**Project 2 Summary and Reflection Report**

This course has been monumental in my progress as a programmer, and that isn’t lip service. I admit that I had grossly underestimated the importance of proper testing, and the true effort it takes to test effectively. When presented with the challenge of unit testing the three features of the grander project, I had to carefully consider the approaches I would take to test successfully. Each test required a different, but similar, approaches depending on the requirements. The first assignment, Contact and contact service, required size limiters and could not be null for each of the values. To test this, multiple unit tests were generated for each variable in question; for example, the contact first name had to be no more than 10 characters and could not be left null. To test this, I generated a test variable that passed in values that were within the parameters and asserted that the contacts were being created when passed through the “addContact” by using the “AssertEquals” override. When passing the new variables in comparison of what the variables should be, the statement would come out as “true” to prove that the method worked. I also verified failed tests to make sure the exceptions are pulled in a failed test. When testing the delete method for deleting a contact I used a count method to count the number of contacts generated, then compared it to the number after the “deleteContact” method was implemented. If the number came out smaller, then it worked properly. Each contact also had to be associated with a unique ID. I tested this by printing the unique ID for the created contact after a new contact was generated. This was easy to test, I verified that with each new contact, a unique ID was properly assigned within the parameters set. The appointments were a little bit different, as I needed to reference a calendar library and establish schedule sessions. The testing process itself was about the same, creating and deleting schedules, but required the breakdown of multiple schedules on a single calendar day. This is tested by viewing printouts and counts, as previously done, but also verifying that coexisting schedules weren’t also deleted when a single schedule was.

The value of a test can be determined the coverage of that test. The values of my coverage tests came out to about 96.4% for my Contact Service test, and 100% for my Contact test. The task service test came out to be 87.9% and 100% for the task test. Lastly, the appointment test came out to be 100% with the appointment service being 98.7%. This properly defends my overall testing quality of the generated classes and the tests generated in testing. Coverage accounts for some of the value, but it also needs to be technically sound. My code was technically sound as demonstrated by the simple solutions of lines 50 through 52 in the task service test. Putting the print lines in the proper place would give us the proper count, if the print lines were above the execution of the method, it would be completely different. This is perhaps a simple example, but careful consideration was used throughout the entirety of the project to make sure that the processes were executed with the expectation of the Java code being executed from top to bottom. As far as the efficiency is concerned, the longest code (Schedule service test) executes in .069 seconds. The code was designed to run more efficiently using hashmaps. Originally the code was written using array lists, the same code took about .09 seconds to run. On a much larger scale, this can contribute to a significant amount of additional and unnecessary lag; greatly hurting the codes’ efficiency. When I finished originally writing the code, I had also written unnecessary lines that slowed the process. For example, I had originally had an individual count function written into each test. I streamlined the process, and created a single method that could be called (evident in lines 52 and 63 in the schedule service test class).

When it comes to testing techniques that I employed, I relied heavily on boundary value analysis (BVA) within the unit tests. BVA was used to verify that the character limitations set were effective and through the proper exception. If no exception was thrown, or if the exception thrown was not the appropriate one it would signify a flaw in the code. BVA is a form of white box functional testing that tested the functionality of the written code with little reliance on the structure. However, the codes’ structure, as mentioned earlier, was also a point of concern when writing the code. BVA was easily verified in the contact service and the task service test by creating variables both within and outside of the limits. As for the appointment service I used code that would generate a unique ID automatically using a randomizer set to 10 characters. With this method, I tested its capabilities by printing out the unique ID with the schedule and could easily count the generated variables. I didn’t implement every type of testing technique as it would be unnecessary. What I mean by that is that implementation testing is reserved as a final test before release, and compatibility testing would be necessary if we were concerned about using different operating systems (not a concern at this time). Security testing would also have been unnecessary as it did not pertain to this particular project, it is to be noted, however, that security testing is a high priority and should be carefully reviewed prior to code release.

The mindset I maintained was a focus of the limitations, restrictions, requirements, and efficiency of the code. My primary focus was fulfillment of the requirements asked to be fulfilled by the client. Without fulfilling the requirements, there would have been no purpose for any other focus, as this would have deemed the rest of the project as unnecessary. I wrote the code by referencing the requirements first, but almost equal in priority was the limitations. Each limitation and specification was very black and white, therefore it was very important that I maintained caution when employing the code, as failure to fulfil these limitations would also not be satisfactory to the requests of the client. The relationships between the codes were also a factor of concern, as each aspect of the program will at some point be unified into a singular working project as in the classes would be unified and accessed through a main class. With the potential of the classes being conjoined, the way that the code is written should be easily adaptable to the other aspects of the codes. Through testing I was also able to verify that the setters and getters in the original code were communicating and variables not only passed through one aspect of the code, but through any respective class that utilized said classes’ methods. To clarify my point, When I used the addContact method in the Contact Service class, I was able to create a new object in the Contact Service Test by referencing the class’s addContact method, which utilized the Contact class’s set parameters and variables. I tried to avoid any bias nature within the code by streamlining and simplifying the process to remain as general as possible. This limits the potential of a biasness while also establishing more forgiveness and flexibility in the usability of the code. I used standard practice as well, to try to prevent a biasness. I tested the code to all of the test variables I could imagine while prioritizing the testing in regard to the described requirements, further enforcing the prospect of an unbiased test. It can be a potential problem if a programmer uses biased techniques to test their code, preventing truthful results from displaying a failed test; which could in turn pass a test that is not as bug-free as it could be. To demonstrate this we can view the task service portion of the test. My addTask method worked in my original writing of the code, however the efficiency of the code was lacking because I was using an arrayList to store my values. When I refactored the code to include a hashMap, I was unable to get the “addTask” method to function fully. I could have tweaked this code to get it to “pass”, however that was not only morally wrong, but it would have demonstrated a bias nature to my code. I have since fixed the code to have a working addTask code, but I did not falsify a passing test out.

This last example regarding the addTask method can also be used as an example of my commitment to quality and the discipline I have to maintain as a software engineer. I didn’t want to falsify my test results, although it could have been done. By choosing not to take “the easy way out”, I decided to allow the test to fail and continued to work on a solution even past the deadline. I am not thrilled that I didn’t submit a 100% fully functional code, but I also don’t want to allow a code that isn’t fully functional to be put into service. Cutting corners can result in a less functional, and potentially harmful, code and should be avoided at all costs. If I had written the code to strictly pass the test, it would not be a functional code in any other aspect. As I continue on as a programmer I will reference both my well written code, as well as acknowledge my not-so-well written code, like using an array list instead of a HashMap to improve code efficiency. By acknowledging the latter, I can strive to consistently fulfill more functional code and continuously grow as a developer.