**Summary and Reflections Report**

#### **Unit Testing Approach for Each Feature**

For each feature—**Contact**, **Task**, and **Appointment**—I used **JUnit** as the framework for unit testing. The approach focused on validating the behavior of each class and its interactions with other components. Here’s how I applied unit testing:

* **Contact**: The primary focus was on validating the field constraints (e.g., ensuring non-null values for name, email, and phoneNumber). My approach ensured that invalid inputs would throw exceptions, maintaining data integrity. Example: The ContactTest.java contains tests like

**assertThrows(IllegalArgumentException.class, () -> new Contact(null,**  **"**[**test@example.com**](mailto:test@example.com)**", "1234567890")).**

* **Task**: The Task class required testing for task creation, ensuring the immutability of the taskId, and validating the name and description fields. A key test case was to ensure that taskId could not be modified after initialization. Example: The TaskTest.java checks immutability by asserting that taskId is not changeable after creation.
* **Appointment**: The Appointment class tested field validation and immutability of appointmentId. It also tested date constraints to ensure that appointments could not be scheduled in the past. Example: In AppointmentTest.java, I used

**assertThrows(IllegalArgumentException.class, () -> new**  **Appointment("12345", invalidDate, "Past Appointment"))**

to ensure that past dates were invalid.

#### **Alignment with Software Requirements**

My testing approach was closely aligned with the requirements. Each class had clear rules (e.g., field lengths, null values, or immutable fields), and I tested these using valid and invalid inputs. The tests addressed boundary cases (e.g., descriptions exceeding 50 characters) and edge cases (e.g., task creation with null or invalid values), aligning with the software's specifications.

#### **Quality of JUnit Tests**

The quality of the JUnit tests can be assessed through **coverage percentage** and their effectiveness in validating business logic. While I didn't directly use a tool like JaCoCo to measure the exact coverage, I ensured thorough coverage by testing both normal and edge cases. For example, tests like assertThrows(IllegalArgumentException.class, () -> new Contact(null, "[test@example.com](mailto:test@example.com)", "1234567890")) ensure that invalid input scenarios are properly handled.

#### **Experience Writing JUnit Tests**

Writing JUnit tests was an integral part of the development process. I followed the TDD (Test-Driven Development) approach to identify potential issues early. Ensuring technical soundness involved checking constraints for each feature—ensuring that every exception thrown was appropriate for the given input scenario. For instance, in AppointmentTest.java, invalid dates were tested through assertThrows, guaranteeing that only valid dates were accepted.

For efficiency, I employed tests that were concise and reusable. For example, I tested boundary conditions with minimal code duplication, using validDate variables to test all features with similar date validation rules.

### **Reflection**

#### **Testing Techniques**

* **Unit Testing**: This was the primary technique used. Unit testing focuses on testing individual components in isolation. For each feature, I created tests that verified the business logic (e.g., immutability, null checks) without depending on other system components. This ensured that each class functioned independently.
* **Boundary Testing**: I employed boundary testing to check limits like the maximum length for descriptions (50 characters) and appointment IDs (max of 10 characters). This technique is essential for verifying edge cases.

Other techniques I did not use:

* **Integration Testing**: This would have tested the interaction between different components, which wasn't necessary in this project as the focus was on unit testing individual classes.
* **End-to-End Testing**: I didn't implement this as the focus was on isolated functionality rather than a full workflow.

In future projects, these techniques may be beneficial when verifying the integration of these services with other systems or user interfaces.

#### **Mindset**

When writing tests, I maintained a **cautious** mindset, recognizing the importance of ensuring the stability of each component before progressing. Testing the individual features thoroughly made me appreciate the complexity of the system and how even minor changes could have broader impacts.

In terms of limiting **bias**, I focused on testing each feature as if I were a new developer unfamiliar with the code. This avoided assumptions based on my knowledge and helped identify hidden bugs. A potential issue with bias when testing one's own code could involve overlooking corner cases or failing to see errors that seem obvious when looking at someone else’s code.

Finally, discipline in **commitment to quality** is crucial. I made sure that every function had a corresponding test case and avoided rushing the process to finish quickly. By maintaining quality from the start, I ensured the code was robust and avoid future technical debt. In real-world scenarios, neglecting proper testing or code structure can lead to more significant problems down the line.

### **Conclusion**

In this project, I implemented a comprehensive unit testing strategy that validated the core functionality of the Contact, Task, and Appointment services. This process helped identify potential issues early on and ensured that the application’s back-end components operated correctly. By maintaining a disciplined, cautious, and unbiased approach to testing, I ensured the integrity and reliability of the system. Moving forward, I aim to continue applying these principles while expanding my testing techniques to cover integration and end-to-end scenarios for more complex systems.

### **References:**

* Beck, K., & Gamma, E. (2004). *Test-driven development: By example*. Addison-Wesley Professional.