**Project Two**

Scott Vanderwilt

Southern New Hampshire University

CS 320: Software Testing, Automation & Quality Assurance

Dr. Angel Cross

4/16/2022

Project Two

My approach for unit testing the three features was directly aligned with the software requirements. For example, in the Customer class one of the requirements was that the firstName field could not be null and it could not be longer than 10 characters. To test this requirement, I first created a valid Customer instance to ensure that the object was being created correctly. Then I attempted to create a Customer object with a null for the firstName argument and made sure that an invalid argument exception was thrown. I then attempted to create a Customer object with a firstName argument longer than 10 characters and ensured that an invalid argument exception was thrown for that as well. These tests showed that the object was created with valid arguments and an object was not created with proper response sent to the user for invalid arguments.

The quality of the JUnit tests I created is shown by both the passed tests and by running the Java coverage tool showing that I had over 80% coverage on the tests. The passed tests show that the expected results were returned from the test code I ran. The nearly 100% coverage shows that the tests that I ran for the classes I was testing went through every part of the code in the class. This means that there was no branch of code that was left untested.

Graphical user interface

Description automatically generated

This screen shot shows that the java classes all have over 80% coverage.

It is important to ensure that code is technically sound. I accomplished this by utilizing the training that I’ve received from numerous courses and after writing the code treating it like it was written by somebody else and it’s my job to find the mistakes. There were a couple of spots where the code I wrote for the classes didn’t return proper responses for incorrect data, so I had to revisit that code to ensure it could be tested properly.

A screenshot of a computer

Description automatically generated with medium confidence

This screenshot shows code created for the ContactService class that throws exceptions when invalid arguments are used for the Contact class. This allows for the code to be tested with bad data.

I ensured the code was efficient by reusing methods and techniques that I had used in other tests. This allowed for a more streamlined process developing the tests and not having to reinvent the wheel.

Several different techniques have been developed in the field of software testing. These techniques can be separated into functional and non-functional testing. Functional tests are the tests that are done to ensure the requirements are met. Non-functional tests are the tests that focus on the performance, usability, security, and other areas that the software user’s needs are being met.

The software testing techniques that I have used for each of the milestones are functional as the assignments were to create software that satisfies the requirements and test them. I primarily used unit testing by using JUnit testing. In the individual tests I used the state transition technique. This involves changing the input values to get every possible outcome. I also used boundary value analysis technique which involves entering input values that include maximum and minimum edge cases that are both valid and invalid entries.

The software testing techniques that I have not used in the milestones so far include equivalence class partitioning, decision table-based testing, or error guessing. In the equivalence class partitioning technique you divide the test conditions into partitions that could be treated as the same. An example would be if you wanted to test number inputs you might be able to consider one negative number the equivalent of testing all negative numbers. In the decision table-based technique, you create a truth table for all inputs and rules and have a final row if the input is valid or not. Since our input wasn’t that complicated these techniques were not needed. I also didn’t need to use integration testing as all the milestone assignments were stand alone.

The practical uses for each technique depend on the complexity of the software. If there are a lot of inputs then a decision table-based test would be practical, but it isn’t in systems with only a few inputs. Equivalence testing is very practical and should be used with caution as the tester can group several inputs together that may have different results such as a small negative number and a large negative number. Boundary value analysis testing is the most practical as the tester should ensure that all edge cases perform as expected. State transition testing is also very practical as testers should try to test state changes as thoroughly as possible.

My mindset when developing the tests was to try to find a flaw in the code. I think this strategy is best when the same individual both writes and tests the code. While eliminating bias is extremely hard in these situations, I accomplished this by envisioning myself with two different roles, one being the coder and the other one being the tester. By separating the roles in my mind, I am not as married to the code, and it is easier to find errors. The importance of being disciplined is paramount as it can be quite easy to become so focused on getting the job done that corners are taken. This opens the door for mistakes and in this industry an extra step early in the process will save quite a few later. Taking the extra steps even on small projects allows for good practices to be established and to become habit in the long run.