**Project Two**

My testing approach was *defined* by the software requirements. More specifically, the way I wrote my tests and decided what to test was defined by keywords withing the stated requirements. In the rubric, every Contact class member is given a length constraint and must not be null. Therefore, I created tests for each Contact class member that verify the length constraints and the non-nullability of the members. The only Contact class test that I wrote that was not influenced by software requirements was the testContactCreation() test which tests the Contact constructor method when all fields are valid.

The ContactService tests were more straightforward as there were fewer invalid use cases for which to test proper error handling. Outside of testing valid use cases, the ContactService tests only needed to cover use cases where a Contact with a duplicate ID is attempted to be added to the ContactService, where an invalid field name is passed to updateContactField(), and where a non-existent contactId is passed to updateContactField(). Other cases, like when a null value is passed to updateContactField() are covered by the tests for the Contact class as updating the fields’ values use the Contact class’s setter methods.

I am confident in the coverage percentage of my Junit tests because each member of both classes has tests written to cover all invalid use cases as defined by the software requirements. It is entirely possible that other invalid use cases exist but without these constraints being defined, I cannot test for them.

I ensured that my code was technically sound by writing code that met the bare minimum requirements as laid out in the rubric. The only decision I made outside of the requirements was the data structure in which to store Contact objects for the ContactService class. Even then, the Contact objects needed to be stored in key-value pairs, so a HashMap was an obvious choice. Writing code to match the minimum constraints as defined in the requirements ensured that what was created is exactly what was asked for. For instance, in the Contact class, the only requirements for each class member were a length constraint and non-null values (excluding contact ID which was made non-updateable using the final keyword which requires no testing), so I wrote tests for each class member that validated only these constraints as seen in lines 19 through 28 of the ContactTest file.

I ensured that my code was efficient by only writing code to meet the minimum requirements as defined in the rubric. As stated above, the only code not derived directly from the software requirements was the specific data structure used for the ContactService class and even that was an obvious choice. By writing code that meets the minimum specifications, the classes and tests maintained maximum efficiency.

In Project One, most of the tests written could be placed under the umbrella category of *automated unit tests*. This is because the tests were all conducted using automated testing software (JUnit) and almost all the tests validated the bounds of one function. The only exception to this were the tests which validated the deletion of objects from their respective services. For instance, the AppointmentServiceTest class from the 5-1 Milestone assignment contains a test which validates that a previously added Appointment object is deleted when the *.deleteAppointment()* function is called. For this test to work properly, the Appointment object must first be added to the AppointmentService instance by calling *.addAppointment()*. Because this test covers more than one function, it would better be described as an *integration test* or, a test that tests multiple components to ensure that they work together as a system.

The main type of testing that was not conducted through these milestones is *manual testing* in which I would launch the software and manually explore and test the various features of the software by hand. Doing so would also incorporate other testing methods like *system testing* in which the testing is focused on the system as a whole and *usability testing* in which testing focuses on the software’s user-friendliness. These types of testing were not performed simply because the software components which I was tasked to create were not in any form that manual testing could be performed. The code written for these assignments lays out *models* but no *views* or *controllers* that would enable a person to use them. With just the raw classes, only automated unit testing, *acceptance testing*, and *functional testing* are possible.

After completing this project, I have a better understanding of the practical uses and implications of the various types of testing methods mentioned above. It now seems to me that while each type of testing is important, the order in which these testing methods are employed is just as important. Testing should start at the ground level with a combination of automated unit testing, acceptance testing, and functional testing. I believe this to be the case because these types of testing can be performed on the most basic forms of the desired product’s code and ensures that as the development lifecycle progresses, the client’s expectations are met and most importantly, the smallest units of the software work as intended. Additionally, these specific testing methods should be used throughout the entire SDLC to ensure the end-product maintains a solid foundation. Further into the development lifecycle as components are developed with which users may interact, manual testing methods should be implemented to ensure both that automated coverage is as comprehensive as possible and that the software is user-friendly enough to be used with little to no prior experience.

In acting as a software tester, I employed caution by closely examining the stated requirements for each class and ensuring that each class included features that exactly matched the requirements. Appreciation of the complexity and interrelationships of the code being tested is important as it aids in understanding potential edge cases that may not be obvious from the stated requirements. For example, my tests include a unit test for deletion of a non-existent class object in each of the Service class tests.

I tried to eliminate bias in review of my code by comparing the implementation against the stated requirements from the rubric. My line of thinking is that requirements-driven development may not be free from all bias, but it can be free from developer bias.

Maintaining discipline in my commitment to quality as a software engineer is important as it protects the end-product from defects that could be overlooked if not for careful attention to detail. Additionally, the further that defects are able to proliferate past the initial development stages, the more costly and complex they become to fix.