Jose Alonso

Summary

The unit testing approach for the three key features of contact, task, and appointment services was carefully designed to align closely with the specific software requirements for each feature. For the contact service, the tests concentrated on verifying constraints such as unique, immutable contact IDs, maximum string lengths for contact details, and proper handling of updates and deletions. For example, tests confirmed that adding a contact with a duplicate ID would throw an exception, directly reflecting the uniqueness requirement.

In the task service, the unit tests verified input validation for task IDs, names, and descriptions, ensuring each field met length and non-null constraints. Additionally, create, read, update, and delete operations were tested thoroughly, including fallback behavior when a requested task was not found. This comprehensive testing ensured the system could manage tasks robustly and reliably.

The appointment service presented additional complexity due to the temporal nature of appointments. Tests specifically validated that appointment IDs were unique and constrained in length, but more importantly, they enforced that appointment dates were always set in the future. Defensive copying of mutable date objects was also verified to prevent external modification of appointment data, ensuring data integrity.

This testing approach was closely aligned with the software requirements. Each functional constraint defined in the requirements such as maximum ID lengths, non null fields, and valid dates was covered by corresponding unit tests. For example, the test method that checks invalid

appointment dates by throwing an exception confirms strict adherence to the rule that appointment dates must be in the future.

The overall quality of the JUnit tests is demonstrated by the breadth and depth of coverage across all major code paths including positive cases, boundary conditions, and negative inputs. Coverage metrics, while dependent on tooling, would indicate high coverage since tests address all constructors, setters, getters, and exception-throwing conditions. The use of JUnit assertions such as assertThrows and assertEquals further guarantees precise validation of expected behaviors and error handling.

Writing the JUnit tests was an iterative and thoughtful process. To ensure the tests and code were technically sound, defensive programming techniques were used, such as defensive copying of Date objects in the appointment service. This was verified by comparing copies to originals, ensuring immutability was preserved. Efficiency was maintained by reusing setup methods annotated with beforeEach to initialize common test data, avoiding duplication and minimizing test execution time. Grouping logically related assertions within single test methods also helped maintain clarity and focus.

Reflection

Several software testing techniques were employed throughout the project. The primary technique was unit testing, which isolates and tests individual methods and classes to verify correctness at the smallest code unit level. This was complemented by boundary testing, where inputs at the limits of valid ranges such as maximum string lengths for IDs and names were tested to confirm correct behavior under edge conditions. Additionally, negative testing was widely used by supplying invalid data to confirm that the application robustly rejects or handles these cases gracefully.

Other testing techniques were not utilized but could be beneficial in broader project contexts. For instance, integration testing would verify how different modules such as contact, task, and appointment services interact, ensuring data flows and combined behaviors function correctly. System testing evaluates the entire application as a whole to confirm end-to-end functionality, while performance testing assesses how the system behaves under load, which is critical for scalable, real-world applications. Mutation testing, which involves introducing deliberate faults to assess test suite effectiveness, was not performed but would enhance confidence in test coverage and quality.

The mindset adopted was one of cautious thoroughness and professionalism. Recognizing the complexity and interrelationships of the code was essential, especially given mutable objects like Date in the appointment service, where careless handling could cause bugs difficult to trace. For example, tests ensured defensive copying prevented external mutation of appointment dates, demonstrating a careful appreciation of potential side effects.

To limit bias in reviewing my own code, I intentionally wrote tests targeting failure cases and invalid inputs rather than only testing expected happy path scenarios. This approach helped challenge assumptions and reduce confirmation bias. For instance, tests that deliberately supplied invalid IDs or past dates were essential to uncover weaknesses. I recognize that testing one's own code inherently risks bias, making a disciplined, skeptical approach vital to ensure code quality.

Discipline in quality assurance is paramount. Cutting corners in testing risks accumulating technical debt, leading to unstable, hard to maintain codebases. By committing to thorough testing, covering all requirements, edge cases, and error conditions, I ensure long-term maintainability and reliability. To avoid technical debt, I plan to enforce strict coding and testing

standards, use automated tests integrated into continuous integration pipelines, and regularly refactor code. This commitment safeguards the software's quality throughout its lifecycle.