**Project Two: Summary and Reflections Report**

Rosario Robinson

Southern New Hampshire University

CS 320: Software Test Automation & QA

Instructor Joe Rangitsch

April 16, 2022

**Summary**

When approaching each of the three features required, Appointment Service, Contact Service, and Task Service, I focused on utilizing the tutorials and resources available. This way, I was able to stay aligned with the software requirements necessary for each of the three features required in project one. Starting from module three, I was provided resources such as the textbook, *Mastering Software Testing with JUnit 5*, which provided an understanding of how the testing life cycle works and how utilizing JUnit is beneficial during testing. Aside from using resources, I focused on following each weekly rubric and guidelines provided to ensure I was aligned with the requirements necessary.

The overall quality of my JUnit tests was efficient and taking coverage percentage in mind, it was 80% or higher for each feature required. The reason for this was aside from utilizing the tutorials provided each week, I also utilized outside sources (including reaching out to my instructor) while completing each feature. Considering that I began with the Contact Service application, I did have to understand how the functionalities of JUnit testing worked, so the coverage percentage may have been lower while testing the application. However, one section that gave me issues was the Appointment Service application, due to not being able to solve one of the issues with one of my tests. That feature scored at around 80% due to this.

While writing the JUnit tests for each application’s features, I was relatively able to understand and run the tests accurately. However, as I previously stated, I did run into issues with the Appointment Service application and having one section, which focused on the appointment date field. Aside from this section, the rest of my code for each feature was technically sound, especially when ensuring I properly generate unique IDs for each application. This did require additional research and resources; however, I was able to properly include the correct code to work through each application. For example, from my AppointmentService.java file:

**private String newCreateUUID() {**

**String createUUID = UUID.randomUUID().toString();**

**return createUUID;**

**}**

This code helped to generate a unique ID, while the code below added the unique ID to each appointment.

**public void addNewAppt(String appointmentID, Date appointmentDate, String apptDescription) {**

**Appointment appointment = new Appointment(newCreateUUID(), appointmentDate, apptDescription);**

**appointmentList.add(appointment);**

**}**

In addition to technically sound code, I ensured that my code was efficient throughout each application. For instance, ensuring that my code was not redundant, I focused on each test case’s efficiency to pass each JUnit test, as required via the rubric. Take the ContactTest.java for the Contact Service application, as shown below:

**@Test**

**void testContactTestIdTooLong() {**

**Assertions.assertThrows(IllegalArgumentException.class, () -> {**

**new Contact("8765432123456", "Jane", "Doe", "8004441800", "221 B Baker Street");**

**}); }**

Although this class was long and could be seen as redundant, for the sake of the testing process, I had to ensure that each function — contactID, firstName, lastName, phoneNum, and address — were all accounted for during testing to check for errors. Keeping the rubric and each test case in mind, this was the more efficient way to not only keep my code organized but ensure that I would pass each JUnit test.

**Reflection**

**Testing Techniques**

To understand the software techniques for JUnit tests, as I did during week five of this course, I utilized the resources provided. *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition)* was a great resource to understand how white box testing and black box testing could be utilized as I completed the applications. Overall, I utilized white box testing, since I created the code and tested it myself. This method is used often by developers, whereas black box testing is its opposite. With white box testing, this method requires utilizing structural testing techniques to ensure that each statement within my code ran via JUnit 5 on Eclipse (Hambling et al, 2015, p. 100). White box testing is a relatively straightforward process to utilize during my project, which also assisted me in staying organized and thorough during the testing process.

As I previously stated, I did not utilize the black box testing method as often within this project, since it focuses more on a lack of programming skills and knowledge and a sole focus on the behavioral aspects (GeeksforGeeks, 2020). Considering that I was also provided the rubric and guidelines for each application of this project, the black box testing method was not heavily required.

The practical uses and implications of both white box and black box testing are quite clear based on their methodologies once utilized in the project. White box testing focuses on the logical and programming skills required to implement and test the code, while black boxing focuses on the behavioral aspects with a lack of coding knowledge (or not a considerable amount). Practically speaking, utilizing white-box testing with this project was the better and more efficient choice as a developer, while having the requirements provided by the “customer”.

**Mindset**

My mindset while coding is usually a state of caution, as I want to ensure that not only do I understand what my code is doing, but also it is clear enough for when others must go through it. However, since in this project I am taking on the specific role of a software tester, it is even more important to proceed with caution while coding, especially since JUnit testing is involved. For example, while going through each test case required for each application, I had to ensure that I created not only the original code file that would have the functionality, but a test file that created tests for each file’s requirements. In my Task.java file, I had to ensure that my code met each requirement necessary to check for the length validity and if it appears as null:

**public Task(String taskID, String newName, String newDescription) {**

**if(taskID == null || taskID.length() > 10) {**

**throw new IllegalArgumentException("Invalid task ID!");**

**}**

**if (newName == null || newName.length() > 20) {**

**throw new IllegalArgumentException("Invalid name!");**

**}**

**if (newDescription == null || newDescription.length() > 50) {**

**throw new IllegalArgumentException("Invalid description!");**

**}**

By doing so, I was able to ensure that I am going through each line of code to recognize any errors or issues that may arise. It is always important to understand and appreciate the complexities and interrelationships of the code I am testing because this provides less room for additional stress and errors within my code and when I run the tests as necessary.

Limiting bias in my code is always difficult, as humans are never without bias in anything that they do. However, in this case, it is best to focus on situations that allow me to focus on the requirements only and complete them as necessary. In future situations, however, especially working with companies and their work, it is important to keep my own biases in check. In the case of testing my code, I believe software testers are a highly important part of the lifecycle process. They provide a different view of a person’s code so that they can identify any biases that may have presented themselves. This is an important aspect of development that should be dealt with as early on as possible.

Quality as a software engineer is highly important to me. No matter if issues or situations arise, providing the best quality code I can to a person or company is important to prevent additional complications in the future. Cutting corners only detriments everyone in the process and could lose them more money than necessary. Avoiding technical debt is a highly important aspect to avoid issues and irreversible damage occurring to a client or product by accelerating a process to meet a set deadline (ProductPlan, 2021). Utilizing methods such as Agile allows for QA to have enough time by taking the iterative approach to ensure consistent quality and staying on time within their calendar timeframe (Atlassian). Overall, as a developer in the field, it will take not only me to stay on top of my code, but ensuring I work well within a team to ensure all code is efficient and properly tested.

**References**

Robinson, R. (2022) *Module Four Journal: Unit Testing Approach and Writing JUnit Tests.* Computer Science Department, Southern New Hampshire University.

Robinson, R. (2022) *Module Five Journal: Software Testing Techniques.* Computer Science Department, Southern New Hampshire University.

GeeksforGeeks. (2020, August 5). *Differences between Black Box Testing vs White Box Testing.* <https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/>

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., Williams, P. (2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 2.6 Summary.* BCS The Chartered Institute for IT. Retrieved from <https://app.knovel.com/hotlink/pdf/id:kt00UC2I34/software-testing-an-istqb/life-cycle-summary>

*Technical Debt.* (2021, September 2). ProductPlan. <https://www.productplan.com/glossary/technical-debt/#:%7E:text=Technical%20debt%20>

Atlassian. (n.d.). *Escaping the black hole of technical debt.* <https://www.atlassian.com/agile/software-development/technical-debt>