Mauricio Bautista

CS-320 Software Testing, Automation & QA

Southern New Hampshire University

Prof. Wilson

December 8, 2021

**Final Project Two: Summary and Reflections Report**

**Unit Testing Approach**

The approach I took to the software requirements was close to what was expected by the customer’s idea. For example, in the contact service class, I coded and tested all the customer’s needs, like making sure the contact ID, first, and the last name did not exceed the required characters and that the phone option was exactly ten digits. In addition, I met the requirement of updating the first name, last name, number, and address by contact ID as required by the customer. I took a similar approach in the task service and appointment classes. For example, in the task service class, I ensured that the task ID did not exceed the ten characters and that the task name field did not exceed the 20 characters required. Also, I confirmed I could add, delete, and update a specific task by task ID.

Similarly, for the appointment class, I made sure that the appointment ID did not exceed the ten characters and that the appointment description did not exceed the 50 characters required. But most importantly, I took extra care making sure that the appointment date did not fall in the past, throwing an argument exception. Finally, I tested the add and delete methods by appointment ID, and both ways passed.

Based on coverage percentage, the Junit tests were effective. For example, if we look at the ContactTest class, there is coverage of 51.4%. However, If we look at the methods below, the coverage for all the requirements is 100%. Therefore, this description tells us that we met the customer’s needs.

Table

Description automatically generated

The rest of the classes had similar results. For example, the appointmentTest class had 78.4% of coverage, but all the methods tested at 100%. Once again, showing we met the requirements.

Table

Description automatically generated

**Experience Writing JUnit Tests**

I followed Java’s best practices to ensure that my code was technically sound. For example, I used proper naming conventions to be readable and understood by myself and other colleagues handling the project’s future maintenance. In the example below, I show how using private access modifiers for the strings keeps the fields hidden, preventing users from changing the data. In addition, I used properly commenting to provide additional information on what exactly the code is trying to achieve.

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

To ensure that my code was efficient, I ran coverage JUnit tests. The results show us in green what portion of the code was covered. For example, the screenshots below show in green that all the code was covered and tested successfully.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

**Testing Techniques**

Techniques employed in this project were black-box, white-box, and error guessing techniques. Black box testing mainly focuses on the input and output of software applications, and it is entirely based on software requirements and specifications. For example, black-box testing was used by examining all the requirements and specifications of each class. Then I chose valid (positive) and invalid (negative) inputs to check whether the system under testing processed them correctly. After determining the expected outputs for all the inputs, I fixed them and re-tested them to see if any defects were found in the code.

On the other hand, white-box testing ensures that each statement in the code of a component is executed at least once (Hambling et al., 2015). For example, white-box testing was used in the coverages tests to analyze all the features and statements in the classes. Finally, I used error guessing when I ran into trouble with the code. For example, I learned some techniques and mistakes in the previous miles stones. So, during the appointment class, I had issues figuring out how to implement the date(). I had to do a lot of guessing, use my intuition, and the little experience I had to test all the requirements. But, in the end, the tests were successful. Finally, a technique I did not implement was exploratory testing. Exploratory testing combines testers’ experience with a structured approach to testing where specifications are missing or inadequate (Hambling et al., 2015). Unfortunately, I couldn’t implement the technique due to the lack of other testers working with me.

**Mindset**

Acting as a software tester, I’m unsure if I employed caution. In other words, when I was performing all the tests, I looked for a way to try to make the test fail, intentionally trying different inputs. Then, I stopped the tests when I couldn’t figure out another way to make the test fail.

The complexity and relationship of the code I was testing are necessary because knowing the level of complexity of the code we’re trying makes it easier to understand the relationship between the code under testing. For example, the code written with illegal argument exceptions had a certain level of difficulty in our contact class. Still, it was necessary to use it under the contact test class to ensure I met all the requirements. All the classes had a similar approach. Complexity and interrelationships complement each other.

Bias is a concern when testing own code on the developer side. For example, it is difficult to find defects in something we have created as developers. For instance, I believe that my code implementation is perfect; however, if a real tester examined my work, I bet he would find many faults. Furthermore, the developer’s mindset is to make the system work instead of breaking it. Finally, staying disciplined as a software engineer professional is crucial. In other words, when we are developing applications for customers, we must stick to the implemented plan and test cases provided.

To conclude, we should avoid shortcuts or be lazy when coding. For example, I wrote some incomplete code to save time when working on these requirements, but I didn’t. In the end, I had to correct my error, which caused me more time. These strategies will help us avoid technical debt in the real world.

References

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition).* BCS The Chartered Institute for IT. Retrieved from  
<https://app.knovel.com/hotlink/toc/id:kpSTAIST01/software-testing-an-istqb/software-testing-an-istqb>