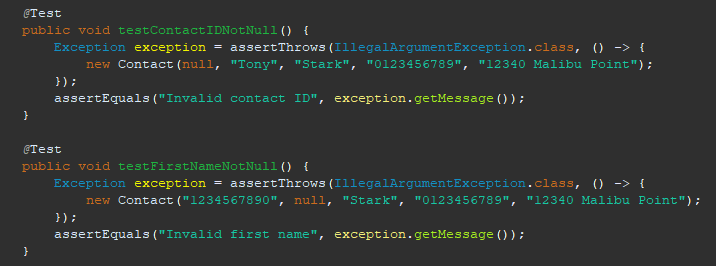
In this project, I primarily employed unit testing and boundary testing techniques. Unit testing involved testing individual components, such as the Contact, Task, and Appointment classes, in isolation to ensure they met the specified requirements. Boundary testing focused on testing the edges of acceptable input ranges, such as the maximum allowable lengths for string inputs and validating that appointment dates were not set in the past. My main goal was to ensure I had a high coverage of the inputs to minimize errors and misuse.

A screenshot of a computer

Description automatically generated

To ensure the Contact class input is valid, I implemented tests to throw errors for no input or for input that wasn’t in the correct format:



For the Appointment class, it was vital that any appointment set not be in the past. For this reason, I used a test to ensure the appointment added would be only in the future.

A screen shot of a computer code

Description automatically generated

In this way, I ensured my code aligned with the software requirements and was effective in making it technically sound.

While unit testing and boundary testing were the main techniques I used, there are other important software testing techniques that I didn’t implement in this project. These include integration testing, which is used to verify the interactions between different modules or services, and system testing, which involves testing the entire system to ensure it functions as desired. Additionally, acceptance testing, which focuses on validating the software against user requirements, would be vital in a real world scenario to ensure that the final product aligns with customer expectations.

Each of these testing techniques has specific practical uses and implications depending on the context of the software development project. For instance, integration testing would be particularly important in a project where multiple services or modules need to interact seamlessly, such as in a microservices architecture. However, oHosystem testing would be critical for ensuring that the entire application works as expected when all components are integrated. Acceptance testing is key in delivering a product that meets customer needs and ensuring satisfaction with the final outcome.

Throughout this project, I adopted a mindset of caution and thoroughness, particularly when it came to testing the various inputs and edge cases. This cautious approach was important because it helped me appreciate the complexity and interrelationships within the code. For example, by rigorously testing the Appointment class to ensure that past dates were not accepted, I was able to prevent potential scheduling errors that could have significant implications for users. Even after implementing all of my tests, I realize that there are still cases where exceptions or errors will not be caught by testing. Keeping that in mind, I wanted to minimize future workloads by catching as many as possible.

To limit bias in my review of the code, I focused on writing objective tests that were based on the specific requirements rather than assumptions about how the code should work. I recognized that testing my own code could introduce bias, so I tried to view the code from the perspective of an external tester. This helped me identify potential issues that I might have otherwise overlooked.

Being disciplined in my commitment to quality was a key focus throughout this project. I understood that cutting corners in writing or testing code could lead to technical debt, which would ultimately result in more work and potential issues down the line. To avoid this, I adhered to best practices in both coding and testing, ensuring that each component was thoroughly tested and validated. Moving forward, I plan to continue this disciplined approach by prioritizing quality and maintaining a strong focus on thorough testing to prevent the accumulation of technical debt.