Nicholas Liousas

6/21/2025

CS 320

Project Two Summary & Reflections Report

Part 1: Summary

Throughout the project, I used unit testing to check the Contact, Task, and Appointment features of the mobile application. I created separate JUnit test files for each class and tested both successful object creation and edge cases such as null values, string length violations, and duplicate IDs. My main testing approach was to isolate individual behaviors and confirm whether each rule in the requirements was followed by the class logic.

One feature that was harder to test than the others was the appointment date in the Appointment class. I had trouble at first getting the past-date logic right, but that was partially user error. Once I figured out how to compare the appointment date with the current system date, things started to click. My contact phone validation also helped me understand how to properly check for an exact 10-digit format using string pattern matching.

My tests were closely aligned with the project requirements. For example, I wrote tests to make sure that invalid inputs would throw the correct exceptions, and that valid inputs would be accepted. In the Appointment test file, I included a test to confirm that entering a past date would cause the constructor to fail, which followed the requirement that appointments can’t be scheduled in the past. I also used true/false checks in my service test files to confirm that adding or deleting based on unique IDs worked properly.

I believe my JUnit tests were effective because I tested both normal use and edge cases across all features. I covered valid input, missing input, and inputs that broke field length rules. Honestly, I’m proud of how the whole appointment service test file turned out. I didn't know what to expect while writing tests, but by the end of the project it felt more natural.

At first, writing JUnit tests felt confusing, especially when I wasn’t sure what counted as “enough.” But by Module 5, it started to make more sense. I realized that each validation rule needed its own test case, and once I started thinking of it that way, everything became easier to follow along.

I made sure my code was technically sound by sticking to clear validation rules and meaningful test cases. For example, I tested that null values were properly rejected, and that fields couldn’t go over their character limits. I also kept my code efficient by using simple logic to loop through lists and match IDs before performing updates or deletions. I felt that this approach was both readable and good enough for the scale of the project.

Part Two: Reflection

The main testing techniques I used were unit testing and white-box testing. Since I was familiar with how the code worked, I was able to write tests that focused on specific conditions like invalid input, character limits, and making sure errors were handled the right way. White-box testing helped me check that each rule in the assignment was actually being followed by the code.

Some testing techniques I didn’t use were black-box testing and integration testing. Black-box testing is when you test the inputs and outputs without knowing how the code works inside. Integration testing checks how different parts of a system work together, like how a database and user interface might connect. Since this project didn’t involve a database or UI, those techniques didn’t really apply.

That being said, I can see where those other testing techniques would be useful. Unit testing is great during early development, especially when building core logic. Black-box testing would be more useful if someone else was testing my work without seeing the code. Integration testing would matter more in large-scale projects where multiple parts need to connect smoothly.

I tried to be cautious in my testing mindset because I knew that small issues in logic could create bigger problems. For example, if I missed a null check in the Contact or Task class, it could make the whole service fail. I appreciated how each part of the project was connected to other parts, so I tried not to take any shortcuts.

To avoid bias, I tried not to assume that my own code was already working correctly. I asked myself what would break the code instead of only thinking about what would work. If I had been testing someone else’s code, I know I would’ve looked at it more critically, so I tried to use that same mindset for my own work.

Finally, this project reminded me why it’s important not to cut corners with testing and code quality. It’s tempting to not go through all the steps, especially when the code already seems to be working, but that’s how bugs get missed. In the future, I plan to avoid technical debt by continuing to write tests as I go, not after the fact, and by keeping my logic clear and easy to follow even if it takes a little more time.