Lyth hammad

CS-320 Project Two

08/13/2025

Summary

Unit Testing Approach for Each Feature

For ContactService, my unit testing approach focused on verifying every requirement from the specifications. I tested that contact IDs were not null, did not exceed the maximum allowed length, and remained immutable after creation. I also tested that first and last names were not null and met the character length restrictions, that phone numbers contained exactly 10 digits, and that addresses did not exceed the length limit. The service's add, update, and delete functions were tested to ensure they behaved as expected, with invalid inputs resulting in exceptions.

For TaskService, I used the same methodical approach. I confirmed that the task ID met all requirements, that names and descriptions stayed within their length limits, and that null values were rejected. I also tested the add, update, and delete functions to ensure the service handled both valid and invalid scenarios correctly.

For AppointmentService, I verified that appointment IDs were valid, that dates were not set in the past, and that descriptions met their constraints. I tested the service's ability to add, update, and remove appointments, ensuring that any invalid ID or past date was rejected.

Alignment With Software Requirements

My testing approach directly aligned with the project's requirements because every individual requirement was tied to at least one test. For example, when the specification required that phone numbers must be exactly 10 digits, I included tests that used both shorter and longer numbers to confirm that the service rejected them. Similarly, for appointments, I included tests that used dates set in the past to verify that the service threw an exception in those cases. This ensured complete coverage of the stated requirements.

Quality and Effectiveness of JUnit Tests

I measured the quality of my JUnit tests using the Eclipse coverage tool. The results were:

- ContactService and Contact: 87.4% coverage
- TaskService and Task: 85.7% coverage
- AppointmentService: 85.6% coverage

The only lines not covered were simple getters or default return statements. This high coverage confirmed that my JUnit tests effectively executed the majority of the functional paths and edge cases within the application.

Experience Writing JUnit Tests

While writing the tests, I focused on identifying and covering edge cases. I made sure that each test followed a clear structure with setup, execution, and verification steps. This made my tests easy to understand and maintain. To keep them efficient, I avoided redundancy by grouping similar cases into a single test where appropriate, which reduced unnecessary repetition while still maintaining thorough validation.

Reflection

Testing Techniques Used

The main testing techniques I employed were:

- Unit Testing – I tested each class and method in isolation to confirm that they met the requirements independently.
- Boundary Value Analysis – I tested inputs at the minimum, maximum, and just beyond allowable limits, such as maximum character lengths for names and addresses.
- Equivalence Partitioning I grouped inputs into valid and invalid categories, testing one representative from each to ensure coverage without excessive repetition.

- Black-Box Testing I tested functionality based on expected outputs for given inputs without relying on knowledge of the internal code implementation.

Testing Techniques Not Used

- Integration Testing – Not used because this project focused on testing each service separately rather than their interaction.
- System Testing – Not used because I did not test the entire mobile application workflow, only the backend service logic.
- Regression Testing – Not applied since there were no code changes after the initial test creation, though it would be valuable in future updates to ensure new changes do not break existing features.

Practical Uses of These Techniques

Unit testing is ideal for validating small sections of code early in development, boundary value analysis is essential when working with strict input limits, and equivalence partitioning reduces the total number of required test cases while still maintaining strong coverage. Black-box testing is especially useful when validating functionality without bias toward the underlying implementation.

Mindset

Caution and Complexity Awarenes

I approached testing with caution by thinking about what could go wrong and testing inputs that might cause issues, such as null values, overly long strings, and dates in the past. This helped me identify edge cases and appreciate the complexity of the validation logic in each service.

Limiting Bias in Testing

Because I was testing my own code, I made a conscious effort to remain objective. I designed

tests under the assumption that my implementation might have flaws. This mindset prevented me from unintentionally ignoring certain invalid scenarios simply because I believed my code would handle them.

Commitment to Quality and Avoiding Technical Debt

I believe that cutting corners in testing creates long-term risks, so I committed to achieving high coverage and thorough test cases. To avoid technical debt in the future, I plan to maintain reusable, well-documented test cases, consistently run regression tests after code changes, and track any limitations for future improvements. This ensures that quality remains a priority throughout the software development lifecycle.