Merrik Wright

CS320 – Project 2

In the recent project where I developed contact, task, and appointment services, my unit testing approach was structured around ensuring that each feature met the specific requirements provided by the client. For each service, I implemented JUnit tests that focused on validating the core functionalities, such as adding, updating, and deleting objects, while also enforcing constraints like uniqueness and field length limits. For example, in the ContactService, I ensured that the contactId was unique and not updatable, and that fields like firstName and phone adhered to length and format restrictions. This approach was aligned with the software requirements, as I worked to incorporate each requirement into the test cases.

The effectiveness of my JUnit tests can be defended through the high coverage percentage achieved during testing, as we did the testing of each small portion, then the testing of the completed project. I used coverage tools like JaCoCo to ensure that at least 80% of the code was covered by the tests. This high level of coverage indicates that the tests exercised most of the code paths, reducing the likelihood of undetected bugs. Additionally, I implemented both positive and negative test cases to ensure robustness. For example, in the TaskService, I wrote tests to handle valid task creation as well as scenarios where the taskId was null or too long. These tests confirmed that the code correctly handled both expected and unexpected inputs.

Writing JUnit tests was a little bit of a process that required careful consideration of edge cases and potential weak points. I ensured that the code was technically sound by rigorously validating inputs before processing them. For instance, in the AppointmentTest class, I checked that the appointmentDate was not in the past, as required by the specifications. To maintain efficiency, I avoided redundant tests and focused on those that provided meaningful coverage. For example, instead of testing every possible invalid input individually, I grouped similar invalid inputs in a single test case, which streamlined the testing process without sacrificing thoroughness.

Throughout this project, I employed several software testing techniques, with unit testing being the most prominent. Unit testing is characterized by testing individual components in isolation from the rest of the system, which helps to identify issues at an early stage in the development process. I focused on testing the functionality and constraints of each service class independently, which allowed me to quickly pinpoint and fix bugs. This method of smaller testing was crucial to success. Additionally, I utilized boundary testing, particularly for fields with length constraints, to ensure that the code behaved correctly at the edges of acceptable input values.

There are other software testing techniques that I did not employ in this project, such as integration testing, system testing, and user acceptance testing. Integration testing involves combining individual units of code and testing them as a group to ensure that they work together correctly. This type of testing is crucial for catching errors that may not be apparent when units are tested in isolation. System testing, on the other hand, evaluates the entire system as a whole, verifying that it meets the specified requirements in a real-world environment. User acceptance testing (UAT) is conducted with the end-users or stakeholders to confirm that the system meets their needs and expectations. While these techniques are essential for larger projects or later stages of development, the scope of this project was focused on unit testing due to the relatively small scale and modular nature of the services.

The mindset I adopted during this project was one of caution and thoroughness. As a software tester, I approached the code with the understanding that even small oversights could lead to significant issues later on. It was important to appreciate the complexity and interrelationships of the code, especially when considering how changes in one part of the system could affect other areas. For example, when testing the AppointmentService, I considered how updates to an appointment's date could impact other services that might depend on accurate scheduling. This cautious approach helped me to identify potential issues that might not have been immediately obvious.

To limit bias in my review of the code, I made a conscious effort to view the code from the perspective of an end-user, rather than as the developer who wrote it. This involved questioning assumptions and testing scenarios that might seem unlikely but could still occur in real-world usage. Bias can be a concern when developers test their own code because they might unconsciously avoid testing edge cases that they believe are unlikely to fail. To combat this, I wrote tests that deliberately challenged the code's assumptions. For example, testing the TaskService with a variety of edge cases, such as extremely long task descriptions, would ensure that the code handled these situations well enough.

Finally, the importance of being disciplined in my commitment to quality as soon-to-be software engineering cannot be overstated. Cutting corners when writing or testing code can lead to technical debt, which can accumulate over time and become increasingly difficult to address. In this project, I avoided technical debt by adhering to best practices in code design and testing. For instance, I wrote clean, maintainable code with clear separation of concerns, which made the codebase easier to test and debug. Going forward, I plan to continue prioritizing quality by regularly refactoring code, writing comprehensive tests, and staying current with industry best practices. By maintaining this disciplined approach, I aim to produce reliable, high-quality software that meets both current and future needs. This class has been monumental in fortifying my abilities.