**Project Two**

Mitch Sfakianos

Software Test Automation & QA

Angel Cross

August 14, 2022

My approach aligned with the software requirements to an exact extent. For each requirement in the appointment, contact, and task classes and their services, I surveyed all the requirements and refactored and adjusted the code to all properly align with standards. Specifically, each private variable in the main class initializations was checked thoroughly to match the preliminary software requirements such as not being able to be null or over certain character limits. I ensured that my code for the JUnit tests was technically sound by researching the functions that I was implementing from different libraries and debugging the code to make sure the tests were all passed and output proper errors when applicable. An error that I ran into was that I had tried to do an assert false method on a feature that actually threw an error, so I fixed this bug by having it return false instead of throwing an error message, which made the tests pass. I also ensured that the JUnit tests were efficient since they do not over-test, cutting out some repetition that was avoidable. Since I used the same methods of achieving requirements for certain variables and I knew that those methods were all identical, then the test would only go through one variable test rather than checking them all.

The software testing techniques that I employed in this project were testing whether or not variables could go through when they don't meet the service's requirements. Another technique I implemented was making sure that identities were unique within a database. Both of these requirements included my JUnit tests and manual tests through proper debugging and rewriting of code. The practical uses of dynamic testing such as JUnit tests are broad since they can be very handy for large software that needs a lot of tests to be done instantaneously. Plus, unit testing tests pieces of the software on a class basis, so if the main service is not working and the error messages are not clear, then unit tests could be run and failures will be readily identifiable, making the broken piece of code that caused the whole service to crash recognizable much faster than a manual readthrough of the whole code. A type of testing that I did not implement though was use case testing by running the services manually.

In acting as a software tester, I employ a moderate amount of caution. I generally follow guidelines and stick to represented or documented examples of the methods that I am using to test my own code. The amount of caution I would use in a project correlates with the size and complexity of the code base. Since these classes were all small and nearly identical, the tests were not as broad as I would have made them if it were within the context of a more complex project. It is important to appreciate the complexity and interrelationships of the code one is testing since if even one piece is astray, then the whole cannot function.

I tried to limit bias by implementing tests for features that I was sure would not necessarily need testing. For larger scale projects, being sure of yourself and your own code is not enough for it to function properly, and certain pieces of code can always be overlooked by someone who has been writing and focusing on it for a long time. The unit tests act as a fresh set of eyes in this way. A biased software developer who was overly confident could end up having a lot of failed tests revealed through a thorough unit testing.

Being disciplined is important in a commitment to quality as a software engineering professional. Recreating these similar services requires attention to detail in refactoring the code and hashing out tests for each facet of the program. It is important to not cut corners when it comes to testing code, since you never know where something could have gone wrong.