Workshop: Agile Practices for Testing

JOOSE-2

January 14, 2016

1 Introduction

This exercise has three purposes:

- to introduce two *agile* software engineering practices to improve the quality of your source code called *test first programming* and *pair programming*.
- to increase your familiarisation with `Scanner`s
- to increase your experience of Computer Aided Software Engineering (CASE) tools, specifically JUnit.

1.1 Agile Methods

Agile methods are a collection of *software development methods* that prescribe processes and practices for the professional development of software. You can find out more about agile methods from the web, or from this book:

> Kent Beck and Cynthia Andres. Extreme Programming Explained. XP Series. Addison Wesley/Pearson Education, second edition, February 2005.

For this exercise, you are going to try two different software practices.

1.1.1 Test First Programming

The purpose of testing is to identify incorrect behaviour in a software program. Testing involves the design and implementation of *test cases*, each of which describes a possible input for a program and the expected output. Test case suites (collections of test cases) are designed to cover the inputs to a program as comprehensively as possible. In principle, if the test suite is complete and correct, then any code which passes all the tests should also be correct. So, if we write the tests first, we shift the problem of writing correct code to writing correct test cases. The test cases also make the expected behaviour of the program under test explicit. For this exercise, you will use the JUnit package to develop a small suite of test cases, as practice for the assessed exercises. You can find out more about JUnit from the project website.[1]

[1] `http://junit.org/`

1.1.2 Pair Programming

Source code review is a technique for identifying and correcting errors in software. The idea is that 'many eyeballs' scrutinising a source code listing are more likely to identify and correct mistakes, all without actually running the program. Pair programming combines source code reviews with the implementation itself. Pairs of programmers sit at a single terminal and work on one problem. One programmer (the pilot) controls the mouse and keyboard. The other programmer (the navigator) observes the work, identifies mistakes and suggests corrections.

1.2 Scanners

A `Scanner` instance reads a stream of characters, breaking them up into tokens. By default, a `Scanner` uses white space (blanks, tabs, end-of-line characters) to separate tokens; you may specify that different characters (or patterns of varying complexity) be used to separate tokens in the stream.

Of particular interest is the ability to create a Scanner from a `String` (a stream of characters from the character at index 0 to the character at index length-1).

1.3 Procedure

Your tutors will organise you into programming pairs. One of the pair should be the programmer and the other the navigator. The pilot will do the work of controlling the mouse and keyboard, while the navigator watches and checks for mistakes. Note, this is *not* an excuse for the navigator to take time out! For the first hour, you should spend the time working on the test cases. Swap round the roles for the second part of the exercise, when you do the implementation. [2]

1.4 The Problem

The United Kingdom went through a process of *decimalisation* of its currency in the 1970s, meaning that a decimal relationship between denominations of currency (pounds and pennies) was established, replacing the variable relationships that had been used until then. As a consequence it is often hard to work out the value of items priced in the old (sometimes called Imperial) currency. The programming task is to implement a utility for parsing and converting Imperial Pound Sterling currency amounts into the modern decimal Pounds Sterling amounts.

The relationships between the currencies is as follows:

- an Imperial penny (denoted 1d.) is the smallest type of imperial currency;

[2] In each lab session, there is a 50% probability of a triple. In this case, there will be two navigators watching one pilot and you should swap round after 40 minutes.

- an Imperial shilling (denoted 1s.) is worth 12 Imperial pennies;
- an Imperial pound (denoted £ 1) is worth 20 shillings (and thus 240d.);
- a decimal penny (denoted 1p) is the smallest type of decimal currency;
- a decimal pound (also denoted £1) is worth 100 decimal pennies;
- a decimal pound is worth the same as an Imperial pound; and
- as a consequence of the above, an Imperial penny is worth 100/240 (= 5/12) decimal pennies.

An amount of Imperial money is written like this:

```
£ 2.1s.4d.
```

Which means two pounds, one shilling and four pence (notice the dots after each amount).

The task is split into two parts:

- parsing old currency values. This will be achieved by writing an implementation of the `CurrencyParser` interface. This interface has a single method:

```java
package uk.ac.glasgow.senotes.currency;

import java.util.List;

public interface CurrencyParser {

  /**
   * Reads a String describing a currency specific
   * format amount of money.
   * @param currency
   */
  public List<Currency> parseCurrency(String currency);
}
```

- converting the old currency into decimal values. This will be achieved by writing an implementation of the `CurrencyConverter` interface.

```java
package uk.ac.glasgow.senotes.currency;

import java.util.List;


public interface CurrencyConverter {

  /**
```

```
     * Adds an amount of currency to the converter.
     */
    public void addCurrency(Currency d);

    /**
     * Returns a list of currency denominations
     * converted from the
     * input currency to the new currency.
     */
    public List<Currency> dispense ();
}
```

Both of these two interfaces operate on the `Currency` abstract class which has a single abstract method:

```
package uk.ac.glasgow.senotes.currency;


public abstract class Currency {

  /**
   * The amount of currency represented by this object.
   */
  public int amount;

  public Currency(Integer amount){
    this.amount = amount;
  }

  /**
   * @return the value of this currency relative
   * to it's smallest denomination (penny).
   */
  public abstract Integer getPennyValue();
}
```

The `Currency` class has a number of sub-classes, each of which implement the `getPennyValue()` method:

- `Currency`
- `ImperialPenny`
- `ImperialShilling`
- `ImperialPound`
- `DecimalPound`
- `DecimalPenny`

The architecture and interface specification for the program has already been specified and stub classes for the implementation have also been generated. All that is required is to prepare test cases and implement the functionality.

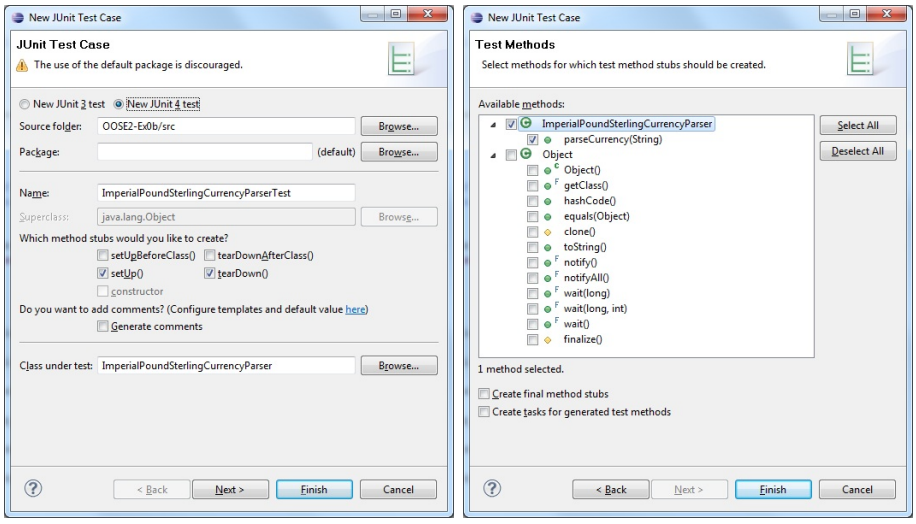
In Eclipse, create a new project named "currency"; then copy all the source package provided into the new project, *so that the package layout matches*. The code should now compile, but not actually do anything useful.

1.4.1 Test Cases

After you have got the code to compile, it is time to leave the implementation for the time being and start on your test cases. We will use the JUnit package (integrated into Eclipse) to do this. To get started create a new JUnit test case. Right click on the: `ImperialPoundSterlingCurrencyParser` class in the Package Explorer pane and select:

New → JUnit Test Case

from the context menu. This brings up a dialog similar to Figure 1a.



(a) (b)

Figure 1: Creating a JUnit test case.

Now:

1. Make sure that the New JUnit 4 test case radio button is selected.
2. Make sure the Name field contains "ImperialPoundSterlingCurrencyParserTest".
3. Make sure the Class Under Test field contains "ImperialPoundSterlingCurrencyParser".

4. Make sure the setUp and tearDown boxes are checked.
5. Click 'Next'. The dialogue should now look like Figure 1b.
6. On the Test Methods pages check the boxes for the `parseCurrency()` method.
7. Make sure that the other methods are unchecked.
8. Click Finish
9. When asked to add JUnit to the build path, click 'Yes'.

This will generate a test case class called `ImperialPoundSterlingCurrencyParserTest`, in the same package as the implementation class, with 3 methods:

```
package uk.ac.glasgow.senotes.currency;

import static org.junit.Assert.*;

import java.util.List;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;


public class ImperialPoundSterlingCurrencyParserTest {

  @Before
  public void setUp() throws Exception {

  }

  @After
  public void tearDown() throws Exception {
  }

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

}
```

Notice that each of the methods have an *annotation*, each of which provides information to JUnit about how the method should be used:

- `@Test` denotes a test method to be invoked each time the test case is run;

- `@Before` denotes a method to be run before each test method in a test case. The `setUp` is useful for creating a consistent *fixture* to be tested by each of the different test methods; and
- `@After` denotes a method to be run after each test method in a test case. The `tearDown` method is useful for cleaning up any program state that is left behind by the fixture after a test method is invoked.

the `testParseCurrency()` method also contain a single line:

```
fail("Not yet implemented");
```

This is the default *assertion* for a test case - when no other information is available assume that the test method will fail.

We can try running the tests. Select the `ImperialPoundSterlingCurrencyParserTest` class in the Package Explorer pane and then select:

Run → Run As → JUnit Test

from the menu bar. This will open the JUnit view at the bottom of the Eclipse IDE window, which will look something like Figure **??**.

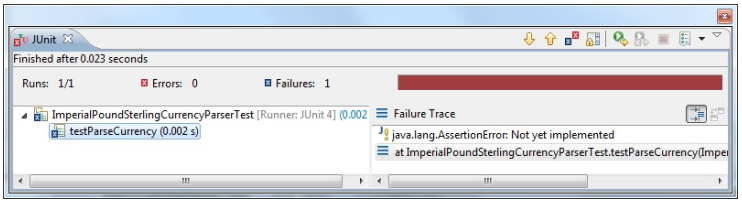


Figure 2: running JUnit in Eclipse running JUnit in Eclipse on the `ImperialPoundSterlingCurrencyParserTest` test case

Notice that the report lists the number of test methods run (1) and the number of fails (1). The red bar on the upper right hand side will turn green once our implementation passes all the tests we set.

We now need to start populating the test case methods with assertions. These describe the expected input and output combinations from the target methods. The only method to test in the CurrencyParser interface is the `parseCurrency()` method. This should take a String formatted amount of currency and turn it into a list of Currency objects. First, we need to set up the test environment.

Create a new field in the test case class `ImperialPoundSterlingCurrencyParserTest`:

```
private CurrencyParser cp;
```

Notice that the variable's type is an interface, not the class of the assigned object. In the body of the `setUp()` method write:

```
cp = new ImperialPoundSterlingCurrencyParser();
```

This will be invoked before each test method is run. To invoke the target method, paste the following code into the body of the `testParseCurrency` method, and remove the `fail()` assertion:

```
List<Currency> currency = cp.parseCurrency("£2.1s.4d.");
```

Finally, we can check whether the data received back matches our expectation using assertions. Paste the following code into the `testParseCurrency()` method below the last line:

```
assertEquals(2, currency.get(0).amount);
assertEquals(1, currency.get(1).amount);
assertEquals(4, currency.get(2).amount);
```

Assertions are static methods of the `org.junit.Assert` class. If an assertion is failed, this information is reported to the JUnit engine. You can find many other assert methods in the JavaDoc documentation for the class at.[3]

Once you have implemented your first test, you can try running JUnit again. Notice that the test still fails, but for different reasons. Looking at the JUnit panel shows exactly which assertions failed in the tests. The test cases will pass once we have successfully implemented the stub methods.

Repeat the general steps above for the `ImperialToDecimalCurrencyConverter` class. Start by generating some test data by calling the target methods, and then checking the results for correctness. You will need to use both methods together to generate test output data for checking. Use the `addCurrency()` method to change the state of the converter and the `dispense()` method to get the result. Comment each test method with a rational (what are you testing for, and why).

Add some more test data to the test methods, including checks for invalid or unexpected currency formats. You might want to test for amounts with no pounds or pence amounts, for example. You may generate test cases that cause the `InputMismatchException` exception to be thrown, in which case you should wrap the offending test case code with a `try-catch` block.

1.4.2 Implementation

Remember to swap roles at this point so that the pilot and navigator roles are reversed.

You will now need to fill in the method bodies on the unimplemented classes. You now need to replace the commented sections within `ImperialPoundSterlingCurrencyParser` with appropriate code to parse the amounts. Test your code regularly against your test case, as this will show you how far your implementation has progressed.

You will likely need to use the following methods defined for the `Scanner` class:

[3] `http://junit.org/apidocs/org/junit/Assert.html`

`Scanner(String source)`

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

`useDelimiter(String pattern):Scanner`

Sets this scanner's delimiting pattern to a pattern constructed from the specified String.

`nextInt():int`

Scans the next token of the input as an `int`.

A simple pattern for matching the delimiters in the currency string is included in the stub code for `ImperialCurrencyParser`.

Once you have the `parseCurrency()` method working, move on to the `addCurrency()` and `dispense()` methods in `ImperialToDecimalCurrencyConverter`.

1.5 If you Have Time

1. Swap test cases with another programming pair. How many of their tests does your code pass?
2. Investigate how to extend the regular expression used to parse currency amounts to handle other ways of expressing Imperial currency.
3. Tests can be spread across several test methods for a single target method. Consider how you would do this for this application.