**7-2 Final Project**

Cameron Binda

Southern New Hampshire University

CS 320: Software Testing, Automation, and Quality Assurance

Omar Toledo Lopez

3/2/25

**Summary**

My approach to testing was to make sure every function and branch was covered. The software I wrote was pretty simple, so I thought it wouldn’t be much more time invested if I wrote tests until I got 100% coverage, rather than 80% coverage. After I messed up the contact, I was especially motivated to review the requirements multiple times, and make tests for everything. For example, I had tests with good and bad input for everything. There were expected errors for input that was too long or null, and boolean assertions several things, like adding objects to the array list, whether good or bad. I had multiple objects created with the same IDs and tried to add them. I asserted that the first would add, but that the second wouldn’t. The adding methods return the appropriate boolean, so that’s how the assertions would know if it was successful. I also had out of range IDs for updating and deleting. Efficiency wasn’t much of a concern for this project. Usually, for loops are the easiest things that could make your project inefficient. In this case, I needed to use a for each loop to “search” through the lists of objects, since their size would always be unknown. In this case, I only used loops where I needed to. I tried to improve the efficiency of the tests by using a @BeforeEach function, but I’m not sure of the effectiveness of it. It did clean up the code a little but I feel like it didn’t matter because it does everything in @BeforeEach before each test.

**Reflection**

One technique I used was unit testing, because that’s what this class is about. I developed and tested each piece of functionality and then moved up to the next. More specifically, I tested the object classes first, and then the service classes to make sure they worked. Within both, there were multiple units to test. For the objects, there was the constructor, setters, and getters. For the services, there were add, delete, and update methods with multiple branches depending on the needs. I also used automated testing and black-box testing. JUnit is an automated testing software that can verify expected inputs and outputs of a piece of software. I had to create test classes for each java class and code the inputs and expected outputs for each unit. Black-box testing is just functional testing, so, like I just said, I coded the inputs and outputs expected from each unit.

For the techniques I didn’t use, one was non-functional testing, since the code is pretty simple. This technique tests performance, usability, and security. Usability is not really applicable for this project because I have a specific set of requirements for inputs and their outputs. There also wasn’t anything to test for performance because, for this project, the most intensive section was the for each loops, which loop as many times as the list is long, and are unavoidable when using array lists. Security also wasn’t really mentioned at all, but I did use private variables so they could only be updated through functions. I don’t know much about penetration testing so this security measure wasn’t tested. I didn’t do any white-box testing either. This would be something like stepping through the code. You could watch the code as it executes and make sure variables are passed, functions are called, and branches are taken. You could also insert errors to make sure the wrong thing happens. I didn’t really do any level of testing above unit testing, but I’m not actually sure if testing the services counts as integration testing, since I’m not only testing the service, but also the objects that they use. According to the definition, it seems that there needs to be a little more integration, such as a main file with inputs, or somewhere the user can enter inputs.

Unit testing is very useful for catching problems early in any project. Although testing it all together may be more intuitive, it’s better to test the smallest pieces of functionality first to ensure anything using those units won’t have inherent problems. Automated testing, as I’ve learned, is much easier to use compared to my previous method, which was setting up a main file that prints all my outputs. I instead write the inputs and outputs, and it gives me a summary and where any test failed. Black-box testing is a simple way to test. You just have a set of expected inputs and outputs. It doesn’t, however, consider if what's happening inside the code is actually correct. That would be white-box testing. It’s useful in making sure what’s going on in the code is correct. Step through is a common method for this technique. You can watch the code execute line by line, and still ensure your inputs and outputs are correct. Non-functional testing would be more useful for this project if I developed the entire application, rather than just the backend. This is more for testing if the software is usable, performant and secure. Much larger applications and higher testing levels would benefit the most from this as performance becomes a problem with the length and scope of the code, usability is more for frontend or the higher testing levels, and security is more useful for sensitive data. The testing levels above unit testing are, of course, useful for their respective levels of composition. It’s necessary to make sure multiple pieces of functionality work once they’re connected.

My mindset wasn’t much different than any other class at first. For the contact class, I just followed the tutorial and then wrote tests based on the code, because that’s what the announcements and rubric said. After I realized I messed up, I was much more cautious, at least about the requirements. As for testing, I feel like I was pretty cautious since I made sure my tests covered 100% of the code. I also reviewed my code many times to make sure it made sense, especially after the contact class. I tried to limit bias reviewing the code by just reviewing the requirements many times, making sure I was interpreting it the only reasonable way it could, and making sure my code met the requirements. I was obviously biased to believe the tutorial for the contact class was complete, but, again, I tried harder after that. It’s important to be disciplined for reasons like this. I underestimated the effort required for the assignment, rushed through it, and did it wrong. I was basically cutting corners by not really reading the rubric. This is also why it’s important to not cut corners. You can deliver a non-functional product without realizing it, simply because you thought you had it. To avoid technical debt in my career, I’ve pretty much already said what I’ll do. I’ll make sure I read the requirements many times, and compare my code and tests against the requirements many times to make sure I did it right.