CSIS0403/COMP3403
Implementation, Testing, and Maintenance of Software Systems
Department of Computer Science
The University of Hong Kong

Assignment 2: White Box Testing in Practice II

Description

Here we continue: development of mock objects and JUnit test cases, test execution, and collection of coverage metrics using Clover.

Details for Part 2

The objective of Assignment 2 is to unit test the *makeOrder()* method of the *Order* class and, indirectly, other methods of *Order* that are used by *makeOrder()*. There are several ways to do this. Whichever way you choose, you must test the *makeOrder()* method in isolation. That is, you must test without using any of the supplied code other than the *Order* class and the *Ingredient* interface. When unit testing, we don't assume anything about the implementation of other parts of the application and so cannot use such details in testing.

Also, remember that you are not testing the entire application even though it is simple. This is an exercise to test *makeOrder()* and your only concern is that *makeOrder()* does what it is supposed to do. The hints given below will touch on this further.

Tasks

I. Preparation

1) Install Clover as described in the tutorial.
2) Add JUnit to the build path of your project.

II. Implement Mock Objects

3) You need to create mock versions of the application classes used by *makeOrder()*. You need a *mockIngredient* class that implements the *Ingredient* interface, and a *mockIngredientFactory* to create instances of *mockIngredient* as required by *makeOrder()*.

   Hints on mocking:

   a) *mockIngredient*

   Mocks do no significant computation and, typically, their methods will only: (i) return predetermined values to the IUT, and (ii) collect information about how they were called by the IUT. The values returned by methods of the mock to the IUT are usually injected into the mock by the test case in order to drive the particular test execution. All you require from a mock, in general, is that it can provide a response to the IUT when called and also provide data to the test cases to help confirm the IUT called the mock correctly.

   Thus, when mocking ingredients to test *makeOrder()*, you do not care about the detailed behaviour of the different types of ingredient. For example, you don't care about the double dispatch used by ingredients to check compatibility with each other and you will not implement it. For the call to *isCompatibleWith()*, all a mock ingredient has to do is return a

value of *true* or *false* as preset by the test case; it need never call *isValidWith...()* on another ingredient.

Consequently, you will probably find you only need a single *mockIngredient* class. Instances of this class, configured with different data, can stand in place of any of the real ingredient objects.

b) *mockIngredientFactory*

To support testing, each creational method of the *mockIngredientFactory* class has two responsibilities. First it must record calls of creational methods received from the IUT so that test cases can determine the correct methods were called in response to user input. You may find it convenient to use a static array field to record counts of calls. The JUnit Assert library provides an *assertArrayEquals()* method, making it very straightforward to check if the calls are as expected by the test case.

The second responsibility is to return a *mockIngredient* object to the IUT. If you have a single *mockIngredient* class, then all creation methods will return an instance of this class. The objects can be instantiated and configured in the test case and injected into the *mockIngredientFactory* object to be returned in response to calls from the IUT. It is unlikely that any single test will require more than two *mockIngredient* instances.

III. Implement JUnit test cases

4) Implement JUnit test methods to execute each of the test cases in your table from Part 1 EXCEPT the case that executes `case 4` of the switch statement and for which `(choice != null)` evaluates *false*.

The `(choice != null)` condition illustrates a further problem with habitual null checking. Here you would need to be extremely creative to drive a *null* reference into the variable `choice`. You are not required to do so in this Assignment and you need not traverse this edge of the flow graph.

Hints on handling input and output:

a) *Input*

The constructor of *Order* takes a *Scanner* object as an argument. Since the number of user inputs in each test case is small, it is convenient to specify inputs as a string in each JUnit test method and pass the string as an argument when constructing the *Scanner*.

b) *Output*

The constructor of *Order* also takes a *PrintStream* object as an argument. In each test we want to check certain of the messages output by the IUT. A convenient way to do this is to instantiate a *ByteArrayOutputStream* object and pass it as an argument when constructing the *PrintStream*. After execution of the IUT, get the contents of the *ByteArrayOutputStream* as a string and then check that the string contains the expected messages.

You can use the *assertThat()* method with the Hamcrest matcher *containsString()* to perform the check. This will require the static import:

```
import static org.hamcrest.CoreMatchers.containsString;
```

IV. Run tests and generate report

5) Execute the JUnit tests and collect Clover coverage statistics.
6) Generate a Clover HTML coverage report in your workspace and check that it contains your test and coverage results.
7) Zip your Eclipse workspace for hand in.

Deliverables

1. The zipped Eclipse workspace including JUnit test class and methods, your mock objects, and the Clover HTML report in the *report* subdirectory.

Deadline

Tuesday, May 24: 23:55.