The unit testing focused on the main parts of the program, like creating, adding, deleting, and retrieving appointments. The tests for the Appointment class made sure important details, like appointment IDs, dates, and descriptions, were checked for errors. The tests like testInvalidAppointmentId and testInvalidAppointmentDate caught mistakes like bad IDs or using past or empty dates. Tests for the AppointmentService class made sure the system worked for everyday tasks and handled problems like trying to delete appointments that don’t exist or adding duplicate IDs. These tests were created to match how people would actually use the program while also checking unusual or tricky cases.

The testing was designed to match the software’s requirements for managing appointments. It checked that all the key features worked as expected. The testAddDuplicateAppointment make sure that every appointment ID was unique, which was a requirement in the project. This make sure no duplicate IDs could be added to the system. Another test, testDeleteAppointment, confirmed that appointments could be removed correctly and that no leftover data or references were left behind in the system. These tests looked at how the system would be used normally and also prepared for situations where things might go wrong. By doing this, the tests not only met the project’s requirements but also covered extra scenarios to make the system even more reliable.

The JUnit tests were effective because they covered both normal situations and rare edge cases. These tests checked for unexpected inputs, like null values, invalid IDs, and past dates, to make sure the program handled them correctly. The testInvalidAppointmentId tested for null IDs and IDs that were too long to confirm the system responded properly by throwing an IllegalArgumentException. This made sure that the program could deal with incorrect or unexpected inputs without breaking. By being thorough, the tests validated that the program was fully ready for release and worked reliably under all kinds of conditions. This level of testing made sure the system had the ability to handle both everyday use and unusual scenarios.

Writing the JUnit tests showed the need to consider both everyday use and rare situations. The testDeleteAppointment simulated deleting an appointment set for a future date, confirming it was removed correctly without causing issues in the system. Creating the tests showed how important it is to think carefully about different input scenarios, including those that might not happen often. This supports the importance of preparing the system to handle both expected and unexpected situations reliably.

I made sure the code worked properly by adding strict checks that confirmed the input was valid and the program worked as expected. One of these checks made sure that appointment IDs were not missing or null. “assertThrows(IllegalArgumentException.class, () -> new Appointment(null, futureDate, "Description"));” This line made sure that invalid inputs were caught right away, making sure the system was reliable. Each test case was designed with this in mind, checking that all conditions matched the expected behavior.

I made sure to focus on efficiency by keeping test cases simple and avoiding extra complexity but still keep them clear. The testAddAppointment method is straightforward: “if (appointmentId == null || appointmentId.length() > 10) { throw new IllegalArgumentException("Appointment ID cannot be null or longer than 10 characters."); }” This test checked the functionality easily without any unnecessary steps, making sure resources were used efficiently.

The testing process used boundary testing and exception checking to evaluate the program's reliability. Boundary testing verified that the program could handle input values at or near their limits, such as the maximum length allowed for an appointment ID. Exception checking made sure the system could handle invalid actions, like trying to delete an appointment that doesn’t exist.

Stress testing and performance testing were not included in this project because they were not relevant to how the application is expected to be used. Stress testing checks how a system performs under heavy use or with large datasets, while performance testing measures response times and resource usage. Although these tests were not needed for this project, they are important for applications designed to handle high traffic or large amounts of data.

Boundary testing is especially useful for programs that require strict input validation, like forms collecting sensitive user data. Exception checking makes sure systems respond predictably to errors, helping prevent crashes or unexpected behavior. If stress testing were used, it could evaluate scalability, such as how an appointment system handles hundreds or thousands of entries at the same time.

I also had to make sure that the connections between different parts of the code didn’t cause unexpected problems. Such as when testing the testDeleteAppointment function, we made sure that deleting one appointment didn’t accidentally impact other appointments in the system. Using checks such as, “assertThrows(IllegalArgumentException.class, () -> service.deleteAppointment("NonExistentID"));”, makes sure that related parts stayed unaffected.

To reduce bias, the tests covered less obvious scenarios that might be overlooked, such as attempting to add an appointment with an ID that already existed. This made sure the system was tested from multiple perspectives and avoided assumptions about typical usage patterns. On the developer side, bias could occur when testing their own code, as familiarity might lead to missing edge cases. Using external testers or conducting peer reviews is a well-established way to address this issue.

Maintaining strict discipline during testing makes sure we did not take shortcuts, which helped reduce the risk of technical debt in the future. The Appointment class constructor validated all inputs to make sure they met the required criteria: “if (appointmentId == null || appointmentId.length() > 10) { throw new IllegalArgumentException("Appointment ID cannot be null or longer than 10 characters."); }”

By applying validation from the beginning, potential issues and the need for rework were avoided. To further minimize technical debt, I plan to follow best practices, including regular code reviews, automated testing, and maintaining detailed documentation.