**Project Two: Summary and Reflections on Unit Testing for Contact, Task, and Appointment Services**

Working on Project One as a software engineer for Grand Strand Systems gave me the opportunity to design and test core services—specifically contact, task, and appointment features—for a mobile application. The focus of this assignment was on unit testing, and more specifically, writing effective and thorough JUnit tests. This experience pushed me to really think about how to validate each feature, not just as a developer trying to make things work, but also as a tester trying to break things and uncover weaknesses.

For the **contact service**, my approach was to cover all the important behaviors a user might expect—adding a contact, updating their information, and deleting them. I started by writing tests that confirmed expected behavior when valid input was provided. For example, in my ContactServiceTest, I verified that updating a contact’s phone number to a valid 10-digit number worked correctly. Then I followed that up with negative tests—trying to insert an invalid phone number like "123" to ensure an exception was thrown. This aligned with the software requirements because the contact fields had strict validation rules: first and last names could not be longer than 10 characters, phone numbers had to be exactly 10 digits, and addresses couldn’t be over 30 characters. Every test I wrote was rooted in those constraints, and I aimed to cover all possible edge cases.

In the **task service**, I tested similar scenarios. Adding a task, updating its name or description, and removing it. Again, I leaned into the requirements: no null fields, names under 20 characters, and descriptions under 50 characters. I used both positive and negative tests. One test checked that a valid task with the ID "T123" and a description like "Fix login bug" was saved and retrievable. Another intentionally passed in a description with 51 characters to make sure the validation would throw an error. This level of testing helped ensure that my code didn’t just work in ideal conditions, but also in the weird edge cases where things tend to break.

The **appointment service** required a bit more caution because of the time-based logic. I had to test that appointments couldn’t be scheduled in the past. So I used Calendar to generate both future and past dates in AppointmentServiceTest. I verified that a future appointment could be created and retrieved, and I wrote a test to confirm that trying to create an appointment with a past date would throw an exception. I also checked that a defensive copy of the Date object was returned to avoid unintended side effects. These tests weren’t just checking boxes—they were simulating real-world usage and abuse. My approach aligned tightly with the requirements because the logic specifically disallowed null or past dates and limited appointment descriptions to 50 characters.

Looking at test quality, I feel confident that my JUnit tests were effective. All of my test classes ran green, with 100% pass rates, and I had coverage across all major features and logic branches. I didn’t just test the happy paths; I also covered invalid inputs, exceptions, and behavior boundaries. For example, in the AppointmentTest, I included a test to ensure that two identical appointment objects would be considered equal, using assertEquals(), which validates the equals() method was implemented correctly. These kinds of checks gave me peace of mind that the deeper logic worked as expected.

One technique I used throughout was **equivalence partitioning**—I grouped inputs into valid and invalid partitions and made sure to test at least one value from each. For phone numbers, that meant checking both a 10-digit number and a 9-digit number. I also used **boundary testing**, especially for fields with length constraints. For example, when the address field had a max length of 30, I tested with 30 characters, 31 characters, and null. These are simple yet powerful techniques that can catch bugs that don’t show up in basic happy-path testing.

There were some techniques I didn’t use, like **mocking** or **integration testing**. Mocking might’ve helped if I had external dependencies like databases or APIs, but since this project was isolated to in-memory services, it wasn’t needed. Integration testing also wasn’t in scope, but in a full production app, I’d want to test how the services work together, like making sure a contact’s tasks show up correctly in a dashboard view. That’s where techniques like system-level tests or UI tests would come into play.

Each of these techniques has a place. Unit testing (what I did here) is great for early-stage validation of logic. Integration and system testing are useful when multiple modules are wired together. And mocking is essential when your code depends on external systems you don’t want to hit during tests, like payment gateways or third-party APIs. If I were working on a banking app, for instance, mocking would be non-negotiable to simulate transactions safely.

As I took on the mindset of a software tester during this project, I realized how much discipline and caution it requires. You’re not just trying to make the code work—you’re trying to make sure it can’t break. I found myself asking, “What’s the worst thing a user could do here?” That’s how I caught issues like null inputs or unexpected field lengths. Appreciating the interrelationship between parts of the code helped too—like how an invalid phone number in a contact could cascade into problems if that contact was later used in another service. It’s like tightening every screw on a machine—you can’t just focus on the shiny parts.

I also tried to limit bias while testing my own code. That’s tough, honestly. It’s easy to subconsciously write tests that confirm your code works rather than challenge it. To combat that, I wrote tests before refactoring code, and I wrote them with the mindset of a stranger trying to prove me wrong. For instance, when I added a feature to update task names, I immediately tried passing in null and overly long strings just to be sure the setter logic held up. Writing tests like this reminded me how important it is to be humble about your own work.

Finally, this project reinforced how important it is not to cut corners in testing. It’s tempting to skip writing tests when the code “looks fine,” but skipping tests is like skipping seatbelts just because you haven’t crashed yet. Bad things happen fast when code goes unchecked. I plan to avoid technical debt by writing clean, well-tested code up front, documenting my logic, and refactoring early rather than later. For example, in the contact service, I initially wrote duplicate logic for validating fields, but I moved it to private helper methods like validateContactId() to keep things cleaner and easier to test.

In the end, this wasn’t just a coding project—it was a mindset shift. I walked away with a better appreciation of what it takes to write reliable software. I know that bugs will always find a way to sneak in, but with the right tests in place, at least I can say I did everything I could to stop them.