**Summary and Reflections Report**

**Summary**

**Unit Testing Approach for Each Feature**

For Project One, I developed and tested three services: ContactService, TaskService, and AppointmentService. My primary unit testing approach involved using JUnit to create individual test classes for each service and its related models. Each test class was designed to verify correct behavior according to the project requirements.

* **ContactService**: I created tests to ensure contacts could be added, updated, and deleted correctly. I verified that invalid inputs, such as too long IDs or null fields, would trigger exceptions.
* **TaskService**: I wrote tests to validate creating tasks with proper inputs, ensuring that task fields could be updated where allowed, and ensuring that attempts to update restricted fields or use invalid input would fail properly.
* **AppointmentService**: I developed tests to confirm that appointments could be added and updated, with particular focus on ensuring that appointment dates could not be in the past and that other field constraints were enforced.

**Alignment to Software Requirements**

My testing approach was strongly aligned with the software requirements. For example, in the **TaskServiceTest** class, I created a test to ensure that tasks could not be updated with a null name, reflecting the requirement that all fields must be non-null. An example from my code:

assertThrows(IllegalArgumentException.class, () -> service.updateTaskName(task.getId(), null));

This ensured my testing validated the application's error-handling pathways as well as its standard functions.

**Effectiveness of JUnit Tests**

Although I did not formally measure code coverage with a tool, I ensured my JUnit tests covered all methods, edge cases, and failure paths for each service and model. I wrote both positive (valid input) and negative (invalid input) test cases, supporting a high level of code coverage. This comprehensive approach gives me confidence in the overall effectiveness of my JUnit tests.

**Experience Writing JUnit Tests**

To ensure technical soundness, I wrote tests that covered:

* Creating new objects with valid data:

assertEquals("New Task", task.getName());

* Handling invalid data gracefully:

assertThrows(IllegalArgumentException.class, () -> new Task(null, "Test", "Test Description"));

* Ensuring updates respected field constraints:

service.updateTaskName(task.getId(), "Updated Task");

assertEquals("Updated Task", service.getTask(task.getId()).getName());

I ensured efficiency by writing focused test methods that tested only one behavior at a time and avoiding redundant or overly complex test setups. For instance, in the **AppointmentServiceTest**, I reused a basic valid appointment object across multiple tests to keep the code clean and maintainable.

**Reflection**

**Testing Techniques Employed**

The primary testing technique I employed was unit testing. Unit testing focuses on verifying individual units or components in isolation. I used JUnit to verify object creation, updates, and deletions, as well as error handling in case of invalid input.

**Testing Techniques Not Used**

Other techniques not used include:

* **Integration Testing**: Testing how multiple components interact together.
* **System Testing**: Testing the complete system in a production-like environment.
* **Acceptance Testing**: Testing by the client to verify business needs are met.
* **Regression Testing**: Retesting existing functionality after code changes.
* **Performance Testing**: Measuring response times and resource usage under load.

**Practical Uses and Implications**

* **Unit Testing** is critical during development, especially in agile or test-driven development (TDD) workflows.
* **Integration Testing** becomes essential as the system grows, requiring validation of interactions between modules.
* **System Testing** is necessary before a release to ensure the system behaves correctly.
* **Acceptance Testing** helps ensure client satisfaction.
* **Regression and Performance Testing** are crucial for maintaining quality over time and under load.

**Mindset**

**Caution**

Throughout the project, I worked with caution. I carefully validated both valid and invalid input scenarios, appreciating that even small errors could cause larger issues. For instance, I made sure that the **AppointmentService** would not allow past dates, which is a critical constraint that affects business logic.

**Bias**

I actively tried to limit bias when testing my own code. I recognized the risk of assuming that "my code is correct" and instead approached testing with the mindset that bugs were likely present. For example, I purposely wrote tests that passed invalid data into the system to confirm that exceptions were properly thrown.

**Discipline and Avoiding Technical Debt**

Being disciplined about testing is essential for producing high-quality software. Cutting corners leads to technical debt, which means more bugs and more costly fixes later. To avoid technical debt, I plan to:

* Always write comprehensive unit tests.
* Perform code reviews, even self-reviews.
* Keep testing up-to-date with code changes.
* Use static analysis and coverage tools where available.

For example, in future projects, I will integrate tools like JaCoCo for code coverage tracking and use continuous integration pipelines to automate test runs.

**Conclusion**

This project strengthened my skills in writing focused, effective unit tests and thinking critically about testing strategies. It reinforced the importance of thoroughness, caution, and discipline in software testing, all of which will help me in my career as a software engineer.