Project Two

**Summary**

**Unit Testing Approach by Feature**

In the development of the Contact, Task, and Appointment services for our client’s mobile application, I implemented a comprehensive unit testing strategy to validate individual components and ensure adherence to business requirements.

For the ContactService, unit testing focused on input validation for contact fields such as IDs, names, and phone numbers. The tests were crafted to enforce field constraints, character limits, and handle improper input gracefully. In the TaskService, testing targeted the enforcement of business rules regarding description length and title requirements. The AppointmentService tests centered on date handling logic and validation to prevent past or invalid appointment entries.

**Alignment to Requirements**

Each of the tests was written with reference to the software requirement specifications. For example, the constraint that contact IDs must be no longer than 10 characters was rigorously tested. These validations ensured that all acceptance criteria were met prior to merging code into the main repository.

**JUnit Test Quality**

The test coverage results from Eclipse’s integrated code coverage tools showed that over 95% of code paths were covered. This high level of coverage provided confidence that both typical and edge cases were addressed. Tests were organized to isolate each functionality, providing thorough validation of each feature’s behavior in expected and exceptional conditions.

**Writing Experience and Technical Soundness**

The testing experience was smooth and iterative. I structured the tests using JUnit 5 best practices including the use of @BeforeEach for consistent test initialization and descriptive assertion statements to improve readability and maintainability. The use of assertions helped ensure technical soundness by verifying each state transition or returned value.

**Efficient Code Practices**

Test cases were structured to reduce redundancy and maintain clarity. By isolating inputs and reusing test objects across multiple test cases, I ensured that each unit of functionality could be evaluated quickly and clearly. Modularization of test data setup improved efficiency during test execution and future scalability.

**Reflection**

**Techniques Employed**

I employed black-box testing, white-box testing, and unit testing methodologies. Black-box testing allowed me to verify output correctness without concerning myself with internal code logic. White-box testing ensured all conditional branches and logic paths were exercised. Unit testing validated the behavior of individual methods and classes.

**Techniques Not Used**

While robust, this project did not include integration testing, system testing, or exploratory testing. Integration testing would have verified the interaction between services. System testing, valuable before deployment, checks the application as a whole. Exploratory testing, typically performed manually, allows creative discovery of bugs through unscripted use.

**Practical Implications**

Unit testing is crucial during early development phases. Integration testing becomes essential as multiple services interact. System testing helps ensure the application works end-to-end. Exploratory testing can uncover unforeseen issues in real-world use, making it an important layer during beta testing or user acceptance testing.

**Testing Mindset**

I adopted a cautious, detail-oriented mindset. Being thorough with both valid and invalid inputs helped ensure robustness. Understanding how different components related to one another helped identify subtle bugs, particularly when date manipulation or character limits were involved.

**Limiting Bias**

To minimize confirmation bias, I actively tested for invalid, null, and boundary inputs. Taking a step back to test from a user’s perspective revealed assumptions in the code logic. Bias in testing one’s own code is a known concern, so I made an effort to write tests that challenged my assumptions and simulate unexpected behavior.

**Commitment to Quality**

Maintaining discipline and quality is a non-negotiable principle in software engineering. Cutting corners can result in costly bugs and eroded trust with clients. To prevent technical debt, I commit to ongoing test automation, regular refactoring, and clear documentation. These practices not only support code quality but also future-proof the application.

**References**

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice* (3rd ed.). Addison-Wesley.

JUnit Team. (n.d.). *JUnit 5 user guide*. <https://junit.org/junit5/>

Martin, R. C. (2008). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

Oracle. (n.d.). *Java documentation*. <https://docs.oracle.com/en/java/>