Jacob McLemore

Professor Huang

October 19th, 2025

Project Two

**Summary:**

During this project, my main focus was on unit testing three main features of the mobile application: ContactService, AppointmentService, and TaskService. Each service was tested individually using JUnit to ensure that they performed as expected according to the project’s requirements. The testing approach I used was simple but effective. I created intentional tests with both valid and invalid inputs to confirm that the program handled expected behavior and errors correctly. This included writing test cases that purposely triggered IllegalArgumentException errors to verify that the program’s validation logic worked properly.

My testing approach aligned closely with the project’s software requirements because each service was designed to enforce data integrity and validation. For example, in the ContactService, each contact needed a unique ID, along with non-null fields for the name and phone number. I designed JUnit tests that reflected those exact constraints, ensuring that the requirements were verified through automated testing. Similar logic applied to AppointmentService, which required valid date and time fields, and TaskService, which required accurate descriptions and identifiers for each task. These tests provided direct evidence that my approach was built around meeting the intended requirements rather than just testing for functionality.

The overall quality of my JUnit tests was supported by an approximate 80% coverage rate. While not perfect, this level of coverage demonstrated that a majority of the code was being tested and that most logical paths were verified. The tests included both positive and negative scenarios, which helped validate not only that the services worked as expected but also that they failed correctly when given invalid inputs. I know the JUnit tests were effective because failed tests only occurred when they were intentionally designed to fail, which confirmed that the validation logic and exception handling were functioning properly.

Writing the JUnit tests required careful thought to ensure the code was technically sound. For example, I used assertions like  
 assertThrows(IllegalArgumentException.class, () -> contactService.addContact(test));  
 to confirm that errors were caught correctly. I also used methods such as assertEquals() to confirm that objects were being created and stored properly. These small details helped confirm that the logic behind the service classes behaved exactly as the requirements intended.

In terms of efficiency, I avoided redundancy by reusing test data and writing concise methods. For example, I created helper functions to build mock contacts, appointments, and tasks rather than rewriting new objects for every test. This helped keep the code organized and avoided unnecessary repetition. By keeping the tests modular and consistent, I could easily adjust one section without affecting others, which saved time and reduced potential bugs during development.

**Reflection:**

Testing Techniques

The primary testing technique I used was unit testing, carried out with JUnit. Unit testing focuses on verifying that individual parts of the code — in this case, service classes — function correctly in isolation. The strength of unit testing lies in its precision: it allows small sections of code to be tested independently, making it easier to locate and fix problems early in development. I also incorporated negative testing, where I intentionally used invalid inputs to ensure the program threw proper exceptions. This is an important practice because it tests the system’s ability to handle incorrect data safely and predictably.

Some testing techniques I did not use in this project included integration testing, system testing, and boundary value analysis. Integration testing focuses on checking how multiple modules interact together. For example, ensuring that the TaskService and AppointmentService share data correctly. System testing is performed at a higher level, validating that the entire application works as a complete product and meets all requirements. Boundary value analysis tests inputs at their extreme limits, such as the longest possible name or the earliest allowable date, to ensure stability under edge conditions. These techniques weren’t necessary here because the assignment focused only on verifying individual service functionality rather than full system behavior.

In practice, these other testing methods are very useful in larger projects. For example, integration testing is critical when an application relies on multiple APIs or databases that need to exchange information accurately. System testing ensures that after all modules are combined, the overall product still performs according to expectations. Boundary value analysis is often applied in safety-critical systems or financial software, where even small input errors can lead to major issues. While not used in this project, understanding when and how to apply these techniques is important for real-world software development.

**Mindset:**

While working on this project, I approached testing with a mindset of caution and precision. Acting as a software tester required me to look at my code from the perspective of someone trying to break it rather than defend it. I tried to anticipate potential failures or misuse by the end user. For instance, I tested what would happen if a null contact was added or if a task ID was duplicated. These cases helped me see how small mistakes could cause much bigger problems later. Appreciating the complexity and interrelationships of the code was essential because each service depended on proper data handling because one small issue could easily cascade into multiple failures if left unchecked.

Limiting bias was also an important part of the process. When you write and test your own code, it’s easy to assume your logic is correct and unintentionally overlook flaws. To counter that, I wrote tests with the mindset that the code might fail, even when I felt confident it wouldn’t. For example, I wrote assertions to double-check the exact values stored in the services rather than assuming that adding an item automatically succeeded. This mindset helped me spot potential weak points and reinforced the idea that testing isn’t about proving your code right — it’s about finding where it’s wrong.

Finally, I learned the importance of being disciplined about quality and thoroughness in testing. Cutting corners might save time initially, but it leads to technical debt, meaning future issues become harder and more expensive to fix. By running thorough tests early, I was able to catch logic errors that would have been difficult to trace later on. Going forward, I plan to avoid technical debt by sticking to a consistent testing routine, maintaining documentation, and running automated test suites regularly after making changes. These habits not only help ensure code quality but also reflect professionalism and accountability as a software developer.