CSSE4630 Assignment 2: Dynamic Analysis: Code Coverage and Test Generation

Mark Utting, ITEE, UQ

Version 1.0

1 Introduction

This assignment you will learn about:

- using JUnit 5 for unit testing your Java projects (revision?);
- measuring several kinds of test coverage, including statement coverage and mutation score;
- using two black-box test generation techniques to make sure you are systematically testing a good range of the functional behaviour of your classes (and we will measure the resulting code coverage to see if it is higher). The two black-box techniques are:
  1. simple model-based testing (MBT) based on finite-state machine models;
  2. property-based testing (PBT) to check particular properties of your implementation;
- using concolic testing to improve your white-box test coverage scores;

We will perform the MBT and concolic test generation manually, so that you fully understand how these test generation techniques work. (There are tools available, but our primary goal here is understanding the test generation approach, so it is better not to use tools at first.)

For the property-based testing, I recommend you use the **jqwik** library, which integrates well with JUnit 5. (It would be too much of a pain to use property-based testing without tool support, and jqwik is so much fun that we have to try it!)

Your submission for this assignment will include all of your code, plus a report that summarises your results for each section. This report should be called '**Report.pdf**' and should be placed inside the top-level folder of your submission. It should start with a title, your full name and student number, then should have one section for each of the following sections of this assignment. Each section should be as short as possible, ideally just presenting your results as a table and any relevant code fragments or output listings.

1.1 Setup

Create a new project in IntelliJ, choosing the **gradle** option.

Update the new 'build.gradle' file to support JUnit 5, jqwik, and PIT (pitest), by adding the following three sections:

```
dependencies {
    //testCompile group: 'junit', name: 'junit', version: '4.12'
    testImplementation(platform('org.junit:junit-bom:5.7.0'))
    testImplementation('org.junit.jupiter:junit-jupiter')
```

```
    // aggregate jqwik dependency
    testImplementation "net.jqwik:jqwik:1.3.6"
}

test {
    useJUnitPlatform {
        includeEngines("junit-jupiter", 'jqwik')
    }
    testLogging {
        events "passed", "skipped", "failed"
    }
}

pitest {
    //adds dependency to org.pitest:pitest-junit5-plugin
    // and sets "testPlugin" to "junit5"
    junit5PluginVersion = '0.12'

    targetClasses = ['string_sets.MySet0']
    targetTests = ['string_sets.MySet0Test']
    //targetTests = ['string_sets.JqwikTestSet0']
}
```


Make sure that your **group** entry is set to the package that you will be testing:

```
group 'string_sets'
version '1.0-SNAPSHOT'
```


Finally, up the top of **build.gradle** in the **plugins** section, add the following two lines to include the PIT (pitest) mutation testing tool and to tell IntelliJ to use gradle to set up your project dependencies.


```
    id 'idea'          // to generate IntelliJ IDEA project files
    id 'info.solidsoft.pitest' version '1.5.1'
```

Then close the project and re-open it in IntelliJ.

Note: Any time you change your **build.gradle** file, you should either restart IntelliJ, or use **View / Tool Windows / Gradle / refresh-button** to refresh IntelliJ with the new settings.

Note the gradle conventions that you should follow:

- your Java source code packages must go into the **src/main/java** folder. Download the **MySet0.java** file from Blackboard and put it into **src/main/java/string_sets/**.
- your JUnit test classes and packages must go into the **src/test/java** folder. Download the **MySet0Test.java** file from Blackboard and put it into **src/test/java/string_sets/**.

Finally, if you would like to get rid of the '*No Jqwik properties file*' warnings, you can add an empty file at:

```
test/java/resources/jqwik.properties
```

2 Measuring Code Coverage [20 marks]

Your Task: measure statement coverage and mutation coverage of the SUT (**MySet0.java**).

Use suitable tools to measure several metrics of the code coverage of the tests that you were given. You should measure at least:

- Statement coverage (you can use IntelliJ 'Run with coverage' to measure this, or it is included in the PIT report);
- Mutation score (using PIT mutation testing tool);

Start by measuring the coverage of the SUT using just the basic test that is supplied on Blackboard: **MySet0Test.java**. Note that PIT output reports are generated in date-time folders inside **build/reports/pitest**. You should get a PIT mutation score of around 46%, and see output similar to the following:

```
============================================================
- Statistics
============================================================
>> Generated 35 mutations Killed 16 (46%)
>> Ran 19 tests (0.54 tests per mutation)
7:03:27 pm PIT >> INFO : Completed in 7 seconds
```

Add your results in a table with at least the following seven columns (you can add extra columns if you desire):

- Test Suite: the name of the tests you executed (e.g. **MySet0Test**);
- Add: the total number of calls to the **add** method in these tests;
- Remove: the total number of calls to the **remove** method in these tests;
- Size: the total number of calls to the **size** method in these tests;
- Contains: the total number of calls to the **contains** method in these tests;
- Statement%: the line coverage as a percentage;
- Mutation%: the PIT mutation score as a percentage;

3 Model-Based Testing [20 marks]

Your Task: use model-based testing to test the string-set implementation.

Write a small finite-state machine (FSM) model of a **Set¡String¿** object, with at most TWO strings. Since MBT is a black-box techniques, do not look at the SUT code, just write your model for any **Set¡String¿** implementation. (In fact, you can run your tests on some of the standard Java implementations if you like, such as:

- `new HashSet<String>()`
- `new TreeSet<String>()`
- and then the SUT: `new MySet0()` (If you find errors in this implementation, but your tests pass when run on the previous Java implementations, please email me your tests.)

Manually generate tests from your model, using an all-transitions generation algorithm such as the Chinese postman algorithm. Write the resulting test sequences as JUnit 5 tests in **src/test/java/string_sets/MBTMySet0Test.java**, and measure the coverage that they

give. Hint: you can do this by changing the `targetTests` property in your `build.gradle` file, then opening the Gradle tool window (**View / Tool Windows / Gradle**) and refreshing your project then running the **pitest** task there (

Record your model in the report (a photo of the hand-written FSM is fine, or you can draw it using software if you prefer).

Add the resulting statement and mutation coverage into your report, using the same table format as in the previous section. Make sure you are measuring just your model-based tests, not the previous tests as well.

4 Property-Based Testing [20 marks]

Your Task: use property-based testing (and the jqwik tool) to test the string-set implementation.

Write 2-4 JQWIK properties in JUnit 5 tests in **src/test/java/string_sets/JqwikMySet0Test.java**, that each capture some of the interesting behaviour of a set of strings. Since MBT is a black-box techniques, do not look at the SUT code, just write your model for any `Set<String>` implementation.

Execute those properties using JQWIK to see if the SUT satisfies the properties. After you have got your properties working with no errors, measure the statement and mutation coverage of the SUT using all your JQWIK tests.

Record your JQWIK properties in your report. Add the resulting statement and mutation coverage into your report (since we don't know exactly how many times JQWIK is calling each method, you can just put 100 in each of the method count columns of your table). Make sure you are measuring just your property-based tests, not the previous tests as well.

5 Concolic Testing [20 marks]

Your Task: Use concolic testing (a white-box test technique) to generate tests for the SUT. Record your tests in **src/test/java/string_sets/ConcolicMySet0Test.java**.

In this case you must look at the SUT code as you generate your tests. You should focus on generating tests that focus on the **add** method, including its call to the **find** method, which has numerous branches. So each of your tests should be just a sequence of textbfadd calls. But we will use the **size** function as an oracle to check the state of the SUT, so you can also sprinkle assertions using **size** anywhere along your sequence of **add** calls. You can also create a new empty SUT object if your concolic testing analysis requires you to start from an empty set again.

You will start with the **new MySet0()** object (the empty set). Use `null` as your first input value for the **add** method. For each unexplored branch that you find, write in your report:

- the branch condition;
- the direction that you need to explore (i.e. true or false);
- the collected constraints that will enable that direction to be taken;
- some concrete input values that satisfy those constraints (this will typically be the string parameter for your next call to **add**, or a sequence of **add(...)** calls that get the SUT into the desired state to take that branch.

Also, each time you start a new test sequence (with a **new MySet0()** object), copy the full generated test sequence into your report, as well as making it into a new JUnit test inside your **ConcolicStringSetTest.java** class.

When you are satisfied that you have generated enough tests for the **add** method, measure the coverage metrics of the generated tests on the SUT, and record them in your report.

After that, briefly use concolic testing to test the **remove** and **contains** methods *without* looking inside **find**. This should require only 2-3 tests for each method.

Then measure the coverage metrics of ALL your concolic-generated tests on the SUT, and record this result as an additional entry in your report. So you are adding TWO rows to your table in this section: for example with test suite names:

- **ConcolicMySet0Test(add-only)**
- **ConcolicMySet0Test(ALL)**.

6 Conclusions and Reflections [20 marks]

Write a final section in your report entitled 'Conclusions and Reflections' (less than one page), that includes:

1. **[6 marks]** A table and/or graph summarising your coverage results from each section (Sections 2-5), plus the coverage results of running ALL the tests on the SUT;
2. **[8 marks]** Your **Conclusions** about the benefits and limitations of each test-generation approach and how they compare;
3. **[6 marks]** Your **Reflections** on this assignment and what you learned.

7 Submission

This assignment is due on Friday of Week 12 (30-Oct-20 6pm Queensland time).

Submit your whole repository on Blackboard, in a single *.zip file. Make sure you include:

- your Report.pdf file inside the top level folder of your submissionl;
- the 'src' folder, since this contains all your tests in `src/test/java/string_sets`.
- the 'build/reports' folder, since the 'pitest' subfolder contains the output reports from your **pitest** runs. (You can delete your older runs and just keep the final ones if you are worried about having too many);
- your `build.gradle` file so that we can build and run your code if necessary.