**Summary:**

This report explores my unit testing approach for the Contact, Task, and Appointment Services, discusses the approach as it aligns with the software requirements, defends the quality of JUnit tests, and shows my experience writing the tests.

Unit Testing Approach for Each Feature:

* Contact Service: I conducted unit testing by creating test cases to validate contact creation, updates, and deletions, involving both valid and invalid inputs.
* Task Service: I designed unit tests to cover task initialization, additions, updates, and deletions.
* Appointment Service: My unit testing involved validating appointment creation, handling invalid inputs, and verifying appointment date validity.

My unit testing approach closely aligned with software requirements. For instance, in the Contact Service, tests for adding, updating, and deleting contacts confirmed compliance with CRUD operations. Similarly, the Task and Appointment Services' tests validated their related functionalities.

The quality of JUnit tests is demonstrated by the level of coverage achieved. The complete range of test cases makes sure that many paths and cases are addressed. Coverage tools indicated that each class, method, and critical branch was thoroughly tested, providing a high level of certainty in the tests' efficiency (Kanai, 2022). For each of the Contact, Task, and Appointment Service, the coverage was 80% or higher.

JUnit's structured framework helped organize test design. To guarantee technical soundness, I used assertions to validate specific expectations. For instance, in the Contact Service, the assertion **assertEquals("1234567890", contact.getContactID())** confirms the correct contact initialization.

The technical soundness of the code was validated by verifying that the results of the tested methods matched expected results. In the Task Service, the line **Assertions.assertEquals(task, retrievedTask)** confirms task addition. Efficiency was maintained by focusing on each of the code segment. For example, the use of **taskMap.containsKey(taskId)** in the Task Service's deleteTask method avoids unnecessary iterations.

**Reflection:**

I utilized positive and negative testing, boundary testing, and edge case scenarios to make sure I had full coverage. Each technique was chosen based on the possible risks associated with the corresponding services.

I did not utilize performance testing and stress testing due to the project focusing on the functionality of back-end services (Cohen, 2022).

To limit bias, I approached the code playing the role of both a developer and tester. This practice helps in identifying possible issues that may be overlooked when reviewing the code. In the Appointment Service, the line **Assertions.assertEquals(appointmentID, appointment.getAppointmentID())** demonstrates unbiased verification.

Dedication to quality is vital in software engineering. Cutting corners can lead to defects, negatively affecting user’s trust and satisfaction.

To avoid technical debt, I plan to utilize automated testing in my development process, conduct regular code reviews, and continue education.

In conclusion, the unit testing approach for the Contact, Task, and Appointment Services was carefully planned and executed, and closely aligned with software requirements. The quality and coverage of JUnit tests provided a high level of confidence within my code. The experience of writing these tests and using several testing techniques was very helpful, allowing for better code quality. The mindset of caution, limited bias, and dedication to quality are necessary traits for a software engineer professional. This will help ensure quality code is being developed.

**References:**

Cohen, N. (2022, June 29). Performance Testing vs. Load Testing vs. Stress Testing. BlazeMeter.

Kanai, S. (2022). JUnit: A Complete Guide. Headspin. <https://www.headspin.io/blog/junit-a-complete-guide>