SOFT2201/COMP9201 Tutorial 7

Testing

Programmatic Code Testing

The level of testing we are interested in today takes place at about the lowest, simplest scale - testing a single 'unit' of code. This is usually done through frameworks that will take in some inputs, run the code it is testing, and then check what the code outputs to make sure it is correct.

In the general sense, every test of this nature (and most other tests) follow the same process:

- Set up the code to be tested (we call the result of this setup the 'test fixture')
- Give the code inputs (these inputs are known as 'test cases')
- Check the output the code gives for those inputs (these checks are called 'assertions')

We call this the 'given-when-then' test structure.

Question 1: Case Selection

The lecture gave you some strategies for selecting these test cases. As a class, apply these strategies to build a list for the following methods:

- A method that accepts an integer that must be >=0 (if it is negative it throws an exception). If it is given a number between 0 and 100 inclusive it returns the String "That could be a grade." Otherwise it returns "That is definitely not a grade."
- A method that accepts a String. If it is given a String that contains the word "test" it outputs "Yes, we are testing". Otherwise it returns "Why aren't you testing?"

Question 2: API reading

Programmatic tests are a double-edged sword. On the one hand, they provide a benefit because they can be run cheaply and easily and quickly identify potential bugs. On the other hand, it is far too easy to assume 'the tests pass, therefore the code is bug-free' - which is not correct. We need to handle both verification AND validation. A key tool for making sure that verification lines up with what validation has determined is an API reference.

As a class, review the Java 11 documentation for String, and in particular String.repeat (int count).

Assume you know nothing about repeating a String, or how you would expect this method to work. What does this method do, based on its Javadoc description? What do you know about the *preconditions* that govern the count parameter? Are there any preconditions aside from the parameter? What are the *post-conditions* of this method?

As a short aside, consider how useful this javadoc is. Consider how useful it would be to see similar javadoc in your own code.

Question 3: Design a Test Suite: Single Class

Without touching an IDE or writing a single line of code, you should now design a test suite for the Oven API. The class itself you can find as a file in this week's modules along with the API (the java file, not the jar). Make sure you can point at which parts of your test design are the 'given', 'when', and 'then' parts (subheading comments work well).

Compare your tests with the ones designed by other people in your tutorial - which test cases did you select? Did anyone think of an edge case others missed? Can you read and understand everyone's tests (remember tests are often used as documentation for functionality)

Question 4: Implement a Test Suite: Single Class

You should now implement and run your tests on the Oven class. You can check if your tests are working by introducing bugs into the Oven class and seeing what your test suite is able to pick up. (Because the Oven class refers to the Food class you will either need to skip to including the full JAR to run these, or just create the interface with a stubbed concrete class yourself). You should try running tests with both IntelliJ and gradle test to see the difference between these (your exercise and following assignment tests will be run with gradle test).

Question 5: Code Coverage

Your work requires you to ensure a particular level of code coverage. Code coverage is a metric for test completeness, and says 'when running the tests ##% of the lines of source code were executed'. Run your Oven test suite with coverage, and see how much of the code you have been testing.

To run coverage, you may either use IntelliJ's built in "run with coverage" option, or you may use the jacoco plugin for gradle (this plugin is how your exercise and following assignment will be marked). To leverage this plugin, in your build.gradle file add

```
id 'jacoco'
```

to the plugins block, then add

```
jacoco {
    toolVersion = "0.8.4"
    reportsDir = file("$buildDir/customJacocoReportDir")
}

jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination file("${buildDir}/jacocoHtml")
    }
}
```

outside any other blocks. You then run code coverage by running your tests (which will be run any time you call test or build) then running jacocoTestReport. So for example, `gradle build jacocoTestTreport`. Then look for the jacocoHtml/index.html file to view the results.

Where you have missed code, consider what bugs you might have let slip through. Improve your test suite until you have 100% coverage.

Question 6: Case Study

You will be testing three versions of the full Kitchen application, based on the Kitchen API (of which Oven is a part). The API can be found in module 7 on canvas. In this question, you will be given three JARs (which can be found in Exercise 7) that two of them have a bug that breaches the API requirements. Your task will be to indicate which JAR is the working version.

In order to do this you will need to be able to include a local JAR file into your gradle project (if you do things the way we have been this semester then JUnit is already included, though depending on how fancy you want your assertions to be you might want to pull in more of the Hamcrest library).

To include a JAR file, you should create a 'libs' folder parallel to (in the same as) your src folder and build.gradle file - place the JAR you want into that folder. Then add the following to your build.gradle file in the dependencies block:

```
compile files('libs/NameOfFile.jar')
```

You may then import classes from that JAR as if they were in your src directory (if you import your gradle changes in IntelliJ when asked it will even add them to the autocomplete suggestions).

You can also include multiple JAR files in the libs directory, eg named Kitchen1, Kitchen2, etc. - then change your build.gradle file to reference 1, then 2, and so on. You will need to attach your test suite gradle project in the weekly exercise submission.

A couple of tips:

- The bugs will directly breach the API as written - no assumptions of behaviour are required to detect them
- For example, one bug might be (might not be too) that Oven happily accepts a negative maxTemperature or that Potato burns at 50 degrees.
- The test case selection strategies you have been given in the lecture and practiced during tutorials will be all that are needed to find the bugs (that is, there is nothing cruel like "if this int is exactly 10,724 then throw an exception")
- Anything that is not mentioned in the API can be assumed as indeterminate behaviour and you do not need to test for it (this is much, much friendlier than real world testing) - for example, none of the API comments mention maximum or minimum integers or what to do if they overflow, therefore you don't need to worry about testing these.

Additional Note: Unit Tests?

While the answer to this question is out of scope for this unit, it is important that you do not leave here with misconceptions: a lot of the tests you will be writing, particularly for your following assignment, are not unit tests - they are *integrated* tests. These are tests where you allow other classes and objects to interact with your unit under test and potentially influence the outcomes of those tests. The issue here is that a test that fails on one class might be reacting to a bug in another class - reducing the ability of your tests to identify the location of a bug. Again, we will not be solving this issue this semester, but you should look forward to being able to separate a single unit in the future, and remember that until you do so you are not writing true 'unit' tests.