For this project, I created unit tests for the Contact, Task, and Appointment services. My main approach was to write tests that directly matched the requirements from Project One. I reviewed each requirement and wrote at least one test to confirm that rule was followed. For example, the Contact service had a requirement that the contact ID could not be longer than 10 characters. To meet this, I wrote a test to make sure IDs with more than 10 characters were rejected. Another rule said the first name and last name could not be longer than 10 characters, so I wrote tests to check both valid and invalid names. This process helped me make sure the code followed the requirements exactly.

I also made sure to test more than just the “happy path.” While it was important to confirm that the code worked with normal input, I also tested incorrect inputs, such as null values, empty strings, or fields that were too long. For example, in the Appointment service, I tested what would happen if I tried to create an appointment with an empty description. These types of tests showed me that the program could handle real-world situations, not just perfect input.

The overall quality of my JUnit tests can be supported by the coverage results. My tests covered all of the major methods and checked edge cases. The coverage percentage I achieved 81% , showed that most of the program’s code was reached by my tests. This gave me confidence that my unit tests were effective and that the application behaved the way it was supposed to in different situations.

**Experience Writing JUnit Tests**  
Writing JUnit tests gave me hands-on practice with making my code both technically sound and efficient. To keep the tests accurate, I used clear assertions that directly checked the result I expected. For example, in my Contact tests, I used assertEquals("John", contact.getFirstName()); to confirm that the method returned the correct first name. Simple assertions like this made my tests easy to read and easy to trust.

To make my code more efficient, I avoided repeating setup code. Instead of rewriting the same object creation in every test, I created helper methods to build Contact, Task, or Appointment objects. This way, each test could focus only on the specific rule it was supposed to check. For example, if I needed a new Task object, I called my helper method instead of typing out all the fields every time. This cut down on duplication and made my test suite easier to maintain. If something in the object creation needed to change later, I only had to update it in one place.

Overall, writing these tests taught me how to balance detail with simplicity. Each test needed to be specific enough to prove the requirement worked, but not so complicated that it became hard to understand. Keeping them short and clear helped me make tests that were useful both now and in the future.

**Testing Techniques**  
The main testing technique I used in this project was unit testing. Unit testing focuses on small pieces of code, usually one method at a time. These tests are fast to run and easy to repeat, which makes them great for catching mistakes early in development. Another technique I used was boundary testing. This means checking the limits of inputs, such as the maximum or minimum allowed values. For example, I tested IDs that were exactly 10 characters long to make sure they passed, and I tested IDs that were 11 characters long to make sure they failed. Boundary testing helped me confirm that the rules were applied correctly at the edges.

There were other techniques I did not use for this project but are important to know about. One of these is integration testing, which checks how different modules work together. For example, in a bigger system, integration tests might confirm that the Contact service and the Task service share data correctly. Another method is system testing, which looks at the entire program as one complete unit. System tests usually simulate how a real user would interact with the application. I also did not use regression testing, which is when you re-run all previous tests after making changes to confirm nothing new was broken. This is especially important in projects that go through many updates.

Each of these techniques has its place. Unit and boundary testing work well for small projects like this one, where the main goal is to confirm that individual rules are enforced. Integration and system testing are more useful for larger applications with many moving parts. Regression testing is critical when code is updated often, because it helps keep the software stable over time. Knowing when to use each technique is part of being a good tester.

**Mindset**  
When working on this project, I tried to take on the mindset of a tester instead of only a developer. As a developer, it’s easy to assume your own code works, but as a tester, you have to assume it might fail. I approached my tests with caution and asked myself what could go wrong in each situation. For example, I tested not only valid names for a Contact but also blank names and overly long names. Appreciating the complexity of the code meant realizing that even small rules, like a string length, could cause problems if ignored.

I also worked to limit bias when reviewing my own code. Since I was both writing and testing it, there was a risk of being too confident. To fight this, I wrote tests that I thought would break my code instead of just prove it worked. For example, I tested null values even though I normally would not enter them as a user. This mindset helped me find potential errors I might have overlooked.

Finally, this project showed me the importance of being disciplined about quality. It can be tempting to cut corners on testing to save time, but that only causes bigger problems later. Poor testing leads to bugs, rework, and technical debt. To avoid this, I plan to always write tests while I code, focus on edge cases, and keep my test suite updated as the application changes. By sticking to these habits, I can make sure my future projects are more reliable and easier to maintain.

**Conclusion**

This project helped me practice analyzing testing approaches and applying strategies that matched the requirements. By writing JUnit tests for the Contact, Task, and Appointment services, I showed how unit testing and boundary testing can confirm that each rule works as expected. I also learned the value of staying cautious, reducing bias, and committing to quality. While I did not use larger techniques like integration or system testing here, I understand their place in bigger projects. Overall, the project strengthened both my technical testing skills and my mindset as a software engineer.