**7-2 Project Two**

Christopher King

Southern New Hampshire University

CS-320 Software Test Automation& QA

Professor Kraya

December 10, 2023

**7-2 Project Two**

**Summary**

The software requirements were at the forefront of my testing approach. I started by breaking the assignment into smaller part. The assignment already had the requirements broken up into task class and task service requirements, I took it a step further and broke each into individual tasks such as “The task object shall have a required unique task ID String that cannot be longer than 10 characters. The task ID shall not be null and shall not be updatable.”. After I had each broken out, I wrote the required tests that would be needed next to them. This allowed me to make sure that I created all tests required.

To ensure the overall quality of my Junit tests was by utilizing the coverage percentages. For Task.java I managed to get 100% coverage pretty easily, but on the TaskService.java I struggled to get to 100% coverage. It took me a little while to come up with all the correct tests needed. I think this will become easier as I complete more Junit tests since I will be getting used to what is needed to be tested and how. I did better with coverage on this assignment than I did with the last assignment.

To ensure that my code was technically sound, I made sure to use proper naming conventions when creating variables, testing names, classes, and methods. I made sure to make my code clear, simple, and bug free. Below is an example of how I ensured my code was technically sound.

A screenshot of a computer program

Description automatically generated

One of the things I did to help make my code more efficient was to include a @BeforeEach setup so I could reuse code for each test while not having to re-type it out. This helps reduce the time required to create tests and also would require less code changes if the requirements were to change. For example, if the program would change to allow up to 25 characters in the name, we wouldn’t have to update a setup for each test, only the @BeforeEach. Below is an example of my setup.

**Reflection**

For all of the milestones that have been completed so far have utilized Unit testing, in particular we have been utilizing JUnit to perform these tests. For example, my AppointmentTest.java and AppointmentServiceTest.java contain all the test necessary to ensure that the Appointment and AppointmentService classes work properly. Utilizing unit testing allows for fast, isolated and repeatable testing and help to identify issues early on in the system build.

Some other software testing techniques that were not utilized in these milestones are Integration testing, System testing, and Acceptance testing. Integration testing is completed to verify the interactions between components of the software. This checks for issues such as compatibility issues, performance problems, incorrect communication, or data corruption. System testing is completed to verify the system as a whole, with all the components integrated. This checks to make sure the system as a whole are working properly together and checking for defects that result when the components are all working together. System testing is utilized to test the design and behavior of the system, along with the expectations of the customer. Acceptance testing is the last phase of software testing. Acceptance testing is used to find defects that were missed in previous stages, ensure the product is what the customer requires, and test how well the product is developed. The reason these testing techniques were not utilized is due to the project not being far enough along. At this point we are not testing the integration, or doing a system test, or completing an acceptance test. These are all completed at later stages of the process.

Throughout this project I tried to keep a focused mindset on what I was coding and trying to keep track of all possible avenues that would need tested. At times I would take the overly cautious path and create tests that ended up duplicating tests I have already completed. Keeping the complexity and interrelationships of the code kept me on my toes and I found myself re-tracing how certain items were connected together. Due to this fact, I had to ensure that I was testing all possible aspects of which I struggled with at times.

When reviewing my code and writing test cases I did not have much bias since I was second guessing my code on a regular basis. As I was completing JUnit tests, I found myself re-writing portions of my code somewhat regularly. This helped ensure I was not biased when reviewing the code because I knew there were errors in the initial code I created. I believe it would be difficult to limit bias for an experienced coder to review and write their own JUnit tests. The main reason I believe this is because an experienced coder may believe that their code is correct and not ensure that they are testing all aspects. It is the same thing as writing a paper, it is better to have someone else read the paper and critique it because you know what you are trying to get across so your mind may fill in the gaps when someone else’s mind may not.

When working on a product for a customer, quality is a high priority because if the customer is upset with the final product it can result in loss of creditability, hurt the company’s image, add time to the build to fix the issues, and add costs resulting in a lower profit. Being that I come from a quality background in manufacturing, I know the importance of getting things right prior to release. We have had product leave our facility that end up failing resulting in high field service cost, the customer being hesitant to order products from us again. This is compounded by the fact that our products start at $500,000.00, so they need to be correct.