Reginald Cooper CS320

Instructor Farley 10/17/024

Summary and Reflections Report

Grand Strand Systems

October 17, 2024

Summary

As part of the mobile application project for Grand Strand Systems, I developed and tested three key features: the contact service, task service, and appointment service. To ensure the robustness and functionality of these features, I employed unit testing using JUnit. The following summarizes my approach to testing each feature and reflects on the effectiveness of the overall testing strategy.

Unit Testing Approach

1. Contact Service:

For the contact service, I wrote unit tests to cover the essential operations such as adding, updating, and deleting contacts. These tests made sure that all business rules, such as making sure contact details are not left blank or that contact IDs remain unique, were verified through assertions. I followed a test-driven development approach where the tests were created alongside the features, making sure that each functionality was properly validated as it was developed.

2. Task Service:

For the task service, I focused on testing the lifecycle of tasks—creation, modification, and deletion. Unit tests were written to make sure that tasks stick to predefined rules such as character limits on descriptions, proper task status, and date validation. A significant portion of the tests focused on boundary cases, like what happens when invalid inputs are provided, making sure the service could handle unexpected user actions gracefully.

3. Appointment Service:

The appointment service required thorough testing, especially around date and time validation. Unit tests were designed to check for overlapping appointments, invalid date ranges, and the ability to modify appointments without causing conflicts. Since the functionality involved handling time-sensitive data, the tests ensured that the appointment service behaved correctly across different time zones and handled edge cases like leap years or incorrect time formats.

Alignment with Software Requirements

My testing approach was closely aligned with the software requirements for each feature. For instance, in the contact service, one requirement was to ensure that a contact’s first name could not be null or empty. My unit tests specifically verified this requirement by asserting that attempts to create a contact with missing information would result in an error, as shown in `assertThrows(IllegalArgumentException.class, () -> new Contact("", "Doe", "12345", "1234567890"))`. Similarly, in the task service, one requirement was to enforce that task descriptions must not exceed a certain character limit, and this was tested thoroughly in various scenarios using assertions like `assertTrue(task.getDescription().length() <= 50)` to ensure compliance with the length limit.

Defending the Quality of JUnit Tests

The overall quality of my JUnit tests can be defended by the fact that all tests exceeded the 80% coverage threshold. Achieving this high level of coverage ensured that the critical functionalities of the contact, task, and appointment services were thoroughly validated. The tests not only covered happy paths but also edge cases and potential failures, providing confidence that the services will function as expected under various conditions. Code coverage reports showed that essential parts of the code were tested, and no significant gaps were left uncovered.

Experience Writing JUnit Tests

How I Ensured Technically Sound Code:

I made sure that my code was technically sound by sticking to best practices in writing assertions that thoroughly validated the functionality of the methods under test. For example, in the `ContactServiceTest.java` file, I included a test to ensure that a contact’s phone number must follow a specific format:

assertEquals("1234567890", contact.getPhoneNumber());

This ensured that the phone number was set correctly and validated as part of the contact creation process. Also, I tested for invalid inputs using:

assertThrows(IllegalArgumentException.class, () -> contactService.addContact(null));

This line ensures that the `addContact()` method cannot accept null inputs, making the code technically sound by preventing potential runtime errors.

How I made sure of Code Efficiency:

To make sure efficiency, I used techniques like parameterized testing in JUnit 4 for the task and appointment services, which allowed me to test multiple input scenarios without duplicating code. For instance, in `TaskServiceTest.java`, I used parameterized tests to validate task descriptions:

@ParameterizedTest

@ValueSource(strings = {"Task 1", "Task 2", "Task 3"})

void testValidTaskDescriptions(String description) {

Task task = new Task(description);

assertNotNull(task.getDescription());

}

This method allowed me to reuse the same test logic for different task descriptions, reducing redundancy and ensuring that the tests were efficient and maintainable. Additionally, for the appointment service, I reused setup methods to create common test data, avoiding code duplication across multiple test cases, as shown below:

@BeforeEach

void setUp() {

appointmentService = new AppointmentService();

appointment = new Appointment("1", "Doctor's Appointment", "2024-10-17", "14:00");

}

This setup method initialized the appointment service and appointment objects that were reused across multiple test cases, improving the efficiency of the code.

Reflection

Testing Techniques Employed

The primary technique I employed was unit testing, which focused on testing the smallest units of code (such as methods or classes) in isolation. This technique allowed me to ensure that each feature behaved as expected, independent of other features or services. Unit testing is characterized by its focus on testing single functionalities in isolation, which reduces the complexity and allows pinpointing specific issues when they arise.

Other Testing Techniques Not Used

There are several other testing techniques that I did not employ in this project but could be useful in different contexts. Integration testing is one such technique, which involves testing the interaction between multiple components or systems. This is particularly useful when you want to make sure that different modules work together seamlessly. System testing, which tests the entire application in an end-to-end manner, was not employed in this project because my focus was strictly on unit testing the back-end services. This technique, however, is crucial for validating the complete functionality of the system.

For example, in larger-scale projects where multiple developers are working on different components, integration testing would be crucial to ensure that the services developed by each team member work correctly together. System testing, on the other hand, would ensure that the final application meets all the functional and non-functional requirements.

Mindset

As I worked through this project, I adopted a mindset of caution and thoroughness, especially given the complexity of the features I was testing. Testing services like appointments, where multiple factors such as date and time must be handled accurately, required me to carefully consider how changes in one part of the service could impact other areas. For example, adding validation for time zones could affect the ability to schedule appointments accurately across different locations. This mindset of caution helped me to appreciate the interconnectedness of the system, and I ensured that small modifications were thoroughly tested.

I also worked to limit bias during testing by regularly reviewing my test cases to ensure that I was not overly confident about areas of code that I had written myself. Bias could easily creep in when you are testing your own work, leading you to overlook potential issues. To mitigate this, I used peer feedback and code reviews to ensure objectivity.

Commitment to Quality

Being disciplined in my commitment to quality was an essential part of my role as a software engineer during this project. It is easy to cut corners when writing or testing code, especially when deadlines are tight, but doing so can lead to technical debt. I made sure that I avoided technical debt by writing clean, maintainable code and implementing thorough test coverage from the start. By adhering to these practices, I not only made sure the short-term functionality of the services but also made future maintenance and scalability easier. In future projects, I plan to continue following best practices, including regular code reviews, thorough testing, and ongoing refactoring, to avoid accruing technical debt.