**7-2 Project Two**

The testing approach used for each of the three features relied on using the software requirements as a guide to write JUnit test cases. This was done to ensure every requirement was met and verified according to the software requirements. These requirements, such as string character length, not null properties, as well as adding, updating, and deletion functionality, were all tested and verified through JUnit test cases. Each Java class in the AppointmentService, ContactService, and TaskService features host their own corresponding JUnit test case that serve demonstrating this testing approach. An example would be from the TaskTest.java test case for the TaskService feature where the sole purpose of the test case is to ensure that the software requirements are met. This was accomplished by first constructing a test that demonstrates the Task object can be created successfully by creating a Task object with a task ID, task name, and task description. Assertions were then used to test that the data received from the get methods matched the data used when creating the Task object to verify that it can be successfully created. Additionally, assertions were used to test that the constructor properly throws exceptions if strings were too long or null.

Text

Description automatically generated

When constructing the JUnit test cases for each feature, it was not only important to verify each requirement through successful test passes, but also to construct test cases for coverage. Having strong coverage would mean that the class being tested is done so thoroughly. Test points were written for each features test cases until 100% coverage was reached for every Java class. Doing so provides confidence that the features are implemented as intended without errors according to the software requirements.

To ensure that the code throughout the features were programmed in a technically sound and efficient manner, various methods were employed. These methods included ensuring that code is error free, consistent naming conventions, simplicity, and modularity. This includes not ignoring IDE syntax errors, following naming convention best practices, incorporating the least amount of code to meet software requirements, as well as programming with modularity in mind. An example of some of these qualities can be viewed from the Appointment.java class from the AppointmentServices feature.

Graphical user interface, text

Description automatically generated

Instead of having two separate if conditional statements to check if the appointmentID is null and uses more than 10 characters, it is consolidated into one if statement. Programming it this way shortens the amount of code which promotes readability and increases code efficiency. As far as writing technically sound code, this goes hand in hand with meeting software requirements while reaching high JUnit test coverage. As mentioned before, all features programmed have been tested to meet full functional requirements with 100% test coverage according to the software requirements.

The primary testing techniques used for each feature came in the form of static and dynamic testing. Static testing was implemented with static analysis which ran passively by the Eclipse IDE used to write the code for each feature. This allowed for quick assessment in finding errors or defects in code while writing it. As for dynamic testing, specification-based or black box techniques in the form of unit tests written for equivalence partitioning and boundary value analysis were used. Equivalence partitioning was utilized to test if objects throughout the features could correctly be constructed with their expected variable values, while boundary value analysis was used to test character length limits and not null characteristics according to software requirements.

Graphical user interface, text, application

Description automatically generated

Lastly, structure-based, or white box techniques were used throughout the features in the form of statement testing and decision testing for test coverage. Implementing white box testing techniques allowed for confident and thorough JUnit test coverage. An example would be from the TaskTest.java test case from the TaskService feature that tests the creation of the Task object.

Text

Description automatically generated

While many testing techniques were used and implemented throughout the features, there are also other techniques that were not used. Some examples include dynamic black-box techniques such as decision table testing, state transition testing, and use case testing. Additionally, some white box testing techniques like decision testing were not used in these features. While these testing techniques were not used in these features, every testing technique has its own best fit depending on the software project. This can be seen with decision table testing being best used for testing conditional logic in complex systems where different combinations of input conditions can trigger various unpredictable output. Additionally, state transition testing would be useful in testing systems where output is changed by the input conditions. Use case testing would be great for testing much larger systems that can gain from capturing the user-level interactions between mock-users and the system to find issues not present in individual component testing. As for white box techniques, decision testing would be an invaluable asset to systems that rely on complex loops to ensure that thorough testing coverage is met.

Acting as a software tester, it was important to exercise proper caution throughout every aspect of the testing approach among these features. It was known that, without proper and thorough caution when testing, it would be evident that flaws and errors would exist when running the program. This fact became clearer as each layer of code would add more complexity in the form of interconnected relationships among the features. These relationships, such as shared objects being used among different java classes, or the necessity for variable characteristics to be uniform to work properly are among many of the cautions that had to be kept in mind and tested for throughout the process.

Along with exercising caution, it was also important to limit bias when programming and testing. It can be very easy to overlook potential risks when testing programs developed by yourself because you lack the insight of outside critique. To limit potential bias, the knowledge that no one solution is the best was kept in mind when programming and testing. Direct feedback for each milestone was also welcomed and implemented when received for developing each feature.

Lastly, the grand takeaway is that the quality of a software project is the direct result of the discipline practiced when writing and testing code. It is inevitable that cutting corners may lead to a product that can cost financial losses or even lives. Just like we learned with the tremendous debt incurred by the F-35 fighter jet, or the public humility that Microsoft faced with their Tay artificial intelligence experiment, cutting corners is never worth the potential risks that may follow. I believe that it is important to always stress over the finer details when it comes to programming and testing. It is my goal to strive in always keeping an open mind of potential issues and ways the system can break to prevent these things from happening and incurring my own technical debt.