**Project Two**

Noah Coleman

Computer Science, Southern New Hampshire University

CS-320-H7221

Professor Bolton

August 14, 2022

**Project Two**

**Summary**

My testing approach was to follow the software requirements as closely as possible. I wrote the application to the specifications given by the client, and then used tests to ensure that my program met those requirements. For example, in the contact class the requirements were that each contact object had certain attributes, and those attributes had to meet certain criteria. My tests created objects with and without those attributes, and then tested that they were created correctly or incorrectly as expected.

In general, I know that the quality of my JUnit tests was good because of execution and coverage. The coverage rate of my test package for project one is 86.4%. This means that almost every command in the program was tested. Some of my tests had more coverage than others, the contact service test had 99.3% coverage while the task test only had 71.0%. This means that moving forward I should focus on the lower coverage areas, but in general there is a strong testing environment.

I made sure my code was technically sound by following industry best practices and creating test to check correct implementation. For example, I wrote a test to ensure that the contact object created does not have an ID that is too long or null:

Graphical user interface, text, application, email

Description automatically generated

Since both these test pass, I know that any contact created with an improper ID will throw an error and need to be fixed.

I ensured my code was efficient by creating test that check to see if updates made to an object are performed correctly. I want to know that an object is updated correctly, so that the code is not running with out-of-date information. To test this I wrote: Graphical user interface, text, application

Description automatically generated

Since these tests pass, I know that objects are being updated correctly, which means the code is working efficiently.

**Reflection**

The primary software testing technique used throughout this project was dynamic unit testing. First, I had to code each class to meet the specifications of the client. Next, I had to test each class to make sure that it executed correctly. Using Junit testing, I was able to write tests for each method in each class, and then run them one class at a time. For example, when creating a new appointment object, I was able to test that an appointment was added, and then that the appointment could be retrieved. I also used the static testing method of code review. By reading over the code line by line I could find errors in syntax or logic. This helped me fix issues not directly shown using dynamic testing.

One technique I did not use in the milestones was boundary value analysis. I did not test to see what would happen near the border of where a correct or incorrect object could exist. This means there are potential errors that my test did not catch, such as a unique ID with the value of 1. I also did not use equivalence partitioning. This testing technique could be useful in testing the unique IDs for my class objects. It would test that the ID existed in the acceptable number range which would be from 1 to 9999999999, maxing out at ten digits.

Different software development projects have different testing needs. In general, all projects will have need for both static and dynamic testing at some point in the development process. Code review is useful for finding basic syntax and logic errors. When writing code, it is easy to make simple mistakes. These may be found more easily by a colleague or reviewer who hasn’t been staring at the same code for hours. Boundary value analysis is very useful in cases where there is a chance of input being received near the edge of acceptable input ranges. Making sure that the software reacts correctly here will stop unlikely but important glitches.

Partitioning can be useful in projects dealing with large quantities of equations of numbered inputs. Knowing that the computer will accept input in the partitions range will increase the speed and certainty of testing. Junit testing is practical for use in most projects, as it runs class by class test to make sure everything is executed as expected. This will let you know that things are be written and stored properly and that class objects are interacting correctly.

While working as a software tester for the project, I had to ensure that my mindset was focused on functional and efficient code. It is easy to get attached to code and not want to admit when it is not working correctly. Like anything else you create, you have pride in the code that you write. After writing code for this project, I had to step back to look at the code externally and try to distance myself. It was important to be cautious in this step, and to avoid being defensive. For example, when writing the code that gave each object a unique ID, I realized I hadn’t taken the correct approach and that the test was showing this. I had to decide if I should rewrite the code to correct the issue, or rewrite the test to pass. As the mindset should be for functional code, I chose the former.

Bias is inherent when testing your own code, and it is important to try to limit it. It is human nature to reject something that is telling us we are wrong. If the test is telling us that our code is not working correctly, our initial reaction is to blame the test. It is surely a concern on large project that bias will exist when asking a programmer to test their own code, so it should be left to an outside tester. For example, when I was testing the appointment class for the project, I realized that the system was not storing the objects correctly. The tests showed this as they were failing to execute. The appointments were not being added to the array, so the system could not retrieve them. It was easy to blame the tests for failing, showing my bias, but after rewriting the code the tests passed.

Discipline is important in the software engineering profession. It is easy to write something that will appear to function and pass initial test, but the goal is to deliver a product that will work as intended upon release. It takes discipline to use best practices from start to finish. It is important not to cut corners when writing and testing code so that the program will hold up after release. During this project there were opportunities to adjust the test so that they would pass, while knowing the code was not correct. For example, my initial unique ID generation was not functioning properly, and the tests were failing. I had to be disciplined and rewrite the ID generator, and not the test, so that the final product would work as designed.