**Summary and Reflections Report**

### ****Summary****

#### ****Unit Testing Approach****

For the development of the mobile application’s backend services, I employed a unit testing strategy focused on three main features: **ContactService, TaskService, and AppointmentService**. My approach centered on **JUnit testing** to validate individual units of functionality, ensuring that each method performed as expected under various conditions.

#### ****Contact Class****

The Contact class manages individual contact details, enforcing constraints such as unique IDs and required fields. Instead of providing the code, I used the following code lines:

**Code lines:** Contact.java (lines 10-45)

##### **Unit Test Example: ContactService**

* Tested methods for adding, updating, and deleting contacts.
* Verified constraints such as unique IDs and required fields.

@Test

@DisplayName("Test to Update First Name.")

@Order(1)

void testUpdateFirstName() {

ContactService service = new ContactService();

service.addContact("John", "Smith", "3462185546", "1122 Boogeyman St, Houston, TX");

String contactID = service.contactList.get(0).getContactID();

service.updateFirstName("Alex", contactID);

assertEquals("Alex", service.getContact(contactID).getFirstName(), "First name was not updated.");

}

**Code lines:** ContactServiceTest.java (lines 15-55)

I designed the unit tests for ContactService to ensure that all CRUD operations function correctly and enforce constraints. For example, while testing the update methods, I confirmed that invalid inputs triggered exceptions and valid inputs were updated successfully. This helped maintain data integrity and enforce business rules.

#### ****Task Class****

The Task class defines a task with a unique identifier, a name, and a description.

**Code lines:** Task.java (lines 5-40)

##### **Unit Test Example: TaskService**

* Ensured task creation, modification, and deletion were functioning correctly.
* Tested invalid input scenarios to confirm exception handling.

@Test

void testDuplicateTaskId() {

TaskService service = new TaskService();

String taskId1 = service.addTask("Sami Task", "Testing description.");

String taskId2 = service.addTask("Muhammad Task", "Testing description.");

assertNotEquals(taskId1, taskId2);

}

**Code lines:** TaskServiceTest.java (lines 10-50)

The tests I developed were, by any measure, highly effective. There may have been better ways to implement them, but they achieved their intended purpose. The coverage percentage of TaskService was **100%**, verifying its robustness. I also refactored the{ addTask} method to ensure unique ID generation, which was tested in multiple scenarios.

#### ****Appointment Class****

The Appointment class ensures valid appointment scheduling with a proper date format.

**Code lines:** AppointmentP.java (lines 10-55)

##### **Unit Test Example: AppointmentService**

* Ensured appointment creation, rescheduling, and cancellations adhered to constraints.
* Validated time format and overlap restrictions.

@Test

void testInvalidDate() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

new Appointment("1", "2023-02-30", "Annual Checkup");

});

assertTrue(exception.getMessage().contains("Invalid date"));

}

**Code lines:** AppointmentServicePTest.java (lines 10-65)

One of the key challenges in the Appointment module was validating dates to ensure they were in the future. By implementing boundary testing, I was able to verify that incorrect date formats and past dates triggered exceptions while allowing valid scheduling.

My approach aligned with software requirements by ensuring compliance with specified constraints and functionalities. Unit testing ensures that individual components meet specified constraints, such as input validation and length restrictions, preventing invalid data from entering the system (Jain, 2024).

### ****Code Coverage Analysis****

To measure the effectiveness of my unit tests, I analyzed the code coverage metrics from the test execution. The test coverage report provides insight into the thoroughness of my tests across three key metrics:

* **Class Coverage:** 100% (All classes were tested at least once.)
* **Method Coverage:** 100% (Every method within the classes was tested.)
* **Line Coverage:** 91% (311/340 lines executed during tests.)

This coverage percentage demonstrates the effectiveness of the testing strategy, ensuring that core functionalities were thoroughly validated.

### ****Reflection****

#### ****Testing Techniques Employed****

1. **Unit Testing:** Tested individual methods and classes in isolation.
2. **Boundary Testing:** Checked edge cases, such as minimum and maximum allowed values.
3. **Exception Testing:** Verified proper handling of invalid input scenarios.

I adopted a meticulous approach while performing **exception testing** to ensure that incorrect inputs were properly rejected. For instance, updating a task description with more than 50 characters correctly threw an exception, reinforcing the reliability of the constraints.

#### ****Other Testing Techniques Not Used****

1. **Integration Testing:** Not included as we focused on unit testing and did not test interactions between multiple services.
2. **System Testing:** The project did not require testing the entire system as a whole.
3. **Performance Testing:** Since the focus was on functionality, performance aspects were not evaluated.

Each of these techniques is useful in different scenarios. For example, integration testing ensures that services work correctly together, which is crucial for complex applications (Jain, 2024).

#### ****Mindset Adopted in Testing****

While working on this project, I adopted a **cautious and detail-oriented mindset**. Recognizing the complexity and interdependencies of the code was essential. For example, modifying TaskService could impact Task, requiring careful retesting to prevent unintended side effects.

To limit bias, I followed a **test-driven development (TDD) approach**, writing tests before implementation to avoid assumptions about how the code should work. Developers often unconsciously favor positive test cases, leading to **confirmation bias**. Considering both expected and unexpected behaviors ensures higher test quality (Calikli et al., 2010).

#### ****Commitment to Quality****

To maintain code quality, I plan to:

* **Follow a test-first approach** (writing tests before coding).
* **Use code reviews** to get feedback on test coverage.
* **Implement automated test suites** to detect regressions early.

Adding **continuous integration (CI) tools** would allow automated execution of test cases, ensuring consistency in future updates (ShakeBugs, 2023).

### ****Conclusion****

Through this project, I developed a structured approach to **unit testing**, ensuring robust and high-quality tests. By using **JUnit tests** to validate service functionalities, I improved my understanding of **test-driven development, testing techniques,** and the importance of **disciplined coding practices**.

**References**

1. **Calikli, G., Bener, A. B., & Menzies, T. (2010).** "Confirmation Bias in Software Development and Testing: An Empirical Study." IEEE Transactions on Software Engineering.
2. **Jain, S. (2024).** "Software Testing Techniques: Explained with Examples." October 29, 2024. [Source](https://www.browserstack.com/guide/software-testing-techniques)
3. **ShakeBugs. (2023).** "8 Debugging Methods You Need to Know About." June 13, 2023. [Source](https://www.shakebugs.com/blog/app-debugging-methods/)
4. **Farcic, V., & Garcia, A. (n.d.).** "Test-Driven Java Development." [Source](https://techvify-software.com/wp-content/uploads/2022/04/test-driven-java-development-invoke-tdd-principles-for-end-to-end-application-development-with-java-viktor-farcic-alex-garcia.pdf)
5. **Mohammed Alshehabi (2025)** “**pro1\_Completed\_(CS320).”**