Samuel Nawaz

CS-320 Module 7 Project

Prof. Rangitsch

17April2022

**Summary**

My approach to the JUnit was to ensure that every requirement was soundly met. When writing code, I would first draw out an outline of what the requirements were and how they would interact with each other. Then I would code based on that. Initially, I tried to write the whole code and then test, this was a major misstep on my part as I had to scrap what I did and redo everything. However, as I continued coding, I began testing in increments, this really helped me to ensure that everything worked as expected. The more I did this, the higher my coverage percentages got. I set a benchmark for myself of 80% minimum, this ensured that if a test fell below 80%, I would go back and re-analyze what I had just written, in the end this approach saved me a lot of time as I seldomly had to re-check large portions of the code. I chose 80% as my benchmark to ensure that the code was sound, however, anything higher would have taken a lot longer to accomplish. Though it started out slow, as new modules came up it became easier to hit and exceed this benchmark. The higher the percentage the better my code was. I utilized strings and arrays to help ensure my code was good. During the module three assignment ‘Contacts’, I utilized a list to help maintain the incoming data. To store the list of contacts I used: (private ArrayList<Contact> contacts; public ContactService()), and continued from there.

**Reflection:** Testing Techniques

During this assignment I relied on unit testing to help ensure that the code was correct. The unit testing was incorporated when I would test each fragment of the code. By ensuring that each section worked as expected, it became easier to ensure that the code as whole worked properly, and if there were any issues, it was easier to pinpoint exactly where the issue was. I also incorporated acceptance testing. The acceptance testing was incorporated by ensuring that each requirement was included in the code, and then testing to ensure that each requirement worked as expected. This part also included, to a lesser extent, functional testing. The functional testing part was incorporated by verifying that the code functioned as expected. During this exercise I did not utilize black box testing. I can safely say this because black box testing would have required that I have no prior knowledge of how the system was supposed to function and absolutely no access to the code. I can also rule out using regression testing as there were no deployments of code. Finally, I can rule out exploratory testing as I did not search for any hard-to-find defects, and I never deviated from the given parameters. Unit testing is designed to test the code as it is being written to ensure that it works correctly, before too much has been written. Acceptance testing occurs after unit testing and is used to ensure that the end users can use the software and that the customers’ requests are met. Functional testing ensures that the software is working as expected, without issue. Black box testing is testing without being able to see the code or have any knowledge of the system, it is used to test the system with external factors. Regression testing is used to check for new defects that may arise after an update or code change occurs, it usually occurs routinely after the code is deployed. Exploratory testing is used to learn how a system works. It typically includes looking for hard to find defects so that they can be re-created, studied, and corrected.

**Reflection:** Mindset

The mindset that I encompassed during this project was that of a very cautious developer. I spent time ensuring that each requirement was met before I ever began writing the code. I incorporated routine tests to ensure that no major defects or errors arose. I took a very literal step by step approach to this assignment. I also focused on keeping everything organized. By organizing every requirement and its subsequent sub-requirements, I was able to create better outlines and was able to code much more efficiently. This also allowed my code to come out better, which resulted in better, more efficient, testing. I also had a very strict limit on my coverage percentages, anything that fell below 80% would automatically lead to me stopping and addressing the issues that arose. I was very firm on this and stuck to it without exception. Even if the code hit 79%, I would go back and make necessary revisions. This removed any chance of a biased thought taking over as I relied on the system to be my peer review, and the system is always right. Without the system giving real time feedback my code would not have worked correctly. As a developer, it is important to have peer reviews and system feedback to ensure your code is spot on. In my previous role as a tester for the state of NC I would often run into issues where developers would argue tooth and nail that their code was correct when in reality, they were loaded with errors or had missed requirements al together. I remember one time where a developer was arguing that his code was perfect, however, it turned out that he had accidently withheld a critical portion of the code, this was an easy fix that could have cost the state millions had it been deployed. This extra level of verification is necessary to help ensure that bad code is not deployed. This level of discipline really helped me to present the best possible code I could come up with. In the long run, this level of discipline will help me remain vigilant when writing code and/or testing code. By sticking to the rules, I have set for myself, I will not allow a batch of bad code slip past me. I believe that this level of discipline is necessary to properly design code and/or test software.