**Project Two**

**Summary**

For this project, I created unit tests in JUnit for the Contact, Task, and Appointment services. My approach relied heavily on boundary-value and exception tests. I tested the input fields, ensuring that values below and exactly at the maximum length were accepted, and that values over the maximum were rejected. I also checked that null values threw the proper exceptions. With the Appointment service, I confirmed that valid future dates were accepted while past dates were rejected. By keeping my tests closely aligned with the requirements, I was able to immediately identify if any requirement was not met. By naming each of the testing functions and errors descriptively, I was able to easily trace the failure back to the responsible method.

Each test directly reflected a stated requirement, in fact most requirements needed more than one test. For example, the requirement that contact IDs must be unique was supported by a test case that attempted to add a duplicate ID. The requirement that variables have maximum lengths was tested with explicit under, exact, and over values. The tests were done this way to ensure the requirements were satisfied exactly as written, and that coverage was not just overly broad.

The effectiveness of the tests was shown both in their coverage and their results. By the end of the project, the test files contained more code than the service files, which demonstrates the thoroughness of the testing process. Coverage reporting confirmed that nearly every branch and path in the code was getting tested, and any time a requirement was violated, the failure gave precise feedback that made it clear to debug.

Writing the tests was at times repetitive but became straightforward once a consistent structure was established. Having generic variables that exampled the expected input for quick copy/pasting shortened the time to write it all out, and made clear which specific variable was getting tested. Occasionally small typos caused tests to fail, but these issues were easy to track down since the failures pointed directly to the responsible line. The need for thoroughness required me to think critically when designing edge cases that I might not otherwise have considered.

I ensured that the code was technically sound by following established practices such as using private variables, implementing getters and setters, writing clear method names, and including comments for clarity. In other words, adhering to the standard coding practices that I have been learning in this and previous classes. In the tests, I validated even the test variables themselves to avoid human error. For instance, when checking the string length requirement, I verified the variable length before applying it with the assertion:

String thirty = "123456789012345678901234567890";

assertEquals(30, thirty.length());

By confirming the string length, I reduced the risk of a mistake in the test itself undermining the result. In a larger program where a greater number of variables would need to be unit tested this thorough, I would likely have those also be generic variables at the top and individually tested, so that they could be reused for input tests without reverifying each time.

**Reflection**

The techniques I used in this project included unit testing, boundary testing, and exception testing. These methods were effective in confirming that each service handled both valid inputs and invalid edge cases appropriately. Since each milestone focused on a single service, unit testing was the most practical and efficient choice.

There were, however, techniques that I did not use, such as integration testing, system testing, regression testing, and performance testing. These are more useful for larger projects where multiple components interact or where scalability and reliability under heavy loads must be considered. Static testing methods, like reviews or walkthroughs, also were not part of this project, but they can be very valuable for catching issues early before runtime.

The testing methods I employed are broadly useful in software development, particularly for catching input errors and ensuring that programs react correctly to invalid conditions. The techniques I did not use also have important implications. Regression testing, for instance, is critical to confirm that fixes remain effective over time, and system or integration testing ensures that separate modules work together properly. Performance testing becomes essential for real-world readiness: I have personally participated in several MMO beta tests meant to stress test their servers, yet even with that, server crashes are notoriously common on video game launches. Meanwhile static testing strengthens the foundation of a program by preventing defects before the code is even executed.

As someone with an above average amount of anxiety already, and also working in medical billing where errors could be cause for lawsuits or worse, caution is a trait that comes naturally. Perhaps in day-to-day life I overdo it, but in cases like programming and testing, I consider it a boon. For example, when working on the appointment service, I tested both valid and invalid dates to eliminate any ambiguity about the service’s behavior. This cautious approach helped make the code more resilient to unexpected inputs.

I do try to limit bias in testing my own work, but working these assignments did show me I do need to work on this more. Before running each of the tests, I had assumed that each of the tests would pass, because I was certain I had written everything out correctly. However, I learned in my current job that being able to prove I did the work correctly is very valuable for when something does go wrong, and a manager comes asking about it. Sure enough, not all the tests came back positive, showing that I had made errors that I was missing, usually a typo or missed syntax. There was even an instance where a test did pass but for the wrong reason, and I had only discovered it by looking at a different test that had failed. I refer to it as keeping my receipts, as in billing the best way to do this is save screenshots of system information that could be harder to obtain later, and this approach has helped both prove when I did something correct, or else discover an error before someone else does.

Finally, I approached the project with discipline by avoiding shortcuts. I validated test variables before applying them, used helper methods and generic inputs to eliminate redundancy, and made the tests even when I was confident the functions were correct. Had I cut corners, more than a few bugs would have slipped through just in these assignments. I believe this attention to detail would help avoid accumulating technical debt. I plan to avoid that debt by maintaining thorough, consistent test documentation and writing both positive and negative test cases.