Project Two

Caleb Luna

SNHU

For this project, I created unit tests for the three core features of the mobile application: contact, task, and appointment services. My approach focused on validating both correct input handling and rejection of invalid or out of bound inputs.

For the contact service, I tested proper creation and update of contacts while enforcing requirements such as unique identifiers, valid phone numbers, and maximum lengths for names. For example, in ContactTest I included a test where creating a new contact with a null identifier was expected to throw an exception. This directly aligned with the requirement that IDs cannot be null and must remain immutable.

For the task service, I confirmed that each task could be created with a valid ID, name, and description while enforcing length restrictions. In TaskServiceTest, I validated that adding a task worked as intended by first creating a task, adding it to the service, and then checking that the stored task name matched the original input. I also included tests that attempted to update tasks with invalid values, which correctly resulted in exceptions. This ensured that requirements for task name and description lengths were satisfied.

For the appointment service, I validated that future dates were enforced, IDs were unique, and descriptions were within range. In AppointmentTest, I created a case where an appointment was given a past date, and the test confirmed that an exception was raised. This satisfied the requirement that appointments must always be scheduled in the future.

My unit testing approach was highly aligned with the requirements because each test targeted a specific constraint or rule given in the project specification. For instance, the requirement that contact IDs are immutable was confirmed through tests that verified IDs never changed after initialization. Similarly, the requirement that appointment dates must be in the future was directly tested by comparing both past and valid future dates. Each class was validated against its acceptance criteria.

The overall quality of my JUnit tests was high, as measured by both test coverage and the breadth of test cases. Using coverage tools, I was able to confirm that nearly all public methods across the three classes were exercised. In addition, the inclusion of boundary cases, such as maximum string lengths and null inputs, ensured robust coverage. Because all paths of validation logic were tested, I am confident the tests were effective.

Writing these tests required me to carefully balance thoroughness with efficiency. To ensure technical soundness, I used assertions that validated class behavior directly. In TaskTest, for example, I created a task and then confirmed that the stored name was the same as the one provided at initialization. This verified that the object’s state matched expectations. Efficiency was achieved by reusing test objects instead of recreating them unnecessarily. In AppointmentTest, I constructed a valid appointment object and then ran multiple assertions against it, such as verifying its description and confirming that it was not null. This avoided redundancy and made the suite easier to maintain.

The techniques I used included black-box testing, boundary value analysis, and exception testing. Black-box testing allowed me to focus on whether inputs produced the correct outputs without considering internal code. Boundary values ensured that limits, such as maximum name lengths, were respected. Exception testing confirmed that invalid cases, such as null IDs or past dates, were rejected. I did not use integration or system testing, which would have validated interactions between services and the entire application. While unnecessary at this stage, they would be critical in later development.

I adopted a cautious mindset, assuming that any input could cause errors. For example, I deliberately passed null values to confirm that constructors enforced validation. This caution helped me account for edge cases and avoid overlooking hidden flaws. I also worked to limit bias by writing tests that intentionally challenged my own implementation, such as attempting to update tasks with invalid names. Without this mindset, I might have unconsciously trusted my own code.

I remained disciplined in writing complete tests because skipping checks can lead to technical debt. For example, if null input tests were ignored, failures could occur at runtime later. To avoid debt in future projects, I plan to apply test-driven development, which ensures only code that passes defined tests is implemented. This approach, as emphasized in JUnit documentation, improves both reliability and maintainability.

Overall, my JUnit test suites for the contact, task, and appointment services were aligned with requirements, thorough in coverage, and effective in ensuring quality. The use of black-box testing, boundary value analysis, and exception handling validated both expected and invalid cases. Reflecting on this project, I learned the importance of caution, reducing bias, and maintaining discipline. These lessons will guide me in avoiding technical debt and ensuring maintainable, reliable software in future projects.