In Project One, my unit testing approach for the contact, task, and appointment services focused on ensuring thorough coverage and alignment with the software requirements. For the contact service, I implemented unit tests to validate the creation, updating, and deletion of contact entries, ensuring data integrity and proper handling of data inputs. These tests verified the uniqueness of contact IDs and proper handling of invalid input formats, aligning closely with the requirements. For the task service, unit tests confirmed the functionalities of adding, updating, and deleting tasks while ensuring tasks adhered to deadlines and priority levels. This approach was aligned with requirements as tests ensured tasks could not be created with past due dates or without required fields. Finally, for the appointment service, I tested the functionalities of adding, updating, and deleting appointments, with a focus on date validation and description length constraints. The tests were strictly aligned with the requirements, ensuring that appointment dates were not in the past and descriptions were within the specified length.

The quality of my JUnit tests can be defended based on their effectiveness and coverage percentage. The tests provided comprehensive coverage of all key functionalities and edge cases, ensuring that the software behaved as expected under various conditions. The high coverage percentage indicated that nearly all branches and lines of code were executed during testing, highlighting the thoroughness of the tests. Writing JUnit tests involved a structured process that required careful consideration of both normal and edge cases. To ensure the code was technically sound, I employed assertions to verify the outcomes of each method, such as assertEquals("1234567890", appointment.getAppointmentID()) to check that the appointment ID was correctly set. Efficiency was ensured by writing concise and focused test methods that addressed specific functionalities, such as testAddDuplicateAppointment() which succinctly checks that the service prevents duplicate entries.

During this project, I employed several software testing techniques, including unit testing, boundary testing, and negative testing. Unit testing involved testing individual components in isolation to ensure they functioned correctly. This technique was used extensively to test the core functionalities of the contact, task, and appointment services. Boundary testing checked the limits of input values to ensure the system handled edge cases properly, helping to identify off-by-one errors and ensuring the system could handle extreme input values without failure. Negative testing involved providing invalid inputs to ensure the system handled them gracefully, improving the robustness of the system by ensuring it could handle erroneous conditions without crashing.

There were also many experimental methods that I did not use for this project. Integration testing focuses on the interaction between units or combinations of components, tests multiple components together to identify problems with connections and connections System testing checks that the overall integrated system meets specified requirements, and it ultimately validates the internal business process. Acceptance testing determines whether the system meets acceptance criteria or is ready for delivery to the end user, providing final certification before the software runs Performance testing examines the speed, responsiveness and stability of the system under the assigned task to ensure that the system can handle the expected load conditions.

Throughout this project, I adopted a cautious mindset, rigorously checking all edge cases and ensuring comprehensive test coverage. This cautious approach was vital in identifying potential issues that could affect the system's reliability. Appreciating the complexity and interrelationships of the code was crucial, as overlooking these aspects could lead to undetected bugs. For example, testing how appointment dates interacted with the current date ensured that past dates were identified correctly as “invalid”. To limit bias, I critically reviewed my code and tests, writing tests from a user's perspective to identify scenarios I might have missed as the developer. Bias can be a concern when testing one's own code, as developers might unconsciously avoid testing scenarios they believe are correct. By writing tests that intentionally included invalid inputs, I ensured a more unbiased and thorough validation.

Being disciplined in your commitment to quality is important in software engineering. Cutting corners can lead to technical costs, compromising the long-term maintainability and reliability of the system. To avoid technical costs, I plan to follow best practices such as writing extensive tests, conducting regular code reviews, and constantly revising code to improve its quality in order to ensure that all new products will include corresponding unit testing to help maintain high standards. By taking a disciplined approach and using advanced testing methods, I aim to deliver high-quality software that meets the needs of the user and stands the test of time.