**Summary**

For each of the three services, JUnit tests were written to ensure proper creation of contacts, tasks and appointments. I made sure to try and adhere to all of the requirements by systematically working through them. Starting with the Contact Service, each contact has unique IDs and enforced constraints on fields like firstName, lastName, phone, and address. The tests validated requirements like firstName and lastName adhered to the required ten character limit, and the phone met the exact length of ten digits. For the Task Service, tests ensured task creation also followed the specified requirements, including unique taskId, a name with a maximum length of twenty characters, and a description field. Tests also validated the name and description field limits. Finally for the Appointment Service, JUnit tests focused on validating that appointment IDs were unique, dates for appointments were in the future, and fields were not null. For example, in the Contact Service, the requirement that the contact service shall be able to delete contacts per contactId was strictly tested, as seen in the following test:

@Test

public void testDeleteContact() {

Contact contact = new Contact("1234567890", "Tom", "C", "1234567890", "123 Street St");

contactService.addContact(contact);

contactService.deleteContact("1234567890");

assertNull(contactService.getContact("1234567890"));

}

This method ensured the program strictly adhered to the constraint. Each service had similar tests designed around the constraints specified in the requirements. The quality of the JUnit tests was measured by code coverage, ensuring all key features were tested, including boundary conditions such as character limits. For example, this test ensures that the taskId field meets the ten digit requirement:

@Test

void testTaskIdTooLong() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

new Task("12345678901", "TaskName", "Valid description.");

});

assertEquals("Invalid ID", exception.getMessage());

}

This method helped ensure high test coverage and effectiveness by rigorously testing valid and invalid input scenarios. At the end, all of the services had code coverage of above eighty percent, not only adhering to requirements but also proving they were effective.

To ensure technical soundness, the tests were written with focus on edge cases, like exceeding character limits or handling null inputs. Efficient code was achieved by avoiding redundancy in test cases. For instance, in testing constraints for appointments:

@Test

public void testAppointmentIDNull() {

assertThrows(IllegalArgumentException.class, () -> new Appointment(null, new Date(), "Appointment Description"));

}

@Test

public void testAppointmentDateNull() {

assertThrows(IllegalArgumentException.class, () -> new Appointment("1234567890", null, "Appointment Description"));

}

I avoided repeating tests by reusing methods across different test cases for input validation, focusing on representative scenarios to maintain coverage while reducing excess code.

**Reflection**

During this project, I employed several software testing techniques to ensure its quality. One of the key techniques I used was boundary testing, which focused on verifying whether field values adhered to the specified length limits. This method was particularly useful in testing edge conditions, like checking whether string lengths exceeded the maximum limits. Another technique I utilized was black-box testing, which involved testing the application’s functionality strictly based on the software requirements. This approach allowed me to focus solely on the inputs and expected outputs, without considering the internal workings of the application. For instance, in testing appointment dates, I ensured that all dates were set in the future, based purely on input-output behavior rather than the logic behind the date-setting functionality.

There were also some testing techniques that I did not employ in this project. One of these was white-box testing, which examines the internal structure and logic paths of the code. While white-box testing can be beneficial for validating complex logic, my focus in this project was more on verifying input/output behavior rather than inspecting the inner workings of the code. Additionally, I did not perform integration testing, which would be useful for ensuring that different components of the application, such as the Contact and Task services, interact properly. My focus was on unit testing individual services, so integration testing was not a priority for this specific project.

Each of these techniques has practical uses in different software development situations. Boundary testing is especially helpful in applications that require strict input validation, where users might input invalid or extreme data. Black-box testing, on the other hand, is particularly suited for testing whether an application behaves as expected, based on the requirements, without delving into how the functionality is implemented.

Throughout the project, I adopted a cautious mindset to ensure all requirements were fully validated. This was especially important given the complexity of various parts of the code. While writing the tests, I had to ensure that all of the requirements would correctly flag errors. Even minor oversights in these tests could lead to bugs in the application. Additionally, I took time to limit bias in my review of the code by rigorously testing edge cases that might initially seem redundant or unlikely. As a developer, there is always the risk of bias when testing your own code. The familiarity with the implementation can lead to assumptions about its correctness which may not be correct. To avoid this, I tested all scenarios, including those I believed were less likely to occur.

Cutting corners during testing can easily lead to technical debt and introduce issues later on in the development process. For instance, skipping tests for null values or improper inputs might save time initially but would likely cause significant problems down the road. To avoid this, I made sure to write comprehensive tests that covered all code paths. I also regularly refactored my code to remove redundancies, and leveraged automation where possible to keep the quality at an acceptable threshold. Moving forward, I plan to avoid technical debt by continuously refactoring my code, ensuring proper documentation is in place, and implementing robust testing practices that account for both common and edge cases. In the long run, regardless of any project I work on, having this disciplined mentality will greatly improve the quality and maintainability.