CST316 Software Enterprise: Construction and Transition Spring 2016
Lab: Unit Testing

Objectives:
In this lab you will use JUnit to perform unit testing on a simple Banking app.
1. Understand black-box, or specification-based unit testing
2. Practice best practices in black box unit testing in JUnit
3. Understand white-box testing
4. Understand the various forms of code coverage
5. Practice best practices in white box unit testing in JUnit and the EclEmma plugin

Setup:
You will need a few things for this lab:
- The EclEmma Eclipse plugin
- You will maintain a separate documentation file. We prefer Microsoft Word, but you may also use OpenOffice.
- Please make sure you are using JUnit4 in Eclipse, not the older JUnit3.x

Tasks for this lab:
Task 1: Black-box testing
1. Run the program and get a feel for what it does. You can run it directly by right-clicking on Main.java in Eclipse and selecting Run As” → Java Application. You will need to configure the execution to take my.properties as a command-line argument.
2. You will notice a Junit test file, AccountServerTest.java. Run it by right-clicking and doing “Run As…” and select Junit test. How many tests run? Do they all pass or do any fail? Now run the junit task from the build.xml. Did it catch any failures?
3. For any tests that failed in the previous step, go fix the proper code so it makes the test pass (NOTE: do **not** modify the AccountServerTest.java file, this step means to go into the proper regular source file and fix it!). Put a comment just above the code fix: “CST316: Task 1, Step 3”
4. Create a unit test class called CheckingTest.java (use the eclipse tools -> right-click on project, New JUnit Test Case…)
   a. Write a test method for the “deposit” method, and then write one for the “withdraw” method.
   b. In your Word document, provide an explanation of the bases, boundary cases, and equivalence partitions for each of the deposit and withdraw methods and explain how your unit tests provide a complete set of tests.
      i. NOTE: Consider the principle of writing focused tests – one per each “equivalence partition.” Was this done in the AccountServerTest provided to you?
      ii. HINT: Remember to write positive, negative, and exception tests
5. Create a TestSuite AccountServerTestSuite named that runs the AccountServerTest and your CheckingTest test classes together. This is really easy in Eclipse, and basically gives you a way to group common tests together. You will find it under New…Java…JUnit
6. Run JUnit from ant (the task is in your build.xml). Review the output of the report generated, and ensure this report is included in your Word document (cut and paste it in).

Task 2: White-box testing
1. Draw control flow graphs for the Savings.deposit and Savings.withdraw methods.
   a. Turn on line numbering in Eclipse (Preferences→General→Editors→Text Editors)
   b. You may hand-draw these, take a photo, and import into your Word document. USE THE LINE NUMBER as the node label for each statement in your flow graph.
   c. For each graph, list the set of test sequences to achieve i) statement coverage, ii) edge coverage, and iii) condition coverage.
2. Create a new JUnit Test for Savings.deposit in SavingsTest.java.
   a. Savings.deposit actually has a defect wherein it always returns false. Write a unit test method named “testDeposit” that exposes this defect. Then fix the defect and make the test green.
   b. Write a test method for EACH INDIVIDUAL TEST SEQUENCE you identified in 1.c. Yes, some test sequences may address both statement and edge, or edge and condition coverage and so only need to be written once. Please put a header comment clearly indicating what test sequences the test method addresses.
3. Code coverage
   a. Add the EclEmma plugin for Eclipse. You can add this to your Eclipse by choosing one of the options here: http://eclemma.org/installation.html . Personally I prefer the Update Site (option 2) but your mileage may vary.
   b. Once the plugin has been installed and you have restarted Eclipse, run your unit tests again. Then you can view your code coverage results. Do either Window → Show View → Java → Coverage, or right-click on your Eclipse project in the Package Explorer, choose “Coverage As…” and “JUnit Test”, which will re-run your unit tests and also give you a Coverage View in the bottom pane.
   c. In your Word document:

i. Cut and paste your code coverage pane at the bottom of your Eclipse
ii. Answer: What is the overall code coverage for the project? For the ServerSolution.java file?
iii. Answer: Does the EclEmma coverage tool perform node or edge or condition or path coverage? Explain
iv. Answer: What is the code coverage of Savings.java, specifically the deposit and withdraw methods? If these methods are not at 100%, update your unit tests to improve the coverage.
v. Answer: What is the code coverage of Checking.java, specifically the deposit and withdraw methods

d. ServerSolution.java has a pretty low code coverage, at least lower than we'd like as developers (see your 3.c.ii answer). Can you quickly improve it? Implement code to update the coverage percentage, and explain what you did and why in your Word document. In your code, put in a header comment of "CST316 Task 2 Step 3 Part d" where you add your test code.

Optional: In the project you will find an ant script title build_coverage.xml. If you inspect it you will see it adds to the junit ant task by including this strange thing called "jacoco". Jacoco is the latest incarnation of Emma, the coverage tool upon which the EclEmma plugin you are using depends. This ant script can run code coverage using the jacoco library from with Ant and generate some nice reports in HTML, XML, and CSV files. However, the problem is due to a defect in Java on Mac OSX, you cannot run it from Macs. But you should be able to run it just fine on Windows and see these reports. I will post examples on Blackboard. To run, either open in Eclipse (the Ant view) and right-click on the junit target, or run from the command-line (ant –f build_coverage.xml junit).

Going Forward

A reminder that from this point forward you are expected to identify areas of the code to unit test and implement unit tests. This goes for every individual on the project team!