In the development of the mobile application for the client I implemented a methodical unit testing approach using the JUnit framework. The three primary features, which were contact, task, and appointment services, were each supported by their own domain objects and service layers. For each service I wrote tests to validate both object construction and business logic operations, such as adding, updating, and deleting records. A key part of my strategy was sticking to the principle of writing one assertion per test, ensuring that each test method clearly documented a single behavior. This made the tests easier to read, maintain, and debug when failures occurred.

My approach was strongly aligned with the customer’s stated software requirements. For example, the contact object required a non-null, unique ID of no more than 10 characters, and I wrote separate unit tests to validate null input, excessive length, and successful creation. As another example, the phone field was constrained to exactly 10 digits, which I tested through both constructor input and setter methods. In the contact service the requirement to update only specific fields was validated through methods like updateFirstName(), and verified through retrieval using getContact() to ensure the mutation occurred on the stored object, rather than relying on in-memory references.

The overall quality of the JUnit test suite was confirmed through a JaCoCo coverage analysis. The report showed 99% instruction coverage, 100% branch coverage, and 100% class coverage across all testable components. This near-complete coverage assures me that the tests covered not only the happy paths, but also the negative and boundary cases. For example, in the task service, I validated behavior when attempting to update a non-existent task, and in the appointment service, I verified that an appointment date could not be set in the past.

Writing these tests provided a valuable opportunity to refine my skills in defensive programming and test-driven development. I paid close attention to exception handling and invalid input coverage. For instance, in the TaskTest class I included a test specifically for each invalid input, such as testSetInvalidNameNullThrowsException, rather than grouping multiple assertions together. This approach helped me pinpoint failing logic quickly, and reinforced the importance of modularity in testing.

To ensure my code was technically sound I not only validated correct outcomes but also enforced clear separation of responsibilities. For example, in the service classes each update method invoked a retrieval method before applying changes to ensure that the stored object was actually mutated. In TaskServiceTest, this was evidenced by retrieving the task using getTask() before asserting updates. I also ensured efficiency by leveraging @BeforeEach setup methods, and avoiding code duplication in test creation logic.

The primary software testing technique I used throughout this project was unit testing. Unit testing involves writing individual tests to verify the behavior of a single method or function in isolation. Each test focused on one behavior, such as rejecting null input or enforcing string length constraints, and validated either the creation or mutation of domain objects. I complemented these with negative tests to confirm that exceptions were thrown when expected.

Other software testing techniques that were not applied in this project include integration testing, end-to-end testing, and exploratory testing. Integration testing is used to verify that multiple components interact correctly, and would be useful if, for example, these services interacted with a shared database or user interface. End-to-end testing evaluates the behavior of the entire application in a start-to-finish context, while exploratory testing relies on manual interaction by testers to identify issues that automated testing might miss. While these techniques were not relevant for this small in-memory project, they are essential in larger production systems.

Each testing technique has its place depending on the development context. Unit testing is most effective in early development when isolating logic and verifying edge cases. Integration and end-to-end testing become important in distributed or full-stack systems. Exploratory testing is particularly valuable for user-facing applications to verify functionality against business requirements, and uncover usability issues.

Throughout the project I approached my testing with a cautious, detail-oriented mindset. This mindset was essential to maintaining the integrity of the services, especially when dealing with input constraints like fixed length fields or non-null values. For example, in the AppointmentTest class I validated that the constructor would reject dates in the past, which required using java.util.Date comparisons and helper methods to simulate both past and future dates accurately.

To reduce bias in my testing, I intentionally structured my development workflow so that test creation followed implementation. By distancing myself from the implementation when writing tests, I was able to evaluate the code from a more neutral, user-oriented perspective. I also made sure to test for invalid inputs, and not just the happy path, which is a common blind spot when developers test their own code. For instance, even though I knew the contact constructor worked with correct inputs, I tested it with nulls and oversized strings to ensure it would fail gracefully under incorrect usage.

Finally, I maintained a disciplined commitment to quality. I avoided shortcuts, such as grouping multiple assertions into one test, or skipping validation for "obvious" constraints. Doing so helped prevent missed bugs, and reinforced the habit of writing thorough, maintainable code. Technical debt can easily build up from minor decisions, like allowing a setter to skip validation, or skipping one invalid case in testing. I plan to avoid this by continuing to follow best practices, maintaining high test coverage, and writing modular code that is easy to verify.