When developing the mobile application, I approached unit testing for the ContactService, TaskService, and AppointmentService systematically to ensure that each feature worked as intended and met the outlined requirements. For the ContactService, I focused on validating key contact details like IDs, names, phone numbers, and addresses. For example, I tested that the contactID was unique and within the character limit of 10, while phone numbers had to be exactly 10 digits long. Boundary tests, such as checking if the maximum length of an address (30 characters) was accepted, helped confirm that the service adhered to its specifications. Invalid inputs, like null values or excessively long names, were tested to ensure they would trigger exceptions, demonstrating the robustness of the system.

For the TaskService, my focus was on managing tasks effectively, including adding, updating, and deleting them. I wrote unit tests to verify that tasks could be created with valid IDs, names, and descriptions within their required lengths. For example, in testValidTaskCreation(), I confirmed that a task with a 20-character name and a 50-character description was processed successfully. Similarly, I ensured the service would reject invalid inputs, such as tasks with null IDs or descriptions longer than 50 characters. Tests like testDeleteNonExistentTask() confirmed that attempting to delete a non-existent task would throw an exception, validating error handling.

For the AppointmentService, the emphasis was on scheduling and managing appointments. Tests ensured that appointments could only be added with valid inputs, such as future dates and descriptions up to 50 characters long. For instance, in testAddAppointmentWithBoundaryValues(), I tested adding an appointment with the maximum allowable values for both the ID and description to confirm proper handling of edge cases. The service also included robust error handling, with tests like testDeleteNonExistentAppointment() verifying that the application threw appropriate exceptions for invalid operations, such as deleting an appointment that didn’t exist.

Across all three services, my unit testing approach was closely aligned with the software requirements. The tests were designed to confirm that the functionality met the exact specifications outlined for each feature. For example, the requirement for unique IDs in all services was thoroughly tested by trying to add duplicate IDs, which correctly triggered exceptions in the respective services. The boundary tests further demonstrated alignment by ensuring that maximum input values, such as the longest allowable descriptions or IDs, were handled as expected.

The quality of the JUnit tests was evident in their comprehensive coverage of each feature’s functionality. Based on coverage percentage I was able to get above an 80% tested rate. Meaning that 80% of the outcomes were tested. Getting the tests to 100% required more testing of proper values as all the invalid ones were already being tested. For instance, tests like testInvalidContactID() in ContactService, testInvalidTaskDescription() in TaskService, and testInvalidAppointmentDate() in AppointmentService ensured that invalid inputs were consistently rejected. The use of assertions, such as Assertions.assertThrows() for exception handling and Assertions.assertEquals() for verifying outputs, provided clear feedback on whether the application behaved as required.

Writing the JUnit tests was a valuable exercise in maintaining code quality. I ensured technical soundness by testing both valid and invalid scenarios. For example, in AppointmentServiceTest, I wrote testAddAppointmentWithBoundaryValues

()

/\*\*

     \* Tests adding an appointment with boundary values for ID and description.

     \*/

    @Test

    void testAddAppointmentWithBoundaryValues() {

        AppointmentService service = new AppointmentService();

        String maxID = "1234567890"; // 10 characters

        String maxDescription = "a".repeat(50); // 50 characters

        Appointment appointment = new Appointment(maxID, new Date(System.currentTimeMillis() + 86400000), maxDescription);

        service.addAppointment(appointment);

        Assertions.assertEquals(appointment, service.getAppointment(maxID));

    }

to confirm that a 50-character description was accepted without errors, ensuring compliance with requirements. Efficiency was also prioritized by reusing objects and employing utility methods where applicable. In TaskService, for example, I reused a valid task object to test both adding and retrieving operations, streamlining the tests while maintaining thoroughness.   
@Test

    void testAddTask() {

        TaskService service = new TaskService();

        Task task = new Task("123", "Task Name", "Description");

        service.addTask(task);

        assertEquals("Task Name", service.getTask("123").getName());

    }

@Test

    void testUpdateTask() {

        TaskService service = new TaskService();

        Task task = new Task("123", "Task Name", "Description");

        service.addTask(task);

        service.updateTask("123", "Updated Name", "Updated Description");

        assertEquals("Updated Name", service.getTask("123").getName());

        assertEquals("Updated Description", service.getTask("123").getDescription());

    }

By systematically testing the features of ContactService, TaskService, and AppointmentService, I ensured the code was reliable, efficient, and aligned with the requirements. This approach not only validated the functionality of each service but also guaranteed that edge cases and error conditions were properly handled, providing a high level of confidence in the application’s quality.

In this project, I used several software testing techniques. Boundary value analysis was especially helpful for testing input limits, such as the maximum length of IDs and descriptions. I also used exception testing to ensure that invalid inputs, like null values or duplicate IDs, triggered the correct error messages. For example, I used Assertions.assertThrows() to confirm that adding an appointment with a duplicate ID would throw an exception. Another technique I used was CRUD testing, which focused on validating that the add, retrieve, and delete operations worked as intended.

There were some techniques I did not use, such as mocking and performance testing. Mocking is useful for testing systems that interact with external services, as it simulates those interactions without requiring the actual services to be available. Performance testing, on the other hand, checks how well a program performs under heavy use, like handling thousands of appointments. While these weren’t necessary for this project, they are important for larger systems that require scalability or involve external dependencies.

Each technique has practical uses. For example, boundary testing is crucial for user-facing systems to handle input limits gracefully. Mocking is ideal for applications with APIs or microservices, and performance testing is essential for large-scale platforms like online shopping websites. By choosing the right techniques for this project, I ensured the program worked reliably for its intended purpose.

When working on this project, I was cautious and appreciated the complexity of the code. Testing inputs like null values or past dates in testInvalidAppointmentDate() helped me account for real-world scenarios that could cause errors. This mindset ensured that I didn’t overlook any potential issues. I also tried to limit bias by following the requirements closely. For example, the limit of 50 characters for descriptions came directly from the specifications, ensuring my tests were objective and not influenced by personal assumptions.

It’s important to stay disciplined and commit to quality when writing and testing code. Cutting corners can lead to bugs and technical debt, which make future maintenance harder. For example, by writing thorough exception tests like testDeleteNonExistentAppointment(), I ensured the program was resilient to unexpected inputs. To avoid technical debt in the future, I plan to use automated testing tools and conduct regular code reviews. By doing so, I can maintain high-quality code that is both reliable and easy to update.