**Summary and Reflections Report**

**Summary**  
Throughout Project One, I focused on implementing and testing three key features of the mobile application: the contact service, the task service, and the appointment service. Each of these features required careful unit testing to ensure that they behaved according to the customer’s requirements and design specifications. My testing approach involved writing JUnit tests that validated each class’s constraints, functionalities, and error-handling behaviors.

**Unit Testing Approach and Alignment to Requirements**  
For each feature, I began by reviewing the software requirements in detail. For instance, the contact service requirements specified constraints on the length and format of fields like contactID, phone, and address. In the corresponding ContactTest.java, I created tests that constructed contact objects both with valid and invalid inputs to ensure these constraints were enforced. Similarly, in the TaskTest.java, I tested name and description length limits, and in the AppointmentTest.java, I verified that the appointment date could not be in the past. This approach directly aligned with the software requirements, as each test case mapped closely to one or more specified rules. For example, a test for the contact’s phone field length requirement looked like:

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

**new Contact("12345", "John", "Doe", "12345", "123 Main St");**

**});**

This assertion ensured that if the phone number was not exactly 10 characters, an exception would be thrown. This level of specificity and one-to-one mapping between requirements and test cases ensured a thorough and requirements-driven testing approach.

**Effectiveness of JUnit Tests and Coverage**  
I measured the effectiveness of these tests by observing the test coverage percentage provided by my integrated development environment (IDE) and testing tools. By aiming for high coverage—ensuring that all methods and branches of my code were tested—I could be confident that my tests were robust. For example, coverage reports indicated that nearly all lines of code in ContactService.java, TaskService.java, and AppointmentService.java were executed during the tests, including exception-throwing lines. Achieving close to 90% or more line coverage and 100% coverage for critical methods supported the claim that these tests were effective in uncovering issues.

**Experience Writing JUnit Tests and Technical Soundness**  
Writing JUnit tests for this project was both a learning experience and a validation of good coding practices. I ensured the code was technically sound by testing expected successes and failures. For example, in TaskServiceTest.java, I tested the addition and deletion of tasks:

**Task task = new Task("12345", "Test Task", "This is a test description.");**

**service.addTask(task);**

**assertEquals(task, service.getTask("12345"));**

Then I wrote a test for deletion:

**service.deleteTask("12345");**

**assertNull(service.getTask("12345"));**

By ensuring that each operation had a corresponding test, I confirmed that the code behaved correctly under both normal and edge conditions. Furthermore, I tested exception cases, like attempting to delete a non-existent task, which helped guarantee the code did not fail silently or behave unpredictably. These tests confirmed the logical integrity and error-handling capabilities of the services.

**Ensuring Code Efficiency**  
To ensure efficiency, I examined whether my tests were running quickly and avoiding redundant checks. For example, I reused already created objects and integrated multiple assertions in a single test method where appropriate, as long as it did not compromise clarity:

**Contact contact = new Contact("12345", "John", "Doe", "1234567890", "123 Main St");**

**service.addContact(contact);**

**service.updateFirstName("12345", "Jane");**

**service.updatePhone("12345", "0987654321");**

**Contact updatedContact = service.getContact("12345");**

**assertEquals("Jane", updatedContact.getFirstName());**

**assertEquals("0987654321", updatedContact.getPhone());**

By combining these steps into a single test, I reduced unnecessary initialization and teardown, thus maintaining efficiency. This approach also reflected efficient coding practices in the production code, as I avoided complex or redundant logic that could slow down the program.

**Reflection**

**Testing Techniques Employed and Alternatives**

In this project, I primarily employed unit testing and integration testing strategies. Unit testing allowed me to focus on individual classes and methods in isolation, ensuring that each component met its own requirements before integration. These tests were small, fast, and provided immediate feedback. Integration testing, on the other hand, validated that the services and objects worked correctly together, such as adding, updating, and deleting records in the service classes. These techniques are critical for catching bugs early and confirming that interactions between objects or services function as intended.

Other techniques not used in this project include system testing and acceptance testing. System testing would involve verifying the application as a whole in a production-like environment, ensuring that all integrated components work together seamlessly. Acceptance testing would focus on validating that the software meets the end-user’s needs and business requirements, often involving client or stakeholder feedback. These tests would be more applicable at later project stages, closer to deployment, when the code is feature-complete and stable.

In different development projects, unit and integration tests remain the backbone for catching issues early. System and acceptance tests become essential when verifying the product’s readiness for release and ensuring stakeholder satisfaction. Thus, a combination of these approaches can be chosen based on the project’s complexity, timeline, and risk tolerance.

**Mindset**

Working as a software tester requires a mindset of caution and thoroughness. I approached my testing tasks by assuming that code could fail in unexpected ways. Appreciating the complexity of the code and the relationships between various classes helped me design test cases that could reveal subtle defects. For example, I did not just test “happy path” scenarios; I tested invalid inputs, null values, and boundary cases. Without this cautious mindset, it would be easy to miss defects that only appear under rare conditions.

To limit bias, I approached the code as if I were unfamiliar with its implementation details, treating each component as a black box. By doing so, I forced myself to rely on the stated requirements rather than my assumptions of correctness. This approach reduced the tendency to overlook potential problems simply because I wrote the code myself. If I were the developer and tester for the same component, I recognize there is a risk of overlooking errors due to familiarity and overconfidence. One strategy to mitigate bias would be to have peer reviews or pair testing, where another team member examines my tests and code.

**Commitment to Quality and Avoiding Technical Debt**

Discipline in maintaining code quality is essential in software engineering. Cutting corners—such as skipping tests or ignoring known issues—leads to technical debt, which can accumulate and become costly or time-consuming to fix later. By adhering to thorough testing, employing code reviews, and refactoring regularly, I can avoid introducing technical debt into the codebase. For instance, if I notice a duplicated logic block during testing, I should refactor it to a single method to maintain clarity and reduce future maintenance costs.

As a practitioner, I plan to integrate testing into every stage of development. Automated test suites, continuous integration, and regular code reviews will help ensure that I catch defects early. By building quality into the process from the start, I will avoid the pitfalls of technical debt and maintain a codebase that is robust, adaptable, and easier to evolve over time.

**References**

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

Beck, K. (2003). Test-Driven Development: By Example. Addison-Wesley Professional.