**Project Two**

When developing tests for Project One, I focused specifically on testing requirements. Almost all the tests throughout the Task, Contact, and Appointment classes were made to test a certain requirement. For the Task class, one of the first tasks I developed was a test to ensure that the task ID was initialized following the ten-character limitation and that the ID would not be left as null. In lines 8-13 of TaskTest.java, I initially tested that taskID would return the expected output when it was given input that should have met all the requirements. In the second test of TaskTest.java, from lines 16-23, I tested with incorrect input to ensure that despite illegal input, taskID would still not be invalid by the program requirements. I continued a similar structure throughout my tests for Task.java. For the name data member, I tested first using a correct input based on the specified requirements, then tested with incorrect input, and so on. By repeatedly using this structure, I was able to ensure that I tested the program requirements of each class.

With regards to the coverage percentage of the tests, I ensured the tests were effective by ensuring that each test file reached 100% coverage. When developing the tests, I would first write out the tests I felt were adequate and upon finishing would test the coverage of my tests. If there was any gap in coverage testing, I would then add another test to ensure more thorough testing. For example, when creating TaskService.java, I had written almost all of the tests I submitted before testing for coverage. However, when I tested for coverage, I noticed that there was a gap in testing that TaskService was a child of Task. To fill in this gap, I created a new test that I called TaskServiceChildTest spanning from lines 7-16 of TaskServiceTest.java. At line 10, I created a TaskService object and on line 13, I then created a Task object that was initialized with the prior TaskService object. Since this would only be possible if TaskService extended Task, I was able to reach 100% coverage with this additional test.

I ensured my code was technically sound by ensuring that I tested expected and unexpected input, as well as developing my code with the program requirements in mind. Some of the program requirements also seemed to reinforce technically sound code, which was also useful. For example, in the Appointment class, one of the requirements imposed on the date data member was that the date had to be either a current or future date. This reinforced technically sound code since a date set in the past would not make sense per the context and requirements of the Appointment class. To test for this, I created a test titled testDateTooOld in AppointmentTest.java spanning lines 47-57. In this test, I tested to make sure that even the earliest possible date would not “break” the program. To do this, I used Long.***MIN\_VALUE*** to initialize a date. Using this test, I was able to verify that my code in Appointment.java was technically sound.

I tried to ensure that my code was efficient by using the same basic structure throughout the testing process of the Task, Contact, and Appointment classes. For example, in in AppointmentTest.java, I tested that the appointment ID would not initialize the taskID data member with both a valid and invalid value in testTaskID() from lines 7-14 and testTaskIDTooLong() from lines 16-23. Similarly, TaskTest.java and ContactTest.java tested their IDs in the same way in their respective taskID and contactID tests. While these IDs were all slightly different, they were all tested the same way by first testing that the IDs would initialize correctly with valid input and then with invalid output. Throughout all the modules, I would try to mirror similar tests on similar data members so that the tests could be efficient.

Throughout Project One, I primarily used black-box testing techniques. Black-box testing primarily assumes that programs are being tested on the basis of the project’s specifications. I made sure to follow this methodology by testing the program’s requirements throughout the different classes that were developed for Project One. I also used dynamic testing when running tests through Eclipse’s environment. I didn’t utilize static testing as much since the tests I was developing were primarily for small pieces of code.

Black-box testing is always a good methodology for projects that have many different requirements imposed on them. Since it’s a testing technique that’s derived from testing those requirements, it can help ensure that all those specifications are met. It would be a good fit for another project like Project One where a lot of the software being developed needs to meet certain parameters. Dynamic testing can be a good fit for projects that can be easily tested with test data, whereas static testing would be a better fit for work products that are more abstract and are early in development (Hambling, et. al, 2019).

Throughout working on this project, I exercised caution by trying to consider different scenarios and possibilities for why a test might either fail or succeed by not immediately assuming that the most apparent answer might not be the correct one. For example, when I developed a test and ran those tests, there were a few instances where a test initially failed. While this first seemed to me that it indicated a problem in the code being tested, I often found that when running a new test for the first time, the problem might actually be in the test. I also tried to limit bias in the review of my code. Even if I finished developing a class that I felt was correct, I tried to keep an open mind to different solutions. When creating a test for TaskService.java, I noticed that tests involving the tasks map often failed after addTaskTest() in the delTaskTest(). Upon further inspection, I realized that the tests weren’t actually failing because there was something wrong with the tasks map, but that the map in the Test instance was retaining values from previous tests when moving on to the next one. To solve this problem, I added a line to clear the tasks map completely before running the rest of delTaskTest(). By keeping an open mind to different solutions, I was able to fix this problem.

Discipline is an important value to have when testing software. It’s important to not cut corners when developing or testing code to prevent creating technical debt, where a simple solution leads to a longer term issue. I plan to avoid technical debt by carefully considering project specifications and requirements. To avoid cutting corners, I plan to consider different scenarios that these requirements might bring up or where the code could lead to errors through testing a diverse set of scenarios.

# **Works Cited**

Hambling, B., Morgan, P., Samaroo, A., & Thompson, G. (2019). *Software Testing.* Swindon: British Computer Society.