**Summary and Reflections Report**

**Author: Umar Asif**  
**Project: Contact, Task, and Appointment Services for Mobile App**

**Summary**

As part of my work at Grand Strand Systems, I developed and tested the contact, task, and appointment services for a mobile application. My unit testing approach focused on validating object creation, field constraints, update functionalities, and deletion processes using JUnit. Each class was paired with corresponding test classes to ensure complete functionality coverage.

For the **Contact feature**, I tested the creation of contact objects with valid inputs and verified enforcement of field constraints, such as max character lengths and null value restrictions. For instance, the ContactTest class includes a test like assertThrows(IllegalArgumentException.class, () -> new Contact(null, "John", "Doe", "1234567890", "123 Main St")); to verify null ID validation.

The **Task feature** followed a similar approach. I wrote tests to verify that tasks could not be created with null values or invalid field lengths, like task names over 20 characters or descriptions over 50 characters. An example is: assertThrows(IllegalArgumentException.class, () -> new Task("12345678901", "Clean", "Valid description"));, which tests the max length for the task ID.

For the **Appointment feature**, my tests ensured future-dated appointments only, validated string length for descriptions, and confirmed appointment IDs are valid and immutable. For instance, the test assertThrows(IllegalArgumentException.class, () -> new Appointment("DC102", pastDate, "Back to the past")); ensures the appointment date cannot be in the past.

My approach was directly aligned with the software requirements. Each test class mirrored the business rules established in the project directions. The effectiveness of these JUnit tests was measured by achieving 100% test coverage across all classes using IntelliJ IDEA’s built-in code coverage tool.

To ensure the code was **technically sound**, I followed proper unit test structure using Arrange-Act-Assert (AAA) patterns. For example, in ContactServiceTest.java, I added and updated contacts using clear assertions to validate expected behavior:

contactService.updateFirstName("001", "Jane");

assertEquals("Jane", contactService.getContact("001").getFirstName());

To ensure **efficiency**, I minimized redundancy by using helper methods like createValidContact() within my test classes and avoided unnecessary object duplication. This helped streamline multiple test cases that shared setup logic.

**Reflection**

**Testing Techniques**

I employed **unit testing** using **black-box testing** and **boundary value analysis**. Black-box testing allowed me to focus on the input/output behavior of classes without being concerned with internal implementation. Boundary value analysis helped verify that constraints (like max length of strings) were strictly enforced at edge cases.

Other techniques I didn’t use include **integration testing** and **system testing**. Integration testing would involve testing interactions between services (e.g., how a ContactService might work with a NotificationService), which wasn’t applicable in this isolated in-memory implementation. System testing, which simulates real-world usage of the entire application, would only be relevant after building a complete UI.

In broader projects, black-box and boundary tests are ideal during early development phases. Integration and system tests become vital during late-stage development and before deployment to ensure the application behaves as expected end-to-end.

**Mindset**

Throughout the project, I maintained a mindset of **precision and caution**. Each service had detailed rules that had to be enforced exactly. For example, in AppointmentTest.java, I tested that past dates are invalid by creating a calendar instance one day in the past. Appreciating the **interrelationships** of the code was crucial—particularly ensuring the service classes correctly manipulated the underlying object structures.

To limit **bias** as the developer and tester of my own code, I assumed my implementation was faulty until proven correct. I deliberately wrote tests designed to break the code, such as creating objects with null fields, to simulate real-world input errors. Bias can arise when a developer assumes their logic is sound and writes tests that only confirm it. Instead, I focused on writing tests that could **falsify** assumptions.

Being disciplined in pursuing **quality** is fundamental. Cutting corners may seem to save time but leads to **technical debt**—errors that will resurface and demand more costly fixes later. To avoid this, I rely on test-driven development (TDD) principles when possible, and consistently validate all requirements through JUnit tests. For example, in ContactServiceTest.java, I verified not only that updates work but that updates on nonexistent IDs properly throw exceptions, reducing the chance of runtime errors in production.

In the field, I plan to keep using JUnit and automated CI pipelines to prevent regressions and maintain a clean, testable codebase.

**References**

JUnit 5 User Guide. (n.d.). Retrieved from <https://junit.org/junit5/docs/current/user-guide/> Oracle. (n.d.). The Java™ Tutorials. Retrieved from <https://docs.oracle.com/javase/tutorial/>