**Summary**

**Software Requirements Alignment**

My testing approach was aligned to the software requirements because each test was based on one specific requirement. Before starting to write the test, I selected one specific requirement for the test to focus on. Then, I would make a test that would either confirm a correct entry or trigger an error to make sure it was working correctly. Finally, I tested each Junit test case to make sure it worked as intended. This would often involve creating an entry in the process, so I would ensure that the entry either met the requirements or purposefully entered information incorrectly. This allowed me to build test cases off of the software requirements, ensuring that each one would be met if the test passed successfully.

**Effectiveness**

I know that my JUnit tests were good quality and effective based on multiple factors. First, each test was able to be successfully run, meaning that the code itself was valid and running as intended. Second, there were some cases where I would test for a correct entry, and some tests where I would purposefully trigger an error, covering both aspects of the program’s requirements. Finally, I used the software requirements as a guide for how many tests to create to ensure the program would meet the expectations of the customer. For those reasons, I believe these JUnit tests to be good quality and effective.

**Technically Sound**

I ensured that my code was technically sound by frequently testing it to make sure it ran correctly, and paying close attention to error messages as they popped up. Many error messages could be solved easily with a semicolon to end the action, but others were more complicated. One such error code was when I forgot to reference the class the JUnit test was based on, so it did not properly call the variables. This occurred in the line testing if a name matched what was expected. It was solved by properly referencing the other class before executing the action. Continually checking that my code was running as expected was also key, and frequent checks made it easier to catch bugs that would otherwise throw error codes at the end.

**Efficient**

I ensured that my code was efficient by using if loops to check the data that was input against what was expected. The if loop would run by saying ‘if condition, return true’ or ‘if condition, confirm error message’. Once it checked the state of the condition, it would determine how to act based on the state itself. In this way, I could check both situations where things went right, and situations where things went wrong. I also tried to keep my code as concise as possible so as to be more readable and to not convolute the execution process. With these ways, I ensured that my code was efficient.

**Reflection**

**Software Testing Techniques**

The software testing techniques I used for each milestone were selected for their efficiency and thoroughness. JUnit testing was first selected because it would run quickly and return a simple answer of a green bar if the tests passed as expected or a red bar if it met some kind of error. Between only taking a few seconds to test the code and returning an easy to interpret result, it was the ideal way to test the code. I also utilized state transition as a method of software testing. To do this, I entered criteria that I knew would work within the code I wrote, but I also tried to enter items outside of the boundaries that had been set. For instance, I entered names that were too long, or left entries null. This allowed me to test the system behavior to make sure it would generate an error at appropriate times. Overall, it was useful to employ more than one kind of testing technique to ensure that I covered my bases and the program acted like I intended it to act.

**Techniques Not Used**

There were some software testing techniques that I could not use because they were not appropriate to the code I had written. One such technique was Equivalence Class Partitioning, where values can be split up into groups that are considered similar or the same for the purpose of testing. This was not appropriate testing for a phone contact list. Error guessing is another type of technique that was not selected, because it requires the tester to be knowledgeable in the type of error that is most likely to occur (“Software Testing Techniques”). Since my coding skills are not professional, I thought it might be beneficial to avoid this testing strategy for the time being. Overall, there are many types of software testing techniques that are available, but not all of them are appropriate for the type of application being written.

**Practical Uses and Implications**

Each software testing technique has multiple practical applications for different projects and situations. JUnit testing appears to be primarily universal, since the test code can be written and tested within the asme application and can be modified to perform many kinds of tests. State Transition can be utilized with password applications, such as when entering the wrong password three times will lock the account. It is also relevant when granting access after inputting the correct password (“Software Testing Techniques”). Equivalence Class Partitioning could be used when a certain set of values is desirable, but not others, like in a video game. There, a certain variable could have different outcomes in the story based on its value. Finally, Error Guessing is another universal type of testing, with many applications and situations where it could be applied and utilized to confirm errors and bugs within the code. Overall, each technique is important, even if it was not utilized in testing for this project.

**Caution**

In acting as a software tester, I was very quick to employ caution when testing and adjusting the code. The reason for this being that I did not want to create larger bugs in the code while trying to fix smaller ones, but changes did need to be made in order to have the code be functional. It was important to appreciate the complexity and interrelationships of the code I was testing because one wrong move could cause critical errors. For instance, I had a problem where I forgot to create a function for deleting an appointment, which prevented the code from running properly at all in the first place. While it made me nervous to create an entirely new function, it was also necessary to get rid of the error so the code could run. And once the function was completed, the error was resolved, and the code ran without error. While caution is good and needed, it is important not to let it become a hindrance at the same time.

**Bias**

I tried to limit bias during my review of the code by taking a day between when it was written and when I came back to review it. This allowed me to look at the code with fresh eyes and not immediately recall everything I had done in the code. On the software developer side, I cannot imagine that bias would be a concern if I was responsible for testing your own code. This is because I am willing to seek criticism from others as well as myself. I know I am a mediocre coder at best at this point in my career, and so by no means will my code be perfect at first attempt. For this reason, I like to think there will not be bias, and that the review will be fair and focused on making functional improvements.

**Discipline**

The importance of being disciplined in my commitment to quality as a software engineering professional is paramount. Quality is the foundational stone of a project, and can make or break the client’s satisfaction. It is important not to cut corners when it comes to writing or testing code because the errors will be found by the client or those running the software sooner or later. This could lead to a dissatisfied customer and a tarnished reputation. I plan to avoid technical debt as a practitioner in the field by being diligent and honest in my code writing and review. I will thoroughly use FIXME notes to identify places where my code is weak or may be completely nonfunctional. I will also seek out those with more experience than me to show me how to improve my coding skills and the project I am working on. Finally, I will ask for help when it is needed and own up to any mistakes. Overall. I believe that I have the discipline to be successful as a technical coder and efficient in my work.

Works Cited

*Software testing techniques with test case design examples*. Guru99. (n.d.).

https://www.guru99.com/software-testing-techniques.html.