**Summary and Reflections Report**

When writing software I always have the requirements up in a quickly accessible window to reference. Code and tests should always be written to satisfy the requirements in the simplest and most efficient way possible. Some development can even be test driven, where tests are written (to the requirements specification). I know my tests hit a high level of effectiveness because I tested for every requirement and I hit 100% JUnit code coverage in the Eclipse IDE. I made sure my code was technically sound by testing all the data changed by testing the requirements and ensuring the correct errors get thrown. I know the code is efficient because not only does it exit as soon as possible in any error situation but there is no code that is not covered in the tests that are written to the specifications of the project.

I employed many testing techniques on the project to exercise the code as thoroughly as possible. When writing the tests I made sure to have a list of requirements that I would mark off as I wrote tests to fully exercise the requirement. I would make sure to cover all cases of the requirements, such as when it should fail and when it worked whatever action was performed successfully. I made sure that the tests reached full coverage (exercised every written line of code) not not only to make sure code was efficient but that there were no unintended consequences when any code ran. I utilized error testing to ensure that when the code failed it did so correctly and in a controlled manner. In the later modules I used test driven development by writing the tests to the specifications before I had even begun writing the code.

Even though I employed many testing techniques there were many that I didn’t or couldn’t use. I probably could have had more tests for boundary testing, I have written a lot of java in the past so I didn't see it as necessary because I was using self created objects but probably would have done more if working with created objects, lists or databases. The code wasn’t really used together (other than the service class and the objects) so I was not able to test if any of the code had adverse or unintended effects to the other code. These types of errors can build up over testing a large codebase so it is important to test the integration of code as soon as possible. Also because I was unable to assemble a full program, I could not do any true functionality testing by testing any use cases from the users perspective. I also worked alone on this project so I was unable to leverage any code review and/or outside testing. Having other people look over and test your code is important because as the developer you will have an innate attachment and bias towards what and how you have already written the code.

Having worked professionally in software testing, I always carry the mindset of a tester when writing code. I want to write code that can be easily tested to make sure it fits the specified requirements. But it is even hard for someone who is testing minded to limit their bias about their own code. One of the best ways to limit bias is to write the tests before the code has been written, so that the tests are being written to fit the requirements and not the code. An important tool in knowing that you didn’t cut corners is maximizing code coverage. Code coverage can ensure that every line of code is tested. The best way to avoid technical debt is to test early, to make sure that any code put into a project is working as intended, and incorporate regression testing often to ensure that recent code changes don’t negatively affect previously tested code.