Ryan G. Cooper

Professor Wilson

CS-320 Project Two

August 13th, 2023

**Summary**

My code was fully aligned with the software requirements by checking the bounds and features of each requirement. For the task ID, contact ID, and the appointment ID the string length could not be longer than 10 characters, could not be null and could not be updatable. This had me check what was the full type for each one of these IDs. Since their requirements were the same, all three could be set as the same type. The requirement said the IDs had to be strings, but the other attributes were not so plainly stated. Since it says the IDs can not be updateable, I made sure the ID variables were initialized in the constructor and had the Final keyword along with the typing of string. This meant once the ID was made for each contact, task, and appointment, the ID could not be changed. The Final keyword is how I checked that the non-updatable requirement was meant for each class. I did this kind of thorough investigation for each requirement. After that, I would make the JUnit test for each class. After writing and running each JUnit class test, I would check the coverage of the tests within eclipse. I made sure that each one of my JUnit tests would cover 100 percent of the class I was testing before moving on. Since the coverage report would show which tests had not been tested with JUnit, I would make sure to go back and write a test for it. Some of the functions like Delete function for each class, I made two tests for it. One where it deleted a contact, task or appointment ID that existed (Lines 35 – 46, AppointmentServiceTest) and then another where it tried to delete one that did not exist (Lines 49 – 60, AppointmentServiceTest). The test that deletes an ID that exists is to make sure that the function works as expected. The test that deletes an ID that does not exist makes sure the error handler works and throws an error that the ID does not exist. Multiple tests like these are how I ensure my code is technically sound. To ensure that my code is efficient, I make sure to keep things simple. Each classes code was meant to be quick and easy, so the constructors for each class make the initial variables. Then getter and setter functions were made for the variables that need them. An example of this is the task class lines 25 through 43, getters for the ID, Name and Description variables. Then setters only for the Name and Description variables because the ID should not be updated as per the requirements. Then the constructors would check the other requirements like none of the variables can be null or over their character limit, if it was an error would be thrown to the user. This is seen in the task class too on lines 9 through 23. Other ways I made sure my code was efficient was making a hash map for each service class. This way, there could be a lot of contacts, tasks and appointments but a hash map will quickly retrieve them from the data structure.

**Reflection**

Some of the software testing techniques I used were unit testing, functional testing, and boundary testing. Most assessments centered around unit testing, utilizing JUnit tests for each function to ensure their proper functionality. These unit tests were crucial in verifying those functions operated as intended, and in cases where the software didn't meet expectations and failed a unit test, both the test case and the corresponding function were reviewed. Functional tests were important in aligning functions with the specified requirements, particularly evaluating the effectiveness of the add and delete functions within the service program for each class. These tests served as a check to confirm that the software met the requirements for the contact, task, and appointment classes. Lastly, boundary tests validated that input adhered to specified requirements. For instance, in the contact, task, and appointment classes, ID lengths were capped at 10 characters. Boundary tests made sure the IDs were within the limits to ensure they passed successfully and testing outside those bounds aimed to trigger an error, verifying proper error handling.

Several other testing methods, such as integration tests, regression tests, and performance tests, were not used in this project. Integration tests assess that functions work with one other as intended and prevent unexpected errors when combining components that operate well separately. Regression tests ensure that software updates don't compromise the functionality of existing functions, which is particularly important for live service software. These tests instill developers with confidence that changes introduced won't lead to new problems. Performance tests, although not applicable here, are essential for gauging an application's responsiveness under various conditions, critical for time-sensitive operations like websites, enhancing user experience.

Throughout this project, caution was used from the very beginning. This is because I feel the beginning of a project, the developer is building the foundation of the program. If the base is bad, the whole structure will fall apart. That is why I tried to keep things small and simple. From the start I tried to use standard coding practices like making getters and setters. Then make sure that I go and write a test for them. Doing this for each function made the process go by quick with little errors during the development of each class and its test. I tried to limit bias in the review of my code by first using the coverage percentage of the tests. This is the bare minimum of a way to remove bias since a numeric value can be applied to how well the program is tested. Then I tried to look at the boundaries for each requirement and make a test for them. Both inside and outside the boundary should have a test. An example of this is ID lengths for each class was tested this way as stated above. It would be easy to say that any program I make is good, but it is hard to take the time to remove the bias and critically review my program. It is also harder to answer that question without cutting corners and saying yes from the start. This is where I tried to keep my discipline and be sure that the quality for my code is professional. I have seen that cutting corners does not lead to writing or making good tests for programs. The most impact a program can have on people, the more important it is to avoid technical debt as a software engineer. I find it easier to build good habits from the start. Always treating every project with the utmost importance will always be the safest route. This also builds the habit of making good programs and tests so there will not be a worry that any corners were cut. This is a discipline that must be made by each software engineer, or it will come back to bite them.

**References**

Hambling, B., Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing an ISTQB-BCS certified tester foundation guide - 4th edition*. BCS, The Chartered Institute for IT.