Unit Testing Approach

For this project, I implemented unit tests for the three core features of the mobile application: Contact Service, Task Service, and Appointment Service. Each service was tested using JUnit to verify that the customer’s functional requirements were met and that the code behaved correctly in both valid and invalid conditions.

For the Contact Service, the testing approach involved verifying creation, deletion, and updating of contact data. For example, in ContactTest.java, the method testCreateValidContact() confirmed that a contact object could be created successfully with valid data, while another test ensured that invalid inputs such as overly long IDs or null fields threw an IllegalArgumentException. The corresponding ContactServiceTest.java verified the service layer operations, such as adding and retrieving a contact by ID using assertions like assertNotNull(service.getContact("100")).

The Task Service tests followed a similar pattern. The TaskTest.java verified field validation, ensuring that the taskId, name, and description adhered to their respective length and null constraints. In TaskServiceTest.java, I used a @beforeeach method to initialize the service before each test, ensuring isolation and consistency. I then tested the addition and update operations using assertions such as assertEquals("Task1", service.getTask("1").getName()), confirming that the object was correctly stored and updated.

For the Appointment Service, I used date validation as a major focus. In AppointmentTest.java, the test method testValidAppointmentCreation() created an appointment with a future date, while additional tests ensured that attempting to set past dates triggered exceptions. The AppointmentServiceTest.java verified service-level functionality such as adding and deleting appointments by ID. These tests utilized helper methods like futureDate() to generate valid date objects dynamically, ensuring reliability across test runs.

Alignment to Software Requirements

My unit testing approach was directly aligned with the customer’s software requirements for each service. The requirements emphasized that all IDs must be unique, non-null, and unmodifiable, and that text fields had specific length restrictions. The JUnit tests explicitly covered each of these requirements by asserting correct behavior on valid data and exception handling for invalid cases.

For example, the Contact tests verified that phone numbers had exactly 10 digits, the Task tests confirmed that names and descriptions were within character limits, and the Appointment tests ensured that the appointment date could not be in the past. By covering both positive and negative cases, I was able to verify that the software fully adhered to its functional and validation requirements.

Quality and Effectiveness of the JUnit Tests

The JUnit tests were effective in verifying functionality across all services. The test suite achieved high coverage, over 80% of the source code, by including tests for all key methods and potential failure points. The combination of boundary tests (for example, maximum string lengths), negative tests (null and invalid values), and integration-like service tests ensured that nearly all logical branches were executed during testing.

The tests also demonstrated good technical quality. For instance, reusable setup methods in the Task and Appointment tests reduced redundancy and improved maintainability. Assertions were clear and specific, such as assertEquals("123", contact.getContactId()) and assertThrows(IllegalArgumentException.class, () -> new Appointment("1", pastDate, "desc")). These direct assertions verified outcomes precisely and made debugging straightforward.

Code efficiency was also considered by avoiding unnecessary object creation and reusing initialized services between tests. The logical separation between class-level tests and service-level tests maintained clean organization, which is an important part of effective testing architecture.

Testing Techniques

The primary software testing techniques employed in this project were unit testing, boundary testing, and exception testing. Unit testing focuses on validating individual units of code in isolation, which ensured that each service and class behaved as expected. Boundary testing involved testing the limits of acceptable input values, such as the maximum character length for names and descriptions. Exception testing validated that invalid data triggered the correct exceptions, confirming that error-handling logic was implemented properly.

Other software testing techniques not used in this project include integration testing, system testing, and acceptance testing. Integration testing would focus on verifying the interaction between multiple components or services, while system testing would evaluate the application as a whole against user requirements. Acceptance testing, on the other hand, would be performed by the end user to confirm the product meets real-world needs. While these techniques were not required for this specific project, they would be valuable in later development phases when combining the backend services with a user interface or database.

In practical terms, unit and boundary testing are best used during early development to catch logical and validation errors quickly. Integration and system testing, by contrast, are more appropriate for full-stack applications or microservices environments where multiple systems interact.

Mindset

While working on this project, I adopted a cautious and analytical mindset. As a software tester, I understood that even small mistakes in logic could lead to data errors or crashes. It was important to appreciate how different parts of the code, such as the service’s add, update, and delete methods, interacted with each other. For instance, when updating contact information, I had to ensure that the update logic did not inadvertently overwrite non-updatable fields like the contact ID.

To limit bias, I made sure to test my own code objectively by approaching each test as if it were written by someone else. Instead of assuming my methods would work, I deliberately attempted to “break” the code by entering invalid values. This approach helped me uncover subtle issues that might have been overlooked if I had only tested with valid input. For example, I included tests for null or overly long fields that I might not have initially considered during development.

As a software engineer, maintaining discipline and commitment to quality is crucial. Cutting corners during testing can lead to technical debt, issues that accumulate over time and become harder to fix later. I plan to avoid technical debt in future projects by following consistent testing practices, maintaining clear documentation, and using continuous integration tools that automatically run test suites with every update.

By approaching software testing with thoroughness and patience, I can ensure that the code I deliver meets both functional and quality standards, supporting long-term maintainability and client satisfaction.

**References**

Ammann, P., & Offutt, J. (2016). *Introduction to Software Testing* (2nd ed.). Cambridge University Press.

Beizer, B. (1995). *Software Testing Techniques* (2nd ed.). Dreamtech Press.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). Wiley.

Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner’s Approach* (9th ed.). McGraw-Hill Education.

Whittaker, J. A. (2012). *How to Break Software: A Practical Guide to Testing*. Addison-Wesley.