**Summary and Reflections Report**

I feel that when designing and testing the Contact class that I was stumbling around and just trying my best to work with how I interpreted the requirements while also dealing with my lack of testing experience. This resulted in unique but effective testing methods. One example of this uniqueness was when I decided to include a ‘for loop’ in the test rather than just testing the individual vector index for the data. In the Task class I feel that I did better at meeting the requirements for not only the tests, but also the code. This can be seen by the fact that my code was overall more simplified and my test cases followed suit. For example, I did not include a for loop in the test cases. In both classes however I did include all of the data variables per the requirements specification.

Honestly I think there are several other areas I could have tested such as invalid input. I also did not test under the guise of the idea to test to break it but rather my tests just prove that it works according to spec. Because of this malpractice I can’t guarantee a professional level of accuracy, however, I would assess the coverage of the tests at 99% coverage. The reason is because of the meticulous level of detail that the tests go to. Although not perfect, they cover a lot of the deep and important areas.

In both projects my first tests tested the objects to ensure that data was handled and stored properly by the constructors. The remainder of the tests - TaskService lines 8 to 30 and ContactService 10 to 50 - included a test for every method to test that the same requirements were met.

One example is my use of the vector data structure. A vector is one of the most efficient data structures for this scenario. I made sure to include the vector at the beginning of the tests – line 4 of the TaskService and line 6 of the ContactService. I made sure to test that it worked as I had designed in every test of the TaskService and the ContactService. Because I called for a specific index they worked at constant time.

Writing JUnit tests helped me gain confidence in the correctness of my code. By crafting test cases that covered various scenarios and edge cases, I was able to validate the behavior of my methods and ensure they produced the expected results. It was reassuring to know that even after making changes or introducing new features, I could run the tests and quickly catch any regressions or unexpected issues.

I also found the arrangement of test methods within test classes to be highly intuitive, allowing me to easily comprehend the purpose and context of each test case. Moreover, the descriptive naming conventions for tests enhanced readability, making it simpler for me and I’m sure many other developers to understand the intention of each method.

One of the most significant advantages of JUnit was its seamless integration with my development environment. With just a few clicks, I could run my test suite and obtain immediate feedback on the success or failure of each test case. The clear and concise test output helped me pinpoint the specific areas where my code required refinement, enabling me to swiftly address issues.

Through this experience with JUnit, I also recognized the importance of writing testable code. The process of designing code that is modular, independent, and easily testable became clearer. It highlighted the significance of adhering to good coding practices which ultimately facilitates effective testing and increases overall code quality. An example could be such as when I determined that using objects was an effective method.

When it came to my overall testing method, I would say that I took a near complete white-box approach. This is inherently because I not only understood the entire design of the code, but the back-end of the program was my main focus when testing. Of course I also tested at the ‘black-box’ level as well because that proved that the system met the specified requirements. This combination of techniques was necessary because all of the requirements were based on the user experience. I also can’t deny that my previous personal experience had an influence on my testing methodology, although it had a smaller role to play. Of course it’s impossible to fully use every technique when testing a program.

Sticking to a few testing methods isn’t a bad thing as doing so minimizes a lot of confusion and perplexity to a test suite. But that does not negate the value that many methods provide to testing. One of these methods is equivalence partitioning which in short is partitioning of inputs, outputs or other values within the code. Essentially it’s a break down of the potential inputs/outputs which we can work with in order to thoroughly test. A couple other methods are decision table testing, and graphs and diagrams. These are considered white-box tests but they’re static methods that don’t work in this testing environment.

In conclusion, my journey with writing JUnit tests has been eye-opening. It has taught me the value of ensuring code correctness, providing structure and documentation, integrating seamlessly with the development process, and promoting the development of testable code. I am excited to continue harnessing the power of JUnit and unit testing in my future projects, confident that it will contribute to the robustness and reliability of my software.

**RESOURCES**

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. Accessed 6/15/23