Logan Meyer

10/18/2025

CS 320

Project Two: Summary and Reflections Report

For this project, I focused on unit testing the three core services of my mobile application. These three core services are ContactService, TaskService, and AppointmentService. Each service handled its own area of functionality, so I created separate JUnit 5 test classes to verify that all methods worked correctly. My main goal was to confirm that each service could add, update, delete, and validate data according to the software requirements. I ran all tests in Eclipse using the built-in coverage tool. All forty-two tests passed successfully, and the overall coverage across the project averaged about eighty-three percent.

My unit testing approach focused on small, isolated pieces of logic so that I could verify each service without depending on the others. For example, the ContactService tests made sure that a new contact could be created only with a valid ID, name, and phone number. The TaskService tests checked that a task could not exceed the maximum allowed description length and that each task had a unique identifier. The AppointmentService tests confirmed that appointments were scheduled only for valid dates. Each test followed the same structure of setting up data, running the method, and using assertions to confirm the result.

My testing approach aligned closely with the project requirements because each requirement was translated into one or more test cases. For instance, one ContactService test tried to create a contact with an ID longer than ten characters. The test passed only if the service threw an exception or rejected the input. This made it easy to verify that the program behaved as expected.

The quality of my JUnit tests can be defended by the level of coverage and the range of test conditions. The coverage report showed values between eighty-one and eighty-five percent for all services, which means that almost every branch of the program was tested. Because all tests passed, I am confident that the JUnit tests were effective and reliable.

I used a mix of assertions to check different conditions, such as “assertEquals”, “assertTrue”, and “assertThrows”. These assertions are a standard part of the JUnit framework and confirm that expected results match actual outcomes (JUnit Team, 2024). For example, in TaskServiceTest, one test used “assertTrue(taskService.updateTask("T001", "Updated Task", "Updated description"))” to confirm that updates worked as intended. These tests ensured that both valid and invalid paths were handled correctly. I kept each test short, independent, and easy to run in any order, which made the suite efficient and simple to maintain.

The main testing techniques I used were unit testing, boundary testing, and equivalence partitioning. Unit testing allowed me to focus on small sections of code, such as a single method in a class. Boundary testing helped me check input limits, like the exact character length allowed for contact IDs. Equivalence partitioning divided input data into valid and invalid categories to cover more cases without unnecessary repetition. These techniques are widely used to ensure full coverage with a manageable number of tests (GeeksforGeeks, 2024). Together, they confirmed that the program worked for both normal and edge cases.

Other software testing techniques I did not use include integration testing, system testing, and acceptance testing. Integration testing checks how different parts of the program work together. System testing validates the program as a whole, and acceptance testing confirms that it meets client expectations. These methods are more useful in larger projects, while my focus here was on unit-level testing.

This project also helped me develop the right mindset for testing. I learned that testing requires patience and careful attention to detail. Even small changes can affect how other classes behave, so I tried to think ahead and test carefully. Since I was both the developer and the tester, I worked to reduce bias by testing invalid and unexpected inputs. Thinking like a user helped me stay objective and identify issues I might have overlooked.

Being disciplined with testing is an important part of software development. It can be tempting to skip tests when everything seems to work, but that often causes problems later. Writing complete tests for every method helped me prevent bugs and made future changes easier to verify. I plan to keep this same approach in future projects to ensure that my code remains reliable and easy to maintain.

References

GeeksforGeeks. (2024, April 10). *Boundary value analysis vs equivalence partitioning - software testing.* GeeksforGeeks. https://www.geeksforgeeks.org/software-testing/software-testing-boundary-value-analysis-vs-equivalence-partitioning/

JUnit Team. (2024). *JUnit 5 user guide.* JUnit.org. https://junit.org/junit5/docs/current/user-guide/