Testing Tutorial

Introduction

Software Testing is an important part of Software Engineering. A test ensures the right behavior of the code under test. When you have written you first program you most probably tested it by executing the program and see if it produced the right output. If we want to program professionally we want to automate tests so that they can be executed later and validate the behavior also after we change code. That is not the only advantage of automated testing. It is also a good documentation because it shows example usages of the code and it has an influence on the software design: if it is possible to test a code-unit, this unit is decoupled from other code and can easily be reused. The software development process „Test-driven development (TDD)“ suggests that tests should be written before a unit of application code is written so that people think clearly about the expected output before the unit it is written.

JUnit

When you look at the software design of the StockWatcher application you created in Assignment 2, you might already notice some design flaws. One flaw is that it mixes business logic (handling and creating prices) with view code (displaying the prices) in a single class. Your task is to fix this, i.e. you have to create a separate class for the business logic.

The procedure we are introducing in this tutorial is usually meant for larger and more complicated applications. So while some of the steps in this procedure might seem too small for being a step on their own, we illustrate them and the procedure in this tutorial since they are important to understand for working on larger applications.

When changing code, the first step is usually to make the code testable. This procedure is called breaking dependencies because it makes easily testable code independent from code which cannot be tested as a single unit. To break dependencies you often first put the code into a separate method which we can test. And in a second step (when we have the code tested), refactor the code into a separate class. In this tutorial we will follow the same procedure. First, we will take out the business logic code that is tangled with view code in the same method and put it into a new method. After that, we have to break other dependencies. At this point, the code is testable and we will create test cases to test the behavior of the code. After that, we will perform the actual refactoring by taking the business logic code and putting it into a separate class. During the refactoring process we make sure that the behavior stays the same by running the test cases again after each refactoring step.

Step 1: Initial Setup

Before we can start we will need to make a small change to the code. The Unit Test we want to run cannot execute JavaScript code – it would need the GWT compiler for that. But the **Random.nextDouble()** method (**StockWatcher.refreshWatchList()**) directly calls a native JavaScript function. We will get around this problem by replacing the GWT Random method by the Java API Random method (this method can also be compiled to JavaScript and thus won't break the GWT application). After the change your code should look the following:

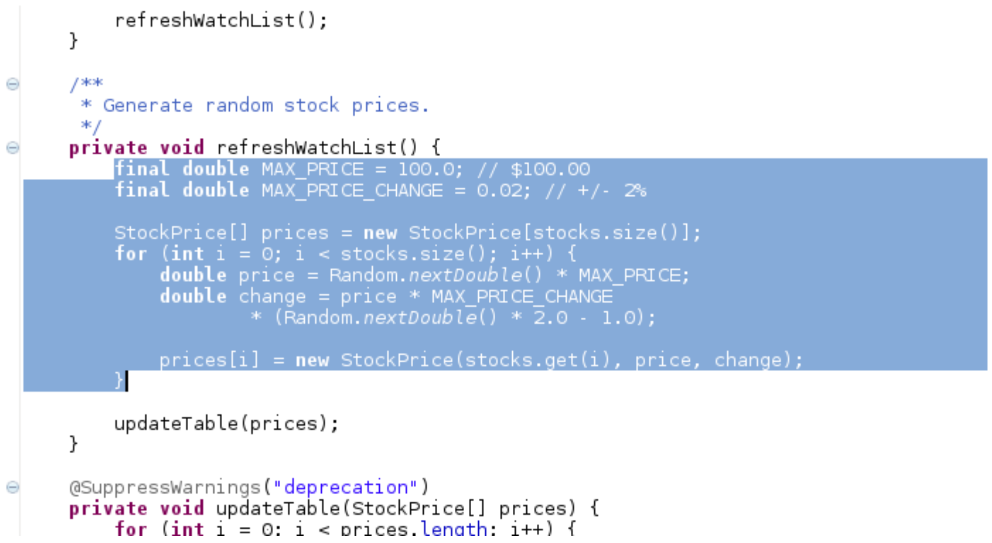
```java
final double MAX_PRICE = 100.0; // $100.00
final double MAX_PRICE_CHANGE = 0.02; // +/- 2%

StockPrice[] prices = new StockPrice[stocks.size()];
for (int i = 0; i < stocks.size(); i++) {

      double price = Math.random() * MAX_PRICE;
      double change = price * MAX_PRICE_CHANGE
                  * (Math.random() * 2.0 - 1.0);

      prices[i] = new StockPrice(stocks.get(i), price, change);
}
```

Step 2: Breaking Dependencies

Before we can write any reasonable test case, we need to make some changes to the code to make it testable. Since the method (**refreshWatchList**) we want to test is private, has no arguments and no return value, writing a test case that executes the method with certain values and then checks the outcome is not possible.

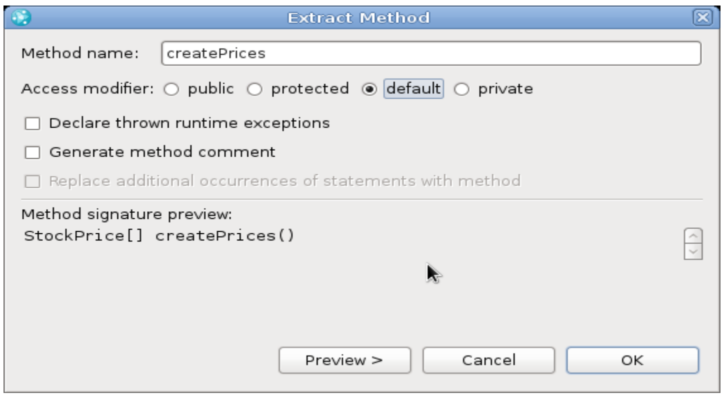
To make our code testable we need to break dependencies (as described above) and untangle business logic from view code within the same method by refactoring it into separate methods. If you examine the code you might already notice that the **stocks** variable as well as the **refreshWatchList** method have little in common with the rest of the class - they are in and do not really belong there (with respect to the functionality they realize). On further examination you might also realize that the check on a change being positive or negative should not be in the method **updateTable**. This tangling of the different functionalities represents a flaw in the design. Later on in this tutorial we will move this code to a new and better suited location, but for now, to make sure that our changes to the code do not alter the behavior of the application, we first want to make the code testable. Therefore, we will have to take the code which creates prices and put it into new method (**createPrices**).

To do so, select the code you want to extract:



```
        refreshWatchList();
    }

    /**
     * Generate random stock prices.
     */
    private void refreshWatchList() {
        final double MAX_PRICE = 100.0; // $100.00
        final double MAX_PRICE_CHANGE = 0.02; // +/- 2%

        StockPrice[] prices = new StockPrice[stocks.size()];
        for (int i = 0; i < stocks.size(); i++) {
            double price = Random.nextDouble() * MAX_PRICE;
            double change = price * MAX_PRICE_CHANGE
                    * (Random.nextDouble() * 2.0 - 1.0);

            prices[i] = new StockPrice(stocks.get(i), price, change);
        }

        updateTable(prices);
    }

    @SuppressWarnings("deprecation")
    private void updateTable(StockPrice[] prices) {
        for (int i = 0; i < prices.length; i++) {
```

Then open the “Refactor” menu (or right click on the selected code and click on “Refactor”) and select “Extract Method (Shift+Alt+M)”. In the dialog enter a name for the method (e.g. “**createPrices**”) and change the access modifier to “default” so that the testcase (which will be in the same package) can access it. Then click OK.



Method name: createPrices

Access modifier: ( ) public ( ) protected [✓ default] ( ) private

☐ Declare thrown runtime exceptions

☐ Generate method comment

☐ Replace additional occurrences of statements with method

Method signature preview:

```
StockPrice[] createPrices()
```

The new **refreshWatchList** method should look like the following:

```
private void refreshWatchList() {
       StockPrice[] prices = createPrices();
       updateTable(prices);
}
```

And the **createPrices** method should look like this:

```
StockPrice[] createPrices() {
      final double MAX_PRICE = 100.0; // $100.00
      final double MAX_PRICE_CHANGE = 0.02; // +/- 2%

      StockPrice[] prices = new StockPrice[stocks.size()];
      for (int i = 0; i < stocks.size(); i++) {
            double price = Random.nextDouble() * MAX_PRICE;
            double change = price * MAX_PRICE_CHANGE
                              * (Random.nextDouble() * 2.0 - 1.0);
            prices[i] = new StockPrice(stocks.get(i), price, change);
      }
      return prices;
}
```

Because the GWT GUI elements call native JavaScript code, we need the GWT compiler to link the sources. And therefore we cannot instantiate the **StockWatcher** class. After we have extracted the business logic methods to a new class, this will not be an issue anymore. But for now, to make the test case working, we temporarily make the method **createPrices** and the field **stocks** static:

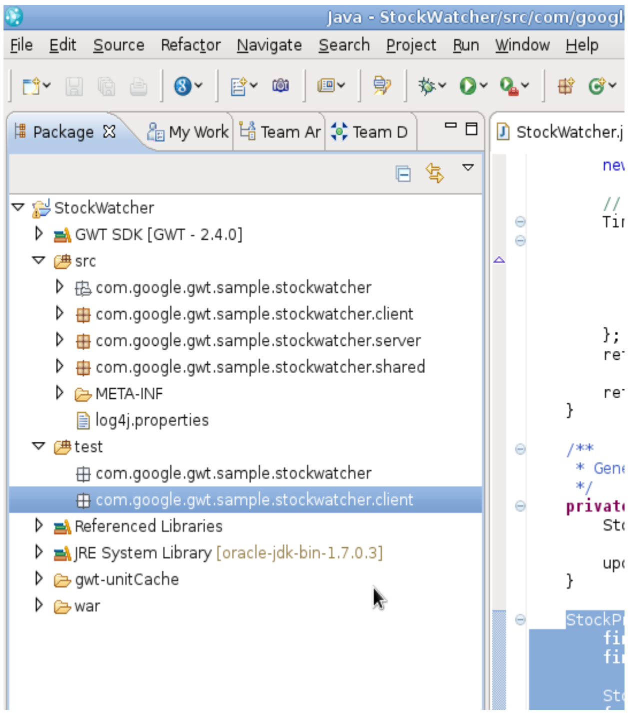
```
private static ArrayList<String> stocks = new ArrayList<String>();
//...
static StockPrice[] createPrices() {
//...
```

To be able to fill in test data, we also change the access modifier of the field **stocks** to package default (remove **private**):

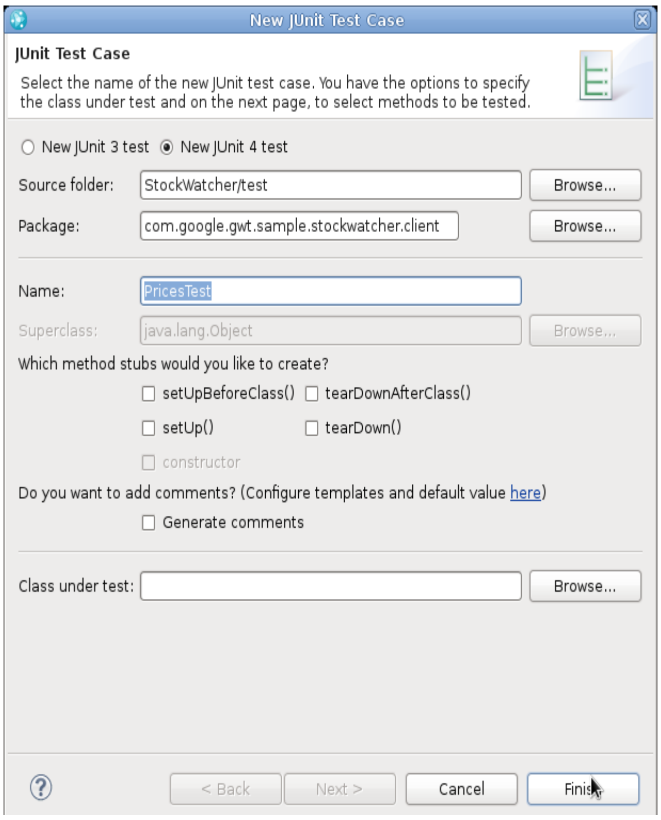
```
static ArrayList<String> stocks = new ArrayList<String>();
```

Step 3: Testing

We can access the method from within the same package. So we have to create the same package in the *test* source folder:
In the Package Explorer view, open the *test* source folder, right click on the existing package *com.google.gwt.sample.stockwatcher* and select *new → package* in the context menu.
As name enter: *com.google.gwt.sample.stockwatcher.client*
The package explorer view should now look like this:


Java - StockWatcher/src/com/googl
File Edit Source Refactor Navigate Search Project Run Window Help
Package
My Work
Team Ar
Team D
StockWatcher.j
StockWatcher
GWT SDK [GWT - 2.4.0]
src
com.google.gwt.sample.stockwatcher
com.google.gwt.sample.stockwatcher.client
com.google.gwt.sample.stockwatcher.server
com.google.gwt.sample.stockwatcher.shared
META-INF
log4j.properties
test
com.google.gwt.sample.stockwatcher
com.google.gwt.sample.stockwatcher.client
Referenced Libraries
JRE System Library [oracle-jdk-bin-1.7.0.3]
gwt-unitCache
war

Now right click on the new package and select *new → JUnit Test Case*.
Set the name to *PricesTest* and click *Finish*.



JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder: StockWatcher/test

Package: com.google.gwt.sample.stockwatcher.client

Name: PricesTest

Superclass: java.lang.Object

Which method stubs would you like to create?

- [ ] setUpBeforeClass()
- [ ] tearDownAfterClass()
- [ ] setUp()
- [ ] tearDown()
- [ ] constructor

Do you want to add comments? (Configure templates and default value here)

- [ ] Generate comments

Class under test:

If you're asked to add the JUnit 4 library to the build path, say OK.

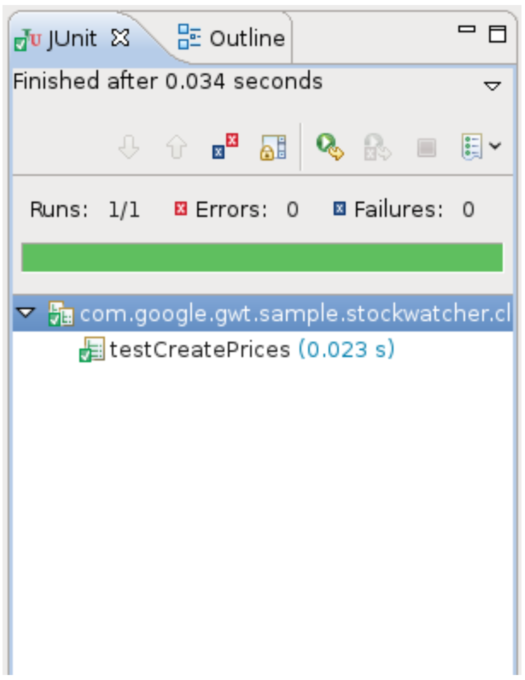
In the new class, add a test method:

```
package com.google.gwt.sample.stockwatcher.client;
import org.junit.Test;
public class PricesTest {
      @Test
      public void testCreatePrices() {

      }
}
```

Now fill in some test data into our static field:

```
      @Test
      public void testCreatePrices() {
            StockWatcher.stocks.add("AAPL");
            StockWatcher.stocks.add("GOOG");
            StockWatcher.stocks.add("AMZM");

            StockWatcher.createPrices();
      }
```

Run the test and see if it works so far by right clicking on the file → Run As → JUnit Test



```
Finished after 0.034 seconds
Runs: 1/1    Errors: 0    Failures: 0
com.google.gwt.sample.stockwatcher.cl
    testCreatePrices (0.023 s)
```

The test should succeed. Now call the method:

```
      @Test
      public void testCreatePrices() {
            StockWatcher.stocks.add("AAPL");
            StockWatcher.stocks.add("GOOG");
            StockWatcher.stocks.add("AMZM");

            StockWatcher.createPrices();
      }
```

And add the test assertions:

```
import static junit.framework.Assert.assertTrue;
import static org.junit.Assert.assertEquals;

// ... irelevant code ommitted

      @Test
            public void testCreatePrices() {
                  StockWatcher.stocks.add("AAPL");
                  StockWatcher.stocks.add("GOOG");
                  StockWatcher.stocks.add("AMZM");

                  StockPrice[] prices = StockWatcher.createPrices();

                  assertEquals(3, prices.length);
                  assertTrue(prices[0].getSymbol().equals("AAPL"));
      }
```

Let’s test what happens if the test fails:

Change the number 3 to 4 and execute the test.

This test only tests if three prices have been created and that the symbol of the first price is the right one (“AAPL”). It does not test if the created values (price and change) are in the right range. Write **two test methods** (in the same class), **one for the price and one for the change**.

Each test hast to be set up in a similar way (the stock has to be filled). You can create a method annotated with *@Before* to fill the stocks. Usually this method is called *setUp()*:

```
      @Before
      public void setUp() {

      }
```

Also **write a new test case** (new class) to test the *StockPrice* class. You only have to write a test that checks if the getChangePercent() method calculates the percentage the correctly.