Module 7, Project Two

Faris Malik

SNHU CS-320

October 17th, 2024

**Unit Testing Approach for Task, Appointment, and Contact Services**

In Project One, my strategy was straightforward because each feature was tested in terms of its methods, specifically the TaskService, AppointmentService, and ContactService. My tests were based on constant checks such as the ability to create, modify and delete entries for each service. With respect to TaskService, I was able to check via unit tests that the ID, name and description of each task were always valid while for AppointmentService tDate and tTime for an appointment were validated as well. Moreover, ContactService tests included checks of phone number and address data validity.

**Alignment with Software Requirements**

The unit tests were closely aligned with software requirements to ensure all functionalities were properly implemented. For example, in the TaskServiceTest, I ensured that the service methods were tested thoroughly for updates to the task name and description:



By addressing edge cases, such as verifying that updates to the task's name and description were correctly implemented, I ensured that the unit tests aligned with design specifications and met the expected requirements.

**Defending the Quality of JUnit Tests**

The quality of my JUnit tests is reflected by the coverage percentage, which reached over 85% for the TaskServiceTest. High coverage percentages indicate that a broad range of functionalities was tested (Amasi & Walia, 2016). For instance, edge cases like null task names and over-limit descriptions were covered, ensuring robust testing across multiple conditions. Though getting 100% test coverage is not easy, the resulting percentage gives a lot of confidence to focus on the tests that are capable to detect any problem (Jammalamadaka & Parveen, 2019).

**Experience Writing JUnit Tests**

Writing JUnit Tests for this project required attention to both functionality and efficiency. In the AppointmentServiceTest, I focused on testing methods separately to ensure clarity and maintainability. For example, the test for adding appointments verifies that the service correctly adds a valid appointment: 

This approach made the tests concise and maintainable while still covering all necessary aspects of the service.

**Reflection**

**Testing Techniques Used**

The primary testing technique employed was unit testing, which isolated each method and ensured it worked as expected. This technique is most suitable for method level verification of functionality (Amasi & Walia, 2016). Also, black-box testing was applied, where all major points were considered without regard to the internal structure of the program. It also enabled me to confirm from a user standpoint the capability of each service.

**Testing Techniques Not Used**

Although unit testing was appropriate for this project, integration testing and system testing were used. Integration testing would have been useful for verifying how the different services interacted, while system testing could have ensured the entire application worked as a cohesive unit (Jammalamadaka & Parveen, 2019). These techniques are more suited to larger projects with multiple connected components.

**Practical Uses and Implications**

While unit testing is sufficient for small, isolated components, integration testing and system testing is important in large applications since different modules needs to work together as a single unit (Amasi & Walia, 2016). For instance, in instance of e-commerce system, integration tests would check that payment service communicates as required with the inventory system.

**Mindset**

**Caution and Complexity**

In working on this project, I took a cautious approach by considering edge cases such as null inputs and boundary values. This was important due to the complexity of managing IDs, dates, and other factors across the services. In the AppointmentService class, I ensured that appointments were validated correctly. The addAppointment() method verifies that the ID is unique, the appointment date is not null or in the past, and the description is valid:

public void addAppointment(String appointmentID, Date appointmentDate, String description) {

if (appointments.containsKey(appointmentID)) {

throw new IllegalArgumentException("Appointment ID already exists.");

}

if (appointmentDate == null || appointmentDate.before(new Date())) {

throw new IllegalArgumentException("Invalid Appointment Date.");

}

if (description == null || description.length() > 50) {

throw new IllegalArgumentException("Invalid Appointment Description.");

}

Appointment appointment = new Appointment(appointmentID, appointmentDate, description);

appointments.put(appointmentID, appointment);

}

By checking for null inputs and ensuring date validation, the service avoids incorrect appointments being added to the system.

**Limiting Bias**

As a way of avoiding bias, I went to the tests with a strong assumption that there were errors in the code. When I was thinking about my design on my own, there was less chance that I would fail to notice some shortcomings. For instance, when I was writing tests for the TaskService, although I thought the original logic was valid I tested invalid task IDs. This increases the risk of making biased assumptions while testing one’s own code, but I tried my best to examine the test from the perspective a third-party tester.

**Commitment to Quality and Avoiding Technical Debt**

Maintaining a strong commitment to quality is essential in software development to avoid accumulating technical debt (Jammalamadaka & Parveen, 2019). Cutting corners, particularly in testing, can result in costly rework down the line. As a software engineering professional, I plan to avoid technical debt by continuing to prioritize thorough testing and quality assurance in future projects. During working on this project, I learned not to rush with some types of tests, as it will be more important to do them thoroughly. As such in the future, I will need to be just as diligent to make sure that my code stays both correct and easily understandable.

**Conclusion**

This unit testing approach worked well in confirming the functionality of the various services that include the TaskService, AppointmentService, and the ContactService in fulfilling the project’s requirements. The tests it offered were quite effective giving high coverage percent and less chances of having errors unfound. Applying an organized thinking approach of operating within a functional and efficient manner, I provided dependable services with minimal technical debts. In future projects, I will continue lot of testing strategies to ensure the code quality and easier to maintain in all the projects.

**References:**

Amasi, N. M., & Walia, G. (2016). A comparative analysis of software testing methodologies. In *Software Testing, Verification & Reliability*. <https://link.springer.com/chapter/10.1007/978-3-319-59647-1_27>

Jammalamadaka, K., & Parveen, N. (2019). A comprehensive examination of software testing methodologies and their associated challenges. *International Journal of Innovative Technology and Exploring Engineering*. <https://doi.org/10.35940/ijitee.F1307.0486S419>