**Summary**

The application was developed to support a client’s operational needs through three core modules: Contact Management, Task Management, and Appointment Scheduling. Each module was designed to handle critical business information reliably while meeting specific software requirements for data accuracy, error handling, and usability. The development approach incorporated defensive programming principles to prevent invalid data entry and operational inconsistencies. Extensive JUnit testing was performed to verify correct functionality, validate edge cases, and ensure the modules met requirements for stability and maintainability. The following sections detail the testing approach and results for each feature, the methods used to ensure code quality and efficiency, and reflections on the testing techniques and mindset applied during the project.

The Contact Management module was implemented using the Contact and ContactService classes. The software requirements specified that each contact must have a unique ID, a valid 10-digit phone number, and complete name and address fields. My unit testing approach ensured alignment with these requirements by testing both valid and invalid input cases. For example, in ContactServiceTest, I used: **assertThrows(IllegalArgumentException.class, () -> {**

**service.addContact(new Contact("001", "Alice", "Jones", "1234567890", "111 Pine Rd"));**

**});**

This confirmed that any contact with a phone number shorter than ten digits would be rejected. Positive-path tests confirmed that valid contacts were accepted and retrievable using their unique IDs. Coverage included adding, updating, and deleting contacts to verify that data integrity was preserved at all stages. The overall quality of these tests was high because they were comprehensive, including both expected and unexpected scenarios, and confirmed compliance with all specified validation rules.

The Task Management module was built around the Task and TaskService classes. The requirements dictated that each task must have a unique ID, a non-empty name, and a description not exceeding a set character limit. The testing strategy aligned with these requirements by covering creation, retrieval, and duplication prevention. For example, in TaskServiceTest, I validated the uniqueness rule with:

**assertThrows(IllegalArgumentException.class, () -> {**

**taskService.addTask(new Task("001", "Task 1", "Description 1"));**

**taskService.addTask(new Task("001", "Task 2", "Description 2"));**

**});**

This ensured that the system did not overwrite or silently ignore duplicate tasks. Additional tests checked for empty names using:

**assertThrows(IllegalArgumentException.class, () -> {**

**new Task("002", "", "Description");**

**});**

By verifying these failure conditions alongside success cases, I ensured that the module adhered closely to its specifications. The test coverage was effective, touching on all known failure modes and ensuring that every business rule was enforced in practice.

The Appointment Scheduling module consisted of the Appointment and AppointmentService classes. The primary requirement was that appointments must have a unique ID, a valid description, and a future date. My testing approach directly targeted these requirements, particularly the time-based restriction. In AppointmentTest, I confirmed that past appointments were blocked:

**assertThrows(IllegalArgumentException.class, () -> {**

**new Appointment("A01", new Date(System.currentTimeMillis() - 1000), "Past Meeting");**

**});**

I also validated correct handling of future appointments:

**Date futureDate = new Date(System.currentTimeMillis() + 86400000);**

**Appointment appt = new Appointment("A02", futureDate, "Planning Session");**

**assertEquals(futureDate, appt.getAppointmentDate());**

Testing included boundary scenarios such as dates just a few milliseconds ahead of the current time, ensuring that the validation logic was precise. This gave me confidence that the module would perform reliably in real-world scheduling situations.

Throughout the process, I focused on making my JUnit tests both technically sound and efficient. Technical soundness came from testing each business rule directly and using assertions that matched the intended logic exactly. For instance, **assertEquals(task, taskService.getTask("001"));** confirmed correct retrieval functionality for the Task module. Efficiency was achieved by using **@BeforeEach** to initialize services and common objects once per test run, reducing repetitive code. In TaskServiceTest, this looked like:

**@BeforeEach**

**public void setUp() {**

**taskService = new TaskService();**

**}**

This approach allowed me to keep the tests concise without sacrificing clarity, making maintenance easier in the long term.

**Reflection**

The primary testing technique used was unit testing with JUnit. This approach isolated individual classes and methods to verify their correctness without external dependencies. I also applied negative testing, deliberately providing invalid input to confirm that the system responded with appropriate exceptions. An example of negative testing was in the Appointment module, where I tested past-date entries to confirm rejection. Techniques not used included integration testing and system testing, which would have verified how modules interact when combined. While these were outside the project’s scope, they would be valuable in later stages to ensure seamless cross-module operations. Unit and negative testing provided fast feedback loops and helped catch logic flaws early, reducing future rework and making the modules more robust.

I approached testing with a mindset of caution, recognizing that even minor oversights could cause significant operational problems for the client. This meant writing tests that covered not just common use cases but also extreme and unlikely inputs. To reduce bias, I avoided making assumptions about my code’s correctness by treating my own implementation as though I were reviewing someone else’s work. Writing some tests before finalizing code also helped keep the focus on requirements rather than personal coding preferences. Maintaining discipline was critical; I did not skip tests for cases I believed were “too obvious” because untested paths can easily hide bugs. This discipline will help avoid technical debt in the future, as the modules can evolve without fear of regressions in their core logic.

The Contact, Task, and Appointment modules meet their respective requirements and have been thoroughly validated through unit and negative testing. The coverage ensures that business rules are enforced consistently and that invalid inputs are handled gracefully. Moving forward, I recommend adding integration testing to verify cross-module workflows, implementing persistent data storage, and expanding date handling to account for time zones and daylight savings time changes. These steps would further enhance system reliability and prepare them for production deployment. The combination of cautious planning, unbiased testing, and disciplined execution has produced a technically sound and efficient codebase that can be confidently built upon in future phases.