Marissa Castellone

8/16/25

7-2 Project Two

To complete this project, I created unit tests for the contact, task, and appointment services. My unit testing strategy was created to make sure that the particular validation criteria needed by each of these services were appropriately applied. I created JUnit tests for the contact service to confirm that contacts could be properly added, modified, and removed. These tests verified that phone numbers and addresses were legitimate, that first and last names did not exceed character restrictions, and that the contact ID was unique. I developed unit tests for the contact, task, and appointment services to finish this project. To ensure that the specific validation requirements required by each of these services were applied correctly, I developed a unit testing strategy. To verify that contacts could be added, changed, and deleted correctly, I wrote JUnit tests for the contact service. These tests confirmed that the contact ID was unique, that first and last names did not exceed character limits, and that phone numbers and addresses were authentic.

This project was completed by  creating unit tests for the contact, task, and appointment services. I created a unit testing technique to make sure that the validation requirements needed by each of these services were applied appropriately. I created JUnit tests for the contact service to confirm that contacts could be added, modified, and removed appropriately.

These tests verified that phone numbers and addresses were real, that first and last names did not exceed character restrictions, and that the contact ID was unique. Unit tests for the contact, task, and appointment services were developed to finish this project. To ensure that the specific validation requirements required by each of these services were applied correctly, I developed a unit testing technique. To make sure that contacts could be added, changed, and deleted properly, I wrote JUnit tests for the contact service. These tests confirmed that the contact ID was unique, that first and last names did not exceed character limits, and that phone numbers and addresses were authentic.

Unit tests for the contact, task, and appointment services were developed for this project. To ensure that the specific validation requirements required by each of these services were implemented correctly, I developed a unit testing methodology. To make sure that contacts could be added, changed, and deleted properly, I created JUnit tests for the contact service. These tests confirmed that the contact ID was unique, that first and last names did not exceed character limits, and that phone numbers and addresses were authentic. Additionally, I kept my testing narrow and isolated, which increased their efficiency. Because each test simply looked at one behavior at a time, it was much simpler to identify failures. For instance, I created a test that was just focused on deletion and did not combine it with any other operation. The test suite remained efficient and well-organized due to this obvious division of responsibilities.

Reflection:

Black-box and boundary testing methods were the main ones I used in this project. I tested the services using black-box techniques, treating them as whole entities and concentrating exclusively on their inputs and outputs rather than their internal operations. This method helped guarantee that the program produced accurate results in a variety of scenarios by simulating how an end user may interact with the system. By paying close attention to edge circumstances like maximum string lengths, null values, and attempts to add duplicate IDs, I also relied on boundary testing. These examples are a crucial component of my test strategy since they frequently expose issues that are missed by standard "happy path" testing. Additionally, there were several methods that I did not apply in this project. Since the requirements focused on exterior behavior, white-box testing—which tests the code's underlying logic—was not required for these services. Additionally, I didn't use system testing, which examines the application, or integration testing, which confirms that various components function well together. Even if these methods weren't necessary in this instance, they are crucial for more extensive projects where many modules rely on one another, and I understand that they will be used in more complicated assignments in the future.

I approached the assignment with caution and attention to detail. I realized that even a minor validation error could jeopardize the system as a whole. For example, the application might enable users to schedule invalid events, reducing its utility, if I hadn't tested that appointment dates couldn't be in the past. I made sure that these guidelines were consistently adhered to by doing extensive testing. While reviewing my own code, I also tried to minimize prejudice. Although it is simple to believe that your code functions as intended, I purposefully created tests that were meant to break my own implementation. For instance, even though I thought my constructors handled null IDs and invalid phone numbers appropriately, I tested them. If I had simply tested for successful scenarios, I might have missed possible problems that this technique helped me find.

When engineers test their own code, bias can be a serious issue. You may unintentionally steer clear of testing cases that you anticipate failing as the code developer. To combat this, I made an effort to think like an outside tester and search for any vulnerabilities in the application. I increased test coverage and reduced bias by doing this. In software engineering, this discipline is essential. Taking short cuts during testing results in technical debt, which frequently causes far more serious issues down the road. For instance, omitting a string length validation test might not result in an immediate failure, but it may generate crashes or security problems in the production environment. It is significantly more expensive to fix such issues after release than it is to find them early using unit testing.

By adhering to best practices like creating tests for each new feature, taking test-driven development into consideration when practical, and taking part in frequent code reviews, I intend to avoid technical debt as a professional. I can deliver dependable software, lower the likelihood of regressions, and create systems that are simpler to maintain over time by practicing quality discipline. Writing thorough JUnit tests as part of the software development process, paying close attention to requirements, and having a strong testing mentality were all emphasized by this project.