In my testing approach, I aligned with the specific requirements to ensure that each was thoroughly fulfilled. I focused on testing edge cases, such as handling duplicate entries and scenarios involving missing IDs, to guarantee that the software behaved as expected.  
 I ensured high coverage by creating tests for valid and invalid inputs, including null entries and overly long names and descriptions. The coverage percentage indicated that most code paths were tested, which gave me confidence in the robustness of the tests.

A screenshot of a graph

Description automatically generated

Figure 1 - Coverage Contacts

A screenshot of a graph

Description automatically generated

Figure 2 - Coverage Tasks

I ensured my code was technically sound by carefully reading it and following all the steps manually. After my Contact assignment was graded, I realized I needed to be more cautious with my reviews. One example of how I made the code technically sound is the testAddDuplicateContactId; the same is true for the tasks, testAddDuplicateTaskId. It tests if an existing object with that ID already exists, which should not because IDs are unique.

A screenshot of a computer

Description automatically generated

Figure 3 - code for testing duplicate IDs

In my task service, instead of duplicating logic within the constructor, I used setters to handle the logic for assigning values to the fields. For example, the setter method setTaskName includes validation to ensure the task name is not empty, preventing duplicating the validation logic inside the constructor. This approach improves the maintainability of the code by centralizing the logic and reducing redundancy.

A computer screen shot of white text

Description automatically generated

Figure 4 - Constructor using the setters methods

My use of software testing methods included unit testing using JUnit, assertion-based tests, and boundary tests. The focus of unit testing is to confirm individual sections separately, like certain methods or classes. These tests make sure that every section functions as anticipated without needing other systems or modules. For example, I checked the making of “Task” and “Appointment” objects, checking that methods and constructors give the right exceptions when given with wrong inputs. By using assertions to match expected results with actual outcomes in these tests, a technique called assertion-based testing. For instance, by applying arguments like assertEquals to confirm method outputs or assertThrows to check incorrect entries, it precisely proves if the application's logic operates as planned. Furthermore, we utilized boundary testing to confirm that inputs like the length of strings don't go beyond set limits. For instance, task names or descriptions of appointments exceeding character limitations are properly managed by throwing exceptions.

There were many other testing methods I didn't use, such as integration testing, system testing, end-to-end testing, and mock testing. Integration Testing is very important because it checks how different parts connect and function together; usually, this tests things like APIs databases or modules that talk to each other. System testing looks at the whole system to make sure all pieces fulfill what they are supposed to do; normally, this occurs when everything in the system is fully connected. Testing from start to finish imitates the activities of users in the system so that it can be made sure that it acts as anticipated across full application stack, from front end till back end. Testing using mock includes utilization of mock objects or dependencies for separating unit being tested away from its outside services. This makes tests concentrate only on inside logic of component with no need for real external systems like databases.

For projects that require verification of accuracy at the method or class level, unit testing is very useful. It shows its highest effectiveness in initial development stages or when there's a need for continuous integration because it offers speedy feedback on small modifications to the codebase. However, unit testing doesn't address larger scale integration problems and mistakes may still occur when numerous components come into interaction. Integration checking assists in intricate tasks where multiple services and modules should operate correctly together. This type of check helps to find problems coming from interaction between different components, making sure they work in harmony. System examination is a crucial step that verifies if the system fulfills given requirements, usually conducted just before launching it. While this method gives a complete understanding of how the system works, it can take a lot of time because all parts of the system must work. End-to-end testing is useful in applications where pretending user interactions are vital to make sure users have an easy-going experience. It confirms that the application behaves as expected from viewpoint of users. In the end, mock testing is useful in service-orientated structures where parts rely on outside resources like databases. By making imitation of these dependencies, I can quicken the test process and make easier the test setting. Nevertheless, there might be a danger of not detecting mistakes that could appear in real-life situations.

Throughout this project, I adopted a cautious and detail-oriented mindset, fully aware that small mistakes could lead to larger issues in the functionality. As a software tester, I was particularly mindful of edge cases. For example, I focused on identifying errors that might arise from duplicate entries and ensuring proper handling of null inputs. Caution was essential because understanding the complex interdependencies within the code helped me prevent cascading failures. When testing the task and contact services, it was clear that even slight oversights could disrupt the assignment of unique IDs, affecting other components like task scheduling or contact management.

To limit bias in my code review, I cross-checked my logic multiple times, treating every section as if it was unfamiliar to me. I consciously avoided assuming the correctness of code simply because I had written it. For example, when testing the testAddDuplicateContactId and testAddDuplicateTaskId functions, I simulated potential user behavior to verify that the validation processes were robust. On the developer side, I imagine bias could become a significant challenge if I were solely responsible for testing my code. Developers might overlook subtle issues, especially when too familiar with the logic. A specific example is my use of setters for validation. Initially, I was inclined to copy the logic into the constructor, which could have introduced redundancy and future bugs.

Maintaining a disciplined approach to quality is crucial in software engineering, as cutting corners can lead to technical debt that accumulates over time. During this project, I committed to thorough testing by including both valid and invalid inputs, ensuring that exceptions were handled correctly. One way I plan to avoid technical debt as a professional is by adhering to best practices like writing unit tests for every new feature and continuously refactoring code to enhance maintainability. By doing so, I aim to prevent future issues that could arise from quick, untested fixes, especially in larger systems where undetected bugs can be costly to resolve.