**Project Two - Summary and Reflections Report**

**Summary:**

While developing the contact, task, and appointment services for the mobile application I was able to align my testing to the software requirements by breaking each project into unit tests and testing to ensure that they met the specified requirements. For each service, I created two JUnit tests to validate the objects Class and the Service class separately. I then broke each documented requirement down and created a test case to ensure the program functioned as specified. For example, in the Contact class, this required testing the creation of a contact object and testing each requirement for the specific attributes. For the ContactService class I created tests to validate adding, deleting, and modifying contact information. The Task Service and Appointment service were tested in a similar manner to align with the software requirements.

The overall quality of my JUnit tests were sufficient considering the test cases covered one hundred percent of code of each service. Using JUnit tests proved to be effective because I was able to create tests, and after running the coverage report, I was able to see what sections of the code were not covered. At that point I could create new test cases that ensured the code base was covered one hundred percent. In addition to confirming I had complete coverage of code, I ensured all possible use cases for correctness by validating elements were added, removed, and retrieved correctly. This included testing edge cases such as trying to access non-existing elements and adding attributes that were null.

To ensure that my code was technically sound I developed test cases to cover all possible interactions with the classes. I first broke each requirement into individual tests and created tests to cover each possible interaction. For example, the requirements specified that unique objects should be able to be deleted. This was a straightforward test, but what happens if the user requests to delete an object that doesn’t exist? To validate this case, I wrote a test to cover this possibility and implemented the same test concept when modifying object attributes. Another example to illustrate that my code was technically sounds was for the requirement of an object to be created with specific requirements for their attributes. When a user wants to modify each attribute, the code then needs to be verified that the modification function meets the original requirements for the object’s attributes. This was implemented by creating separate test cases to test each modifier function to ensure it met the requirements.

One way I ensured my code was efficient was by validating the object ID’s were always unique. The only time an ID can be created is when an object is constructed. There is no method to modify the ID, therefore the ID will always stay unique and meets this requirement. This eliminated unnecessary code and testing to ensure the ID was always unique. I also wrote my tests in a way that were independent of each other so that the results of each test did not affect another test. This is apparent by the way I wrote each test independently to test a single requirement given specific inputs. I could have bundled a few of the tests together, but then the pass/fail criteria for one test may have been based on the results of another. This could lead to issues testing later revisions of the code during regression testing.

**Reflection:**

For each service, I was able to implement multiple testing techniques to validate the unit met all documented requirements. I first implemented static testing by reviewing the requirements to understand how each unit would be coded in a way that meets the requirements. After writing each package, I then reviewed the code looking for any gaps that could be found prior to executing any code. I then began to write Junit test cases to validate the code met all requirements. One technique I used while writing my unit tests was Black-Box testing. In this method of testing, I derived my test cases directly from the specifications documented in the requirements. Here I tested the units to ensure that the desired outputs were achieved when different input parameters were used. This included using typical and edge cases to ensure as many scenarios as possible were tested. Another testing technique that I used was white-box testing. With white box testing I concentrated on the test coverage to ensure that all branches of the code were executed at least once during the unit test.

One testing technique that I didn’t implement while testing each service was decision table testing. With this method of testing all allowable input conditions are listed with every action that can be produced by them. They are then structured into a table where each column represents a test case that identifies the inputs and expected output. This table can then be used to write each test during black-box testing because they are all derived from the specified requirements. I feel like this technique would have been quite useful to organize each test prior to being implemented in my test code.

When it comes to Black-Box testing, I think this technique is extremely important because the test cases are written with expected outputs directly from the specifications. This is a good method for validating software projects because the developer may have failed to implement certain requirements or interpreted the requirements slightly different during development. Black box testing can be used to validate the code meets the specified requirements no matter how they were achieved. White-box testing focuses more on the design of how the code was written and validates that all possible paths are taken during testing. This can be used to find bugs in the project and validate the structure of the program. By achieving a high degree of test coverage, it ensures that tests are comprehensive and cover all relevant branches.

When I first started writing code for the services in this project, my mindset was strictly in developer mode, and I was slowly testing the code by running a main function to output expected results to the terminal. As I began to implement JUnit tests I quickly discovered that running test cases were beneficial to quickly verify if the code would meet the specified requirements. I then expanded the test cases to encompass edge cases and different parameters to verify the code would still be valid. I was able to appreciate the complexity and interrelationships of the code when running the test coverage for each service. When doing this I could visually see what portions of the code I missed and implement a test to validate the missed lines of code. My tests stayed relatively simple to validate individual requirements based on different inputs, but the number of test cases grew quickly. This shows that while the code was relatively simple, to validate its functions could be quite complex when considering all use cases and inputs.

I found it relatively difficult to limit my bias when reviewing my code at first. This was because I was trying to validate that my code was functioning correctly to my expectations rather than to the specified requirements. As I implemented more test cases, my mindset quickly changed to ensuring my code was tested to meet the requirements. I imagine this bias can be a concern for testing your own code because it is easy to cut corners when you think your code is fully tested and covered. When code is tested by an independent tester it improves the quality by removing this bias. It should be the job of the tester to validate the code against any bugs or defects rather than testing to prove the code is written correctly to the developers expectations.

The importance of being disciplined in the commitment to quality as a software engineering professional is crucial to eliminate defects and future escapements in a released product. By ensuring comprehensive independent unit tests are written with quality in mind can be beneficial from a functionality and cost standpoint. When quality is developed into the work product it safeguards the software from technical debt in the field. This quality mindset should include working to coding standards when testing functionality, use cases, and performance of the code.