Project 2 Summary and Reflection Report

For this project, I employed a structured unit testing approach for the TaskService, AppointmentService, and ContactService features. The strategy included creating JUnit test cases that validated the core functionality of each service while ensuring compliance with the provided requirements. Each feature was tested to confirm that the data structures correctly handled adding, updating, and deleting objects while enforcing constraints on data validity. For the TaskService, I ensured that task objects adhered to length restrictions and non-null constraints for taskId, name, and description. The JUnit tests verified that invalid values resulted in exceptions. Similarly, the AppointmentService required validation of appointmentId, appointmentDate (future date enforcement), and description length constraints. The ContactService required enforcing constraints on contact fields such as firstName, lastName, phoneNumber, and address, where each field had specific character length restrictions.

My approach closely aligned with the software requirements by incorporating boundary value testing and negative testing to ensure proper validation. The JUnit tests simulated real-world scenarios, such as adding a task with a duplicate ID, which resulted in an exception (IllegalArgumentException). Attempting to set an appointment date in the past was properly rejected, and updating contact details with values exceeding length limits resulted in test failures. These tests confirmed that the data integrity and business logic were correctly implemented according to the provided requirements.

The quality of my JUnit tests was ensured by achieving high test coverage across all key functionalities. Every method in the service was exercised through tests, verifying both expected and unexpected inputs. A high coverage percentage (above 90%) was maintained by incorporating equivalence partitioning and edge case testing to evaluate the constraints effectively.

Writing JUnit tests provided valuable insights into test-driven development (TDD). I initially encountered challenges in designing edge cases but overcame them by leveraging techniques like assertThrows() for exception handling and parameterized testing for multiple input scenarios. To ensure technical correctness, I tested not only valid inputs but also extreme cases such as very long descriptions exceeding 50 characters, null values for mandatory fields, and attempting to delete a non-existent task or appointment.

I employed unit testing, boundary value testing, and negative testing as my primary software testing techniques. Unit testing focused on verifying individual components, boundary value testing ensured proper validation of input constraints, and negative testing intentionally provided invalid values to verify exception handling.

While I focused on unit testing, I did not use integration testing, performance testing, or system testing. Integration testing validates interactions between components, performance testing measures efficiency under load, and system testing ensures that the entire system works as expected. These techniques were unnecessary since the project relied on in-memory data structures rather than external dependencies.

I adopted a cautious mindset, ensuring that each test case accounted for potential failure points. Understanding the interrelationships between data constraints and application behavior was crucial. By thoroughly testing valid, edge-case, and invalid inputs, I ensured that the application handled real-world conditions effectively.

To reduce bias, I applied objective validation criteria, focusing on requirements rather than assumptions. If I were testing my own code without independent validation, confirmation bias could lead me to assume correctness without rigorous testing. This is why I structured tests to intentionally break the system—validating robustness by testing invalid conditions.

A disciplined approach to testing is essential in software engineering to avoid long-term maintenance issues (technical debt). Cutting corners in testing could lead to bugs slipping into production, leading to costly post-release fixes. To avoid technical debt, I plan to follow Test-Driven Development (TDD), use automated test suites, and document test cases clearly. By maintaining a commitment to thorough testing, I can deliver high-quality, maintainable software while ensuring a robust user experience.

In conclusion, this project reinforced the importance of structured testing methodologies, rigorous validation of constraints, and disciplined software testing practices. By employing unit testing, boundary testing, and negative testing, I ensured that the TaskService, AppointmentService, and ContactService met all specified requirements. Writing effective JUnit tests required careful planning, attention to edge cases, and a strong commitment to code quality. Moving forward, I will continue applying these principles to ensure that the software I develop is robust, reliable, and scalable.