During this project, I built and tested three core features for a mobile application's backend. Those being the Contact Service, Task Service, and Appointment Service. For all three, I focused on unit testing with the JUnit framework. My goal was to make sure each class and method worked on its own before even thinking about how they’d work together. Unit testing is often the first step in reliable development, and it’s considered best practice in Java-based systems (Oracle, n.d.).

Each milestone had rules to follow, like character limits on strings, required fields, or logic like checking if a date is in the past. I made sure my JUnit tests matched those requirements directly. For example, in the Appointment class, the date couldn’t be in the past. I wrote a test with a Calendar object set to yesterday, then used:

assertThrows(IllegalArgumentException.class, () -> new Appointment("A01", past.getTime(), "Description"));

This confirmed the constructor correctly rejected invalid input. The same logic was applied across the contact and task classes to validate names, descriptions, and IDs.

I think my tests were solid because they covered both the valid inputs and plenty of edge cases. For instance, I checked what would happen if fields were too long, null, or duplicated in the service classes. In TaskServiceTest.java, I had:

assertThrows(IllegalArgumentException.class, () -> service.addTask(task2));

to catch duplicate IDs, something that would otherwise break the system down the line.

Writing the tests got easier after the first milestone. I reused setup methods like @BeforeEach to save time and keep the tests clean:

@BeforeEach

public void setUp() {

service = new TaskService();

}

This way, I didn’t have to write the same code over and over for each test case. It also helped to keep everything more readable and organized, which is helpful when working with a lot of similar logic.

**Reflection**

**Testing Techniques**

The main technique I used throughout the project was unit testing, which is where you test small, sectioned parts of your code, like a single method or class, to make sure they work correctly. This is one of the most recommended approaches in both academic and industry settings. According to Microsoft (2022), unit testing improves reliability, makes it easier to maintain code, and reduces bugs early on.

Other techniques I didn’t use include integration testing, system testing, and acceptance testing. Integration testing checks how different pieces of code work together, such as making sure a class and a database play nicely. System testing goes further and tests the whole application from start to finish. Acceptance testing is often done by the customer to verify that the software works as expected from their point of view (Microsoft, 2022). Since my code didn’t involve any external systems like databases or UIs, unit testing made the most sense.

In real-world projects, you’d want to combine all these techniques. Unit tests catch issues early. Integration and system tests catch issues that only show up when different parts interact. And acceptance tests are the final step to make sure it meets real-world use.

**Mindset**

Going into this, I treated each assignment like a checklist of "what could go wrong?" instead of just assuming everything would work. That mindset helped me be more cautious with my testing. For example, I tested not just correct phone numbers but also inputs with letters, null values, and lengths over 10. It wasn’t about just getting it to work, it was about breaking it to see where it failed. That’s something I’ve learned is key in testing as if I don’t do it someone else will and it will need to be fixed regardless.

I also tried to reduce my own bias by treating the tests as their own thing, separate from the code I wrote. Instead of writing a test and assuming the code would pass, I wrote tests to prove the code could fail and that it properly handled bad input. As someone writing both the code and the tests, I know how easy it is to overlook bugs because you think you “wrote it right.” So, I approached the tests like I hadn’t seen the code before, and that helped me stay objective.

It’s easy to skip writing tests when you’re on a deadline or just want to finish. But skipping tests leads to bugs. I plan to avoid that by always writing unit tests when I write new features and taking the time to test edge cases, not just the obvious ones. Testing upfront now saves time and frustration later.

**References**

Microsoft. (2022). Unit testing best practices. Retrieved from https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Oracle. (n.d.). JUnit in the Java Platform. Retrieved from https://docs.oracle.com/javase/tutorial/java/javaOO/examples.html