Angel M. Rivera Moreira

SNHU CS320

Project 2

15 December 2024

**Summary and Reflections Report**

**Summary**

For the development of the contact, task, and appointment services, I adopted a systematic unit testing approach to ensure that each feature met the software requirements and functioned as expected. According to Hambling et al. (2019), systematic testing is critical for ensuring code correctness and robustness, which I applied when validating core functionalities of the **ContactService**. For example, I ensured edge cases were covered, such as adding a contact with a duplicate ID, which would throw an exception. Similarly, for the **TaskService**, I developed JUnit tests to confirm that tasks were being correctly created and updated, focusing on constraints around task descriptions and names. The **AppointmentService** was rigorously tested to validate the integrity of appointment creation, ensuring that invalid date entries or null IDs were handled appropriately.

My approach aligned well with the software requirements, as the tests covered all outlined functionalities and edge cases. For instance, in the **ContactServiceTest.java** file, I wrote tests to confirm that the service could not add contacts with duplicate IDs, ensuring the requirement for unique identification was enforced.

The overall quality of my JUnit tests is demonstrated by their effectiveness and code coverage. Jakubiak (2022) emphasizes the importance of high coverage percentages in JUnit tests, as they reflect thorough validation of all logical paths. By running coverage analysis, I confirmed that all public methods were tested, with additional focus on edge cases to ensure robustness. For example, in **TaskServiceTest.java**, I tested scenarios where the description exceeded the maximum character limit, validating the enforcement of input constraints.

My experience writing the JUnit tests was both challenging and rewarding. To ensure the code was technically sound, I adhered to test-driven development (TDD) principles, writing tests before implementing the corresponding methods. This approach aligns with best practices outlined in Hambling et al. (2019), ensuring that functionality is validated early in development. For example, the following test ensured the **updateTask** method rejected null inputs.

To achieve high test coverage, I intentionally created redundant test cases to validate multiple paths and edge conditions. While some tests overlapped, this approach ensured that no scenario was overlooked, contributing to a more robust validation of the code. The use of JUnit annotations like @BeforeEach to set up test environments also allowed for efficient and reusable test code (Jakubiak, 2022).

**Reflection**

**Testing Techniques**

For this project, I primarily employed **black-box testing** and **boundary testing** techniques. Black-box testing allowed me to verify the functionality of each service without delving into internal implementations, focusing purely on inputs and outputs. Hambling et al. (2019) describe this approach as ideal for functional validation. I applied in **AppointmentServiceTest.java** by providing invalid input data, such as null IDs or past dates, to ensure that the service responded appropriately with exceptions.

Boundary testing was beneficial for validating constraints. In the **TaskServiceTest.java** file, I tested the upper character limit for task names and descriptions, ensuring the service handled maximum and over-limit inputs correctly.

In practical applications, black-box testing is ideal for validating user-facing functionalities, while white-box testing is useful for unit testing critical algorithms or complex systems (Hambling et al., 2019). Mock testing is often necessary when working with external dependencies or APIs to isolate components.

**Mindset**

While working on this project, I adopted a cautious and methodical mindset. As a software tester, I recognized the importance of thoroughly validating the code to ensure it met requirements and handled edge cases. Hambling et al. (2019) highlight the significance of precision and attention to detail in software testing, which I applied when testing interrelated methods in **ContactService**. For example, adding, deleting, or updating a contact could affect the overall state, so I ensured these interactions were validated to maintain consistency.

To limit bias in reviewing my own code, I approached the tests with a critical eye, deliberately trying to break the functionality by inputting invalid data. For instance, I tested the **updateContact** method with null values, ensuring that it failed gracefully. I can imagine that bias would be a concern when testing my own code, as developers may unconsciously test only happy paths. Jakubiak (2022) suggests adopting an adversarial mindset to uncover weaknesses, which I applied by assuming the code would fail and validating edge cases.

Discipline in maintaining quality is critical for any software engineering professional. Cutting corners during testing can lead to undetected bugs, increased maintenance costs, and technical debt. For example, skipping edge-case testing for **AppointmentService** could result in appointments being scheduled with invalid dates, leading to potential system failures. Hambling et al. (2019) emphasize the long-term benefits of rigorous testing in preventing technical debt. As a practitioner, I plan to avoid technical debt by adhering to best practices such as test-driven development, code reviews, and continuous integration to ensure consistent quality.

This project allowed me to develop and validate back-end services using a disciplined testing approach. By leveraging effective testing techniques and maintaining a quality-driven mindset, I ensured that the services met the requirements and were resilient to edge cases. This experience reinforced the importance of testing as a core component of software development.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS Learning & Development Limited.

Jakubiak, N. (2022, December 6). JUnit tutorial: Setting up, writing, and running Java unit tests. *Parasoft*. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/