Bimse"));
  assertThat(l, not(hasItem("Fish")));
  assertThat(l, hasItems("Bumse", "Bimse"));
}
```

The lower half of the code above shows some handy additional matchers provided, making it easy to test substrings ('containsString()') and if items are in an array ('hasItem').

Sidebar 2.3 explains using JUnit in the raw java compiler. Nowadays we always run code using some build management system, like Gradle, or an integrated development environment, like IntelliJ.

Chapter 5 – TDD

The TDD principles and processes ('the rhythm') is the same, the changes are in the JUnit tooling, so below each iteration of the book is updated with figures from a modern JUnit 5 and Hamcrest tool stack.

5.3 – Iteration 1: Inserting Five Cents

Using JUnit 5 and the Hamcrest matchers, the first iteration's test code will look like

```java
package paystation.domain;

import org.junit.jupiter.api.*;

import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.MatcherAssert.assertThat;

import java.util.*;

/** Testcases for the Pay Station system. */
public class TestPayStation {

  @Test
  public void shouldDisplay2MinFor5Cents()
      throws IllegalCoinException {
    PayStation ps = new PayStationImpl();
    ps.addPayment(5);
    assertThat(ps.readDisplay(), is(2));
  }
}
```

Sidebar 5.1. The updated pay station code will use *Gradle* as build management system. Gradle will download the proper libraries (JUnit and Hamcrest) as well as compile and execute. To run the above test case, you would issue 'gradle test':

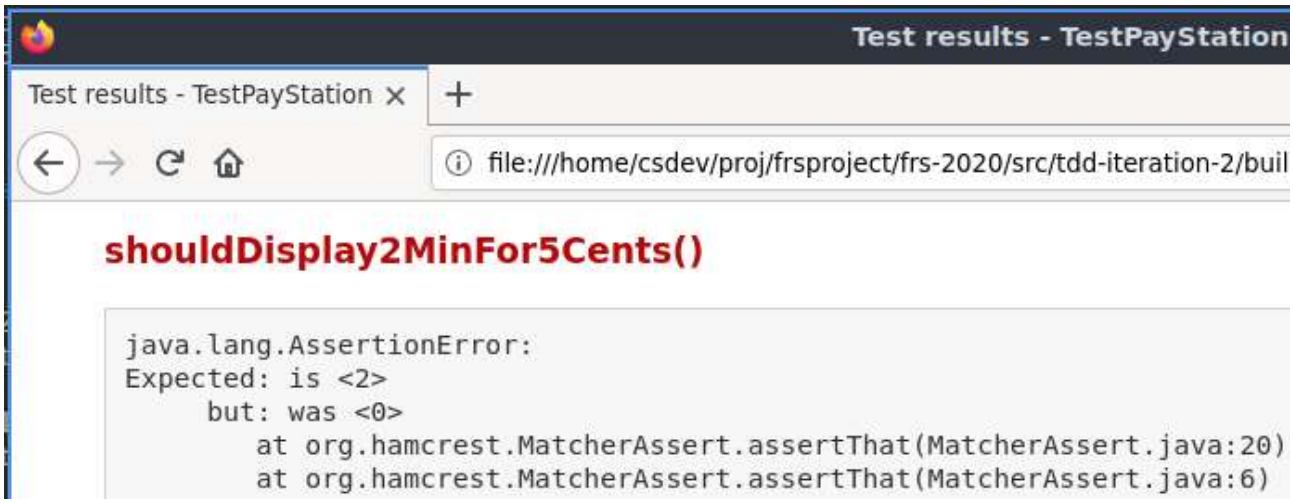
```
csdev@m33:~/proj/frsproject/frs-2020/src/tdd-iteration-1$ gradle test

> Task :test FAILED

TestPayStation > shouldDisplay2MinFor5Cents() FAILED
    java.lang.AssertionError at TestPayStation.java:18

1 test completed, 1 failed
```

Gradle does not provide any detailed information about *why* a test case failed, but instead generates a HTML report that can be browsed.



Normally, you would rather develop in e.g. IntelliJ, and run the test cases within that environment.

You will find the project on www.baerbak.com, on the dedicated link for the 2020 version of the FRS book.

5.4 - Iteration 2: Rate Calculation

The new test case:

```
@Test
public void shouldDisplay10MinFor25Cents()
    throws IllegalCoinException {
  PayStation ps = new PayStationImpl();
  ps.addPayment(25);
  assertThat(ps.readDisplay(), is(10));
}
```

The **refactoring step** is actually different, because JUnit 5 has decided to rename the @Before annotation to @BeforeEach. While I find that quite annoying after having used @Before for more than ten years, it is actually a better name as it clearly states that you run the @BeforeEach method 'before each test method'.

```
/** Testcases for the Pay Station system. */
public class TestPayStation {
  private PayStation ps;

  @BeforeEach
  public void setUp() {
    ps = new PayStationImpl();
  }

  @Test
  public void shouldDisplay2MinFor5Cents()
      throws IllegalCoinException {
    ps.addPayment(5);
    assertThat(ps.readDisplay(), is(2));
  }

  @Test
  public void shouldDisplay10MinFor25Cents()
      throws IllegalCoinException {
    ps.addPayment(25);
    assertThat(ps.readDisplay(), is(10));
  }
}
```

There are also a @BeforeAll method which is run *once* before calling any methods in the test file.

5.5 – Iteration 3: Illegal Coins

JUnit 5 discontinues the way to express catching exceptions known from JUnit 4, and replace it with a *assertThrows* static method. It takes two parameters: the exception to expect, and a lambda function.

As the method returns the thrown exception, it also allows to verify the contents of the exception. So, our test case becomes:

```
@Test
public void shouldRejectIllegal17CentCoin() {
  IllegalCoinException theException =
    assertThrows(IllegalCoinException.class,
                 () -> ps.addPayment(17));
    assertThat(theException.getMessage(),
                 containsString("Invalid coin: 17"));
}
```

5.6 – 5.11 – The following iterations

The following iterations does not introduce any aspect, that is not already covered by the techniques above – it is just standard test-driven development.

Chapter 6 – Build Management

Build management systems today generally follows the *by convention* paradigm, whereas Ant described in the FRS book follows the *by configuration* paradigm. The latter paradigm is similar to a programming language, in which you define *what to do*. Like making a *build-src* target that defines what to do: call the javac compiler.

In later years, I have switched to *Gradle* which instead relies on conventions on how your code is organized, and then by itself provides all the common household tasks associated with software development: compiling code, generate JavaDoc, and running test cases.

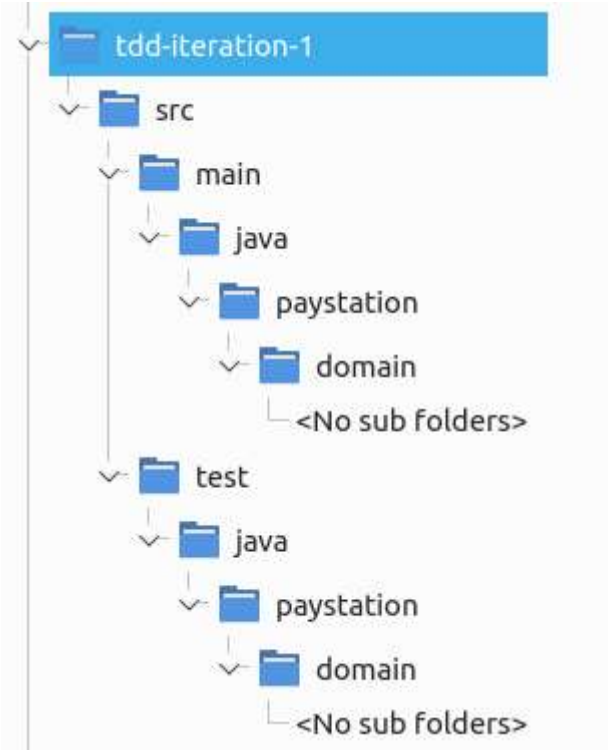
The benefit of Gradle compared to Ant is that your build description is much smaller, as you normally do not define targets and procedures. The liability is that you need to understand the conventions used, otherwise really nothing works, and you need to known the built-in targets.

Another big advantage of Gradle is built-in *dependency management*, that is, Gradle can be told which Java libraries to include in the classpath when compiling and running – these are then automatically downloaded from maven repository (https://mvnrepository.com/).

Thus, it makes little sense to TDD a build description for Gradle, as I do in the FRS book, as everything is declarative in the file.

Source code folder layout

Gradle assumes a standard layout of your folders containing production- and test code, similar to this:



The root folder (here tdd-iteration-1) must contain the build description in a file named *build.gradle*.

If you compare it to §6.3.8 and figure 6.1 in FRS, it is not that different, except gradle support multiple languages, so any Java production source code *must reside* in *src/main/java/(packagename)* while test code in *src/test/java/(packagename).*

Build.gradle

A build.gradle file for the PayStation (which works with Gradle 6.5) is shown below

```
apply plugin: 'java'

repositories {
    jcenter()
}

dependencies {
    // Depend on JUnit 5. Require both API and Engine
    testImplementation group: 'org.junit.jupiter',
       name: 'junit-jupiter-api', version: '5.6.2'
    testRuntimeOnly group: 'org.junit.jupiter',
       name: 'junit-jupiter-engine', version: '5.6.2'

    // Use the Hamcrest matcher library
    testCompile group: 'org.hamcrest',
        name: 'hamcrest-library', version: '2.2'
}
```

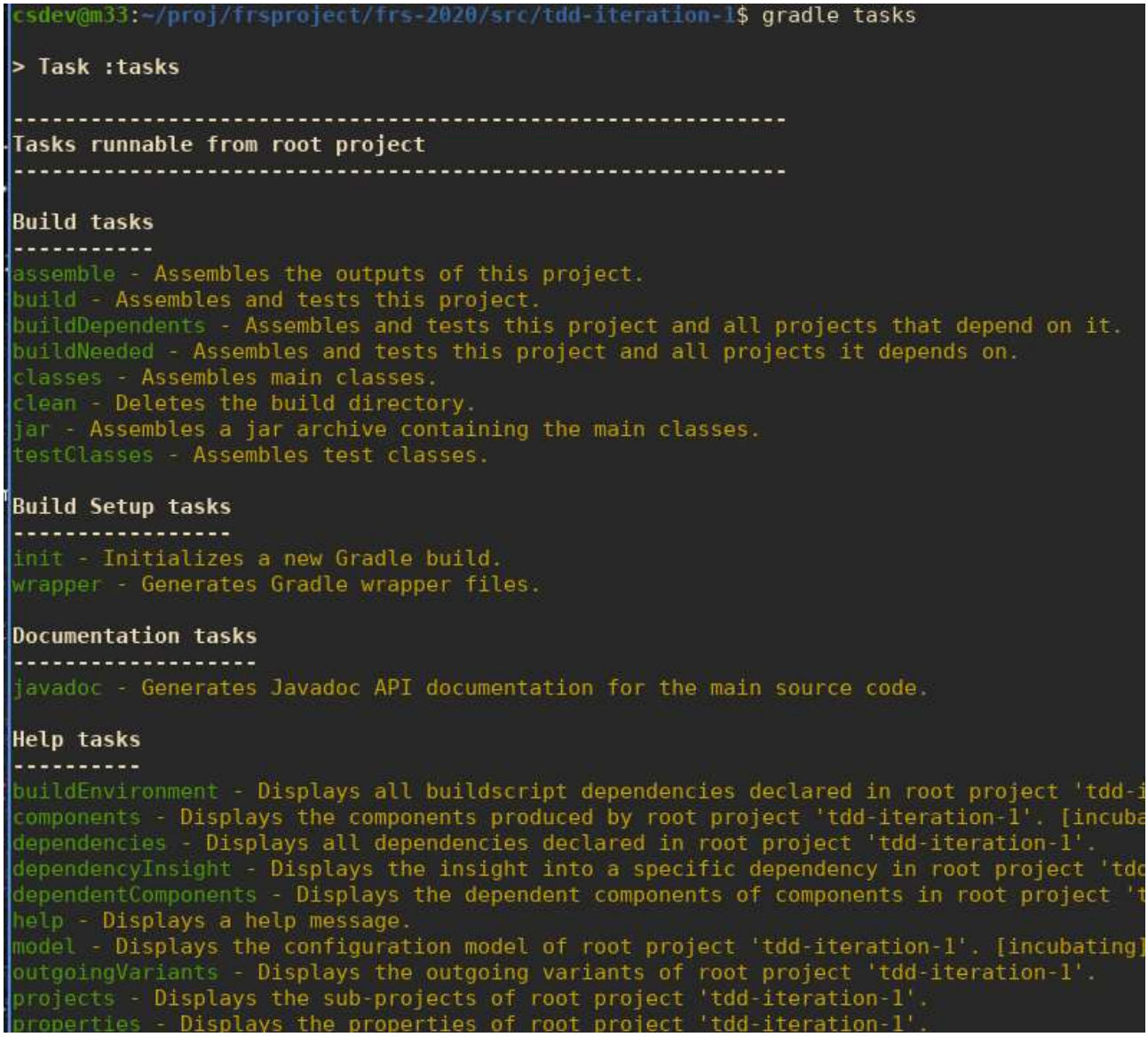
Basically, it just states declaratively that this build description is for Java, that it shall pull libraries from the JCenter repository, and that the source code depends upon three libraries, two for JUnit 5, and one for Hamcrest. You find the exact text to paste into the 'dependencies' section of the build.gradle by searching for libraries at mvnrepository.com. For instance, if you want to include the Unirest Java library, which allows HTTP calls to webservers, I would search for it:

I then choose which version of the library I want, which provides the details needed:

Gradle targets

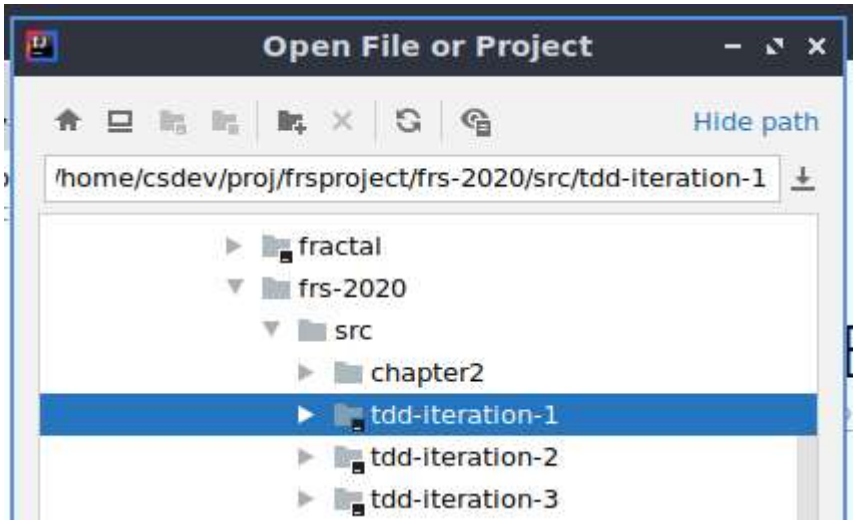
Gradle knows how to compile and test Java programs using standard targets. In daily development I almost exclusively do test-driven development, so the command issued again and again is: **gradle test.**

However, you can get the full (and very long) list of targets by invoking **gradle tasks**.

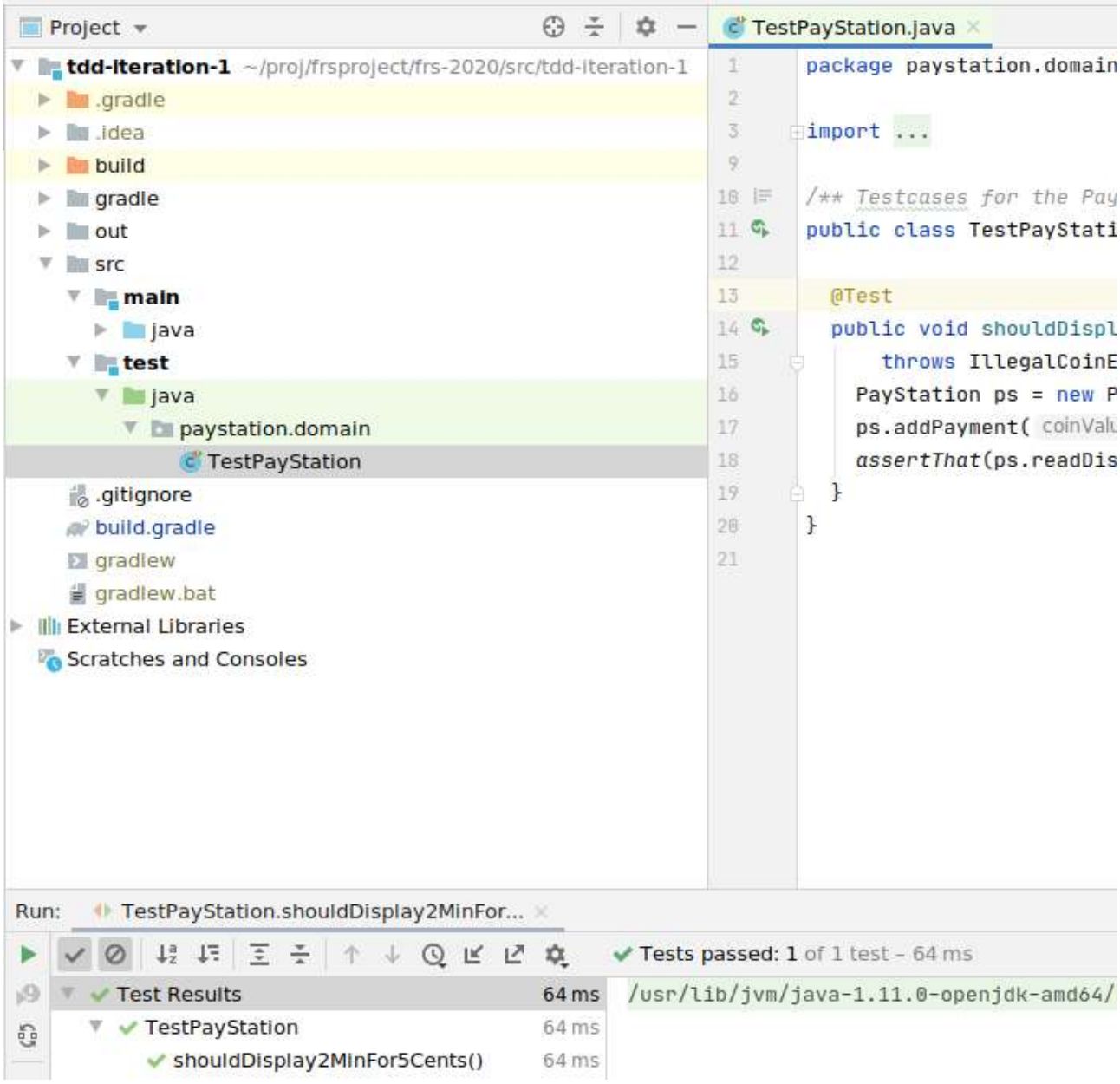


Integration into IntelliJ

IntelliJ is a powerful integrated development environment, which understands Gradle out of the box. To open your project in IntelliJ, just ask it to open the root folder of your gradle project.



Next, IntelliJ will spend some time importing your project. After than you can run the test cases from within IntelliJ.


Project
tdd-iteration-1 ~/proj/frsproject/frs-2020/src/tdd-iteration-1
.gradle
.idea
build
gradle
out
src
main
java
test
java
paystation.domain
TestPayStation
.gitignore
build.gradle
gradlew
gradlew.bat
External Libraries
Scratches and Consoles
TestPayStation.java
1 package paystation.domain
2
3 import ...
9
10 /** Testcases for the Pay
11 public class TestPayStati
12
13 @Test
14 public void shouldDispl
15 throws IllegalCoinE
16 PayStation ps = new P
17 ps.addPayment(coinValu
18 assertThat(ps.readDis
19 }
20 }
21
Run: TestPayStation.shouldDisplay2MinFor...
Tests passed: 1 of 1 test – 64 ms
Test Results 64 ms
TestPayStation 64 ms
shouldDisplay2MinFor5Cents() 64 ms
/usr/lib/jvm/java-1.11.0-openjdk-amd64/