Project Two: Summary & Reflections Report

Klawonn

For the appointment feature, the unit testing approach for the ‘Appointment’ and ‘AppointmentService’ classes focused on ensuring that all critical aspects of the appointment creation as well as the management process were covered. For ‘Appointment’, tests were created in order to validate that IDs were neither null nor exceeded the length limit, dates were not null or set in the past, and descriptions remained within the length limit. For ‘AppointmentService’, tests were added to ensure that appointments could be added, duplicate IDs were not allowed, and appointments were able to be deleted.

For the task feature, the unit testing approach for the ‘Task’ and ‘TaskService’ classes was similar. The ‘TaskTest’ validated task ID, name and description to ensure they were not null and did not exceed the given character limits. The ‘TaskServiceTest’ covered adding, updating, and ensuring invalid data (like too long IDs or null names) were able to be appropriately handled.

The ‘Contact’ and ‘ContactService’ tests ensured that IDs, names, phone numbers, and addresses met the specified criteria. The service layer tests included adding and deleting contacts, ensuring duplicates were not allowed, and updating contact fields. I was also able to ensure that the phone number was not to be null and had exactly ten digits.

My approach was highly aligned with software requirements, as evidenced by the rigorous validation checks involved. For example, the requirement that appoint IDs should not exceed ten characters was strictly enforced, as seen in ‘AppointmentTest.testAppointmentIdTooLong()’, where a test for an ID longer than 10 characters threw an ‘IllegalArgumentException’. Similar alignment was maintained in the ‘Task’ and ‘Contact’ tests, where input validations were matched with the requirements.

The JUnit tests are of high quality because they comprehensively cover both positive scenarios, where valid data is processed correctly, and negative scenarios, where invalid data is correctly rejected. For instance, in the ‘ContactServiceTest’, tests like ‘testAddDuplicateContact()’ ensure that the service behaves correctly under edge cases, which contributes to effective test coverage. If I was able to use coverage tools, it would likely show near-complete coverage, particularly in critical methods like constructors and data validation methods, demonstrating that the tests effectively cover all essential code paths.

Writing the JUnit tests was an iterative process that involved anticipating potential issues and validating assumptions about the codes behavior. Initially, I wrote simple tests to verify basic functionality, but as edge cases were considered, additional tests were added to handle scenarios like null inputs and maximum length violations. The process was challenging but also rewarding, as it required me to really dive for a deeper understanding of different parts of the system and how it interacts with other moving parts.

Technical soundness was ensured by writing tests that cover all edge cases. For example, in ‘TaskTest’, the line ‘assertThrows(IllegalArgumentException.class, () -> { new Task(null, “Task Name”, “Task Description”); });’ ensures that the code properly handles null IDs, which is a crucial validation step. Additionally, assertions like ‘assertEquals(“New Task Description”, task.getDescription());’ in the ‘testTaskDescriptionUpdate()’ method ensure that updates work as expected, confirming the code’s correctness.

Efficiency was ensured by minimizing redundancy in tests and ensuring that each test case was isolated and focused. For example, the use of ‘assertTrue’ and ‘assertFalse’ in the ‘ContactServiceTest’ allowed for quick verification of the services behavior without unnecessary setup or teardown processes. By focusing on the essential aspects of each feature, like length and null checks, I was able to keep the tests concise and focused, which is crucial for maintaining efficiency.

There were several techniques I used for testing. I employed unit testing extensively, with a focus on testing individual classes and methods in isolation. For instance, the ‘TaskTest’ class tests the ‘Task’ class independently of other classes, this allows me to pinpoint specific issues in task creation and validation. I also used boundary testing. Many tests focused on boundary conditions, such as maximum length for IDs or descriptions. This ensured that the system behaved correctly at the edges of input constraints. I also wrote tests to handle invalid inputs and ensure that the system responded with appropriate exceptions, such as in ‘testTaskInvalidId()’ and ‘testTaskInvalidDescription()’.

There were also several techniques I did not use in my testing. I did not employ integration testing. In this project I focused more on unit tests. Integration testing would involve checking how different components of the application work together, such as how the ‘AppointmentService’ would interact with other services. I did not use performance testing, as I found the focus to be more on functionality rather than speed or memory usage. Performance testing would assess how the system behaves under load or how quickly it would process data.

Unit tests are vital in catching bugs early in the development process. They ensure that individual components work as expected, which is crucial for projects that rely on small interacting parts. For example, ensuring the integrity of input validation in the ‘Appointment’ class is critical before it is integrated with other parts of the system. Boundary and negative testing are essential for robust software development, as they ensure that the system can handle unexpected input. This is particularly important in user-facing applications where invalid input is more common. While integration testing was not used, it is crucial in larger systems to ensure that different modules work together effortlessly. For example, in a project where the ‘AppointmentService’ would interact with a notification service, integration tests would be necessary to ensure that appointments trigger notifications properly.

Adopting a cautious mindset was crucial in writing tests for this project. Being able to understand the interrelationships of the code was extremely important, especially when dealing with shared resources like the ‘appointments’ map in ‘AppointmentService’. I talked about the process being iterative, and that really applies when adopting a cautious mindset. It’s important to spend time looking through your code to develop tests, and then even going back and forth between the two, ensuring that all possible boxes are checked. Appreciating a cautious mindset also involved considering edge cases, such as what happens if an appointment ID is null or if the date is set in the past. Without this mindset, critical bugs could be missed.

Limiting bias involved approaching the code from the perspective of the end-user and not assuming that the code would always behave as expected. For example, in ‘ContactTest’, ensuring that null values for the first or last name threw an exception required not assuming that these values would always be provided. I feel being responsible for testing ones own code, could lead to bias and potentially overlooking issues. Things might be overlooked or not thoroughly tested just because you want to believe you did things the right way. It also comes down to the fact that you have been potentially working on this for hundreds of hours. Having a fresh set of eyes, can allow problems to be found that you may have never even noticed. I really find it important that another person always be able to check over your code, especially when that code has the potential to affect others experiences and lives.

Being disciplined in testing is crucial and essential to avoid technical debt. Cutting corners could save time initially but could lead to detrimental issues. For example, skipping tests for null values in the ‘Appointment’ class might lead to crashes in production, which could have been prevented. Avoiding technical debt involves writing comprehensive tests now to avoid having to rewrite or debug complex issues later. This project emphasized the importance of thorough unit testing as a means of maintaining long-term quality code. I also just want to note the responsibility developers have to involve ethics at the forefront of coding and testing. Programs have the potential to impact many lives, and as such should be tested to the best of their ability, to prevent catastrophic consequences.