Summary

My approach was strictly aligned to the software requirements throughout all stages of development. All the code written was either made specifically to fulfill the software requirements, or to make the software more robust for testing purposes and future usage. For example, overloading the equals function for classes was not a requirement, but it allowed for more efficient testing methods and could be useful for future revisions of the software.

I believe that my JUnit tests were very effective at testing the validity of the various classes and services written for the project. Referring to the coverage percentages, the class percentages ranged from 94%-97.4% and the service percentages all reached 100% coverage. The minimum coverage percentage was 80%, so as a result I would say that my JUnit tests were effective.

I ensured that my code was technically sound by going through several revisions of JUnit tests. Looking back at my original milestones, I believe that my proficiency at writing JUnit tests increased significantly. For example, I began using more intricate testing tools such as assertDoesNotThrow() which allow me to test for both exceptions and the lack of exceptions.

I ensured that my code was efficient by avoiding overusing exception throws. I wrote most of the possible exceptions into the object class instead of the service class so that I could limit the total number of exception throws. Inside the JUnit test themselves, I increased efficiency by using the @BeforeEach signal alongside various variables that would be used throughout the tests.

Reflection

The software testing technique that I employed during this project includes equivalence partitioning, boundary value analysis, use case testing, statement testing, and decision testing. Equivalence partitioning, boundary value analysis, and use case testing are all forms of black box testing that allow for the testing of various possible values and use cases both through automation and by hand. All three of those testing methods work hand in hand to test various possible input for the functions. Statement and decision testing are examples of white box testing which can be automated and allow for testing of various possible values and the validity of those values.

Some techniques I did not employ include decision table testing and state transition testing. These two are examples of black box testing and they allow for a visual representation of the system at play. This project did not include any real state changes, nor did it involve complex conditional trees so the usage of these two testing methods would have been superfluous.

When working on this project I adopted the mindset of someone aiming to break the program. In game design, a related field, testing needs to be extraordinarily rigorous and as I have knowledge in that field, I can adopt the mindset of someone who wants to break the system easily. It is important to be cautious about how rigorous a single test can be. As a tester you really must try everything within reason, no matter how confident you are there could always be another case that throws everything off.

I attempted to limit bias by splitting up the days in which I wrote the code for the classes from the days I wrote the tests. The purpose of this is because coming in with a fresh perspective naturally could reduce bias. I do believe that in a greater project, bias is an issue and it's blatantly

obvious too. When building a game a few months ago, I tested everything constantly while working on the game and I thought I had a polished experience. Within fifteen minutes of handing off to a friend to test it they found several bugs that I was too biased to even consider as issues with the game.

Having discipline when programming or testing is essential because every cut corner is a possible fatal error that could have been caught. Most testing methods are not difficult to run and require at most minutes of work each. Avoiding a small time sink to prevent a catastrophe is never a bad idea. I would avoid technical debt by ensuring that I always give my best to every test no matter how simplistic the code is and I would also attempt to get coworkers to review my code and tests to ensure that I am not missing anything due to bias if possible.