Summary & Reflections Report

I utilized an organized unit testing strategy for the Contact, Task, and Appointment services in the SNHU mobile application project. JUnit tests were used to independently test each service to make sure its functionality complied with the established requirements. For example, the ContactServiceTest made sure that incorrect inputs, including null or duplicate contact IDs, were handled correctly while validating the addition, updating, and deletion of contacts. Similar to this, the TaskServiceTest made sure that tasks could be created, modified, and removed efficiently. It also made sure that task names and descriptions didn't go over the allotted length. In order to make sure that null inputs or invalid dates triggered the proper exceptions, the AppointmentServiceTest tested features like altering appointment dates and descriptions.

The software requirements were closely followed by my testing strategy. For instance, tests like testAddContactWithExistingContactId verified that no duplicate IDs were permitted because the Contact service required unique contact IDs. In a similar manner, testUpdateAppointmentDateWithInvalidAppointmentId and testUpdateAppointmentDate were used to thoroughly test appointment requirements, including future appointment dates, in order to verify correct validation.

The high coverage percentage attained during testing (86%) demonstrated the efficacy of my JUnit tests. For example, tests addressed a range of edge cases, including duplicate entries, improper lengths, and null inputs. The application's ability to handle border cases effectively was confirmed by tests such as AppointmentDateCannotBeInThePast and AddressCannotBeMoreThanThirtyCharacters.

Writing JUnit tests required a lot of learning on my part. By adding particular validations, including length limitations and null checks, I made sure the code was technically sound. By throwing exceptions for too many characters, the test TaskNameCannotBeMoreTwentyCharacters, for instance, effectively verified task name length. By combining related validations and utilizing reusable setup techniques, code efficiency was preserved. To improve readability and reduce redundancy, the setUp method, for example, initialized items such as task, contact, and appointment.

I implemented techniques like functional testing, exception testing, and boundary testing. In order to make sure that inputs like names and descriptions stayed within their bounds, boundary testing was utilized for features like length limitations (e.g., FirstNameCannotBeMoreThanTenCharacters in Contact tests). Exception testing confirmed that errors, such as null or duplicate entries raising exceptions, were handled correctly. Core functions, including adding and retrieving records, were verified to function as intended through functional testing.

Integration and performance testing are other methods that are not employed. Performance testing assesses the system's performance under load, something that these unit tests did not need to do (Sawant, Bari, & Chawan, 2012, pg. 983, section 2.2.2). Integration testing, which is more useful later in the development process, verifies how various components cooperate (Chan, Chen, & Tse, 2002, pg. 1).

Every technique has its applications. While exception testing guarantees reliability against invalid inputs, boundary testing is essential for input validation. Functional testing is essential for fundamental development stages since it guarantees core functionality. System-level evaluations are more appropriate for performance and integration testing.

I maintained a vigilant mindset throughout the process, stressing prudence and close attention to detail. For instance, I made sure the application was resilient to erroneous inputs by carefully crafting tests to address edge cases like AppointmentDateCannotBeInThePast and TaskDescriptionCannotBeMoreThanFiftyCharacters. It was essential to understand the code's intricacy and relations. For example, in order to preserve consistency when altering an appointment's date, it was necessary to comprehend how the service interacted with the validation logic.

I critically assessed the code as though I hadn't developed it in order to reduce bias. For instance, in order to make sure the code handled mistakes efficiently, I purposefully created tests to fail at first, such as entering erroneous inputs for task IDs. Because developers can presume correctness without thorough validation, biases in testing one's own code might result in oversight.

Avoiding technical debt requires discipline and a dedication to quality. Taking shortcuts when testing could leave hidden problems that affect subsequent development. During testing, for example, possible runtime errors were avoided by making sure all tests thoroughly tested exceptions. In the future, I intend to maintain high standards for quality and ensure sustainable development methods by thoroughly documenting test cases and doing frequent code reviews.

References

Chan, W. K., Chen, T. Y., & Tse, T. H. (2002). An overview of integration testing techniques for object-oriented programs. *In Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science* (ICIS 2002) (pp. 696-701).

Sawant, A. A., Bari, P. H., & Chawan, P. M. (2012). Software testing techniques and strategies. International Journal of Engineering Research and Applications (IJERA), 2(3), 980-986.